Department of Computer Science
Course number: CSC.T341

コンピュータ論理設計
Computer Logic Design

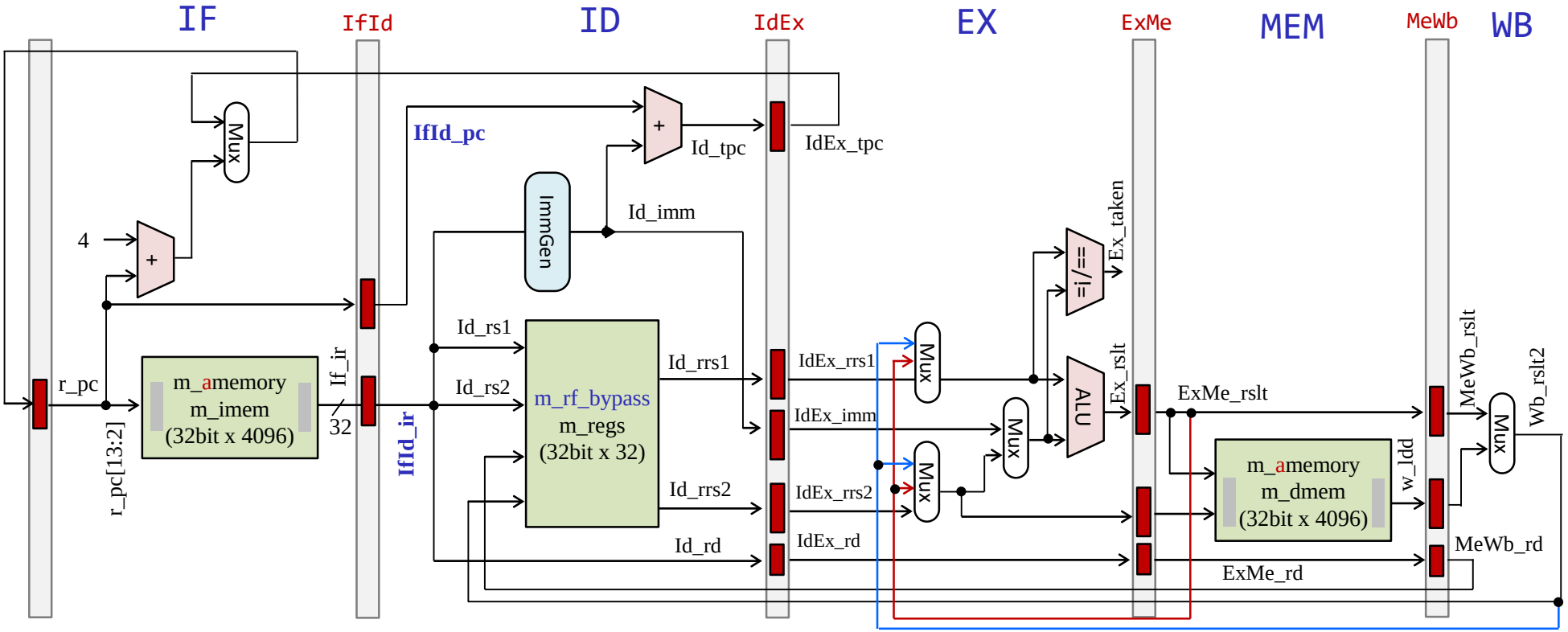
13. Supplemental explanation,
and Preparing for the design contest (group work)

吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
講義：月曜日 10:45-12:25, 木曜日 10:45-12:25

Ver. 2023-05-25a

m_proc20 5段のパイプライン版

- add, addi, sll, srl, lw, sw, beq, bne命令に対応したプロセッサ（データフォワーディング有り）
- IF, ID, EX, MEM, WB のそれぞれをステージとする5段のパイプラインプロセッサ
- ステージ IF と ID の間のパイプラインレジスタには IfId_ から始まる名前を利用する.
- ステージ ID で生成される配線には Id_ から始まる名前を利用する.



code190.v（code185.v を参考に自分で実装する）

アナウンス

- ACRiルームに，高速なコンピュータを導入しました．
- 次のマシンが高速です．空いていれば，ここから選んで使いましょう．
  - vs001 ～ vs010
  - vs101 ～ vs110
  - vs701 ～ vs710
  - vs801 ～ vs810
  - vs901 ～ vs910

第1Q授業学修アンケート実施

- T2SCHOLAにログインして，授業学修アンケートに記入してください．
- 教員側からは回答数のみを確認でき，誰が回答したかはわかりません（匿名性が担保されています）．

Department of Computer Science
Course number: CSC.T341

2023年度の講義と演習は，学術国際情報センター3階 情報工学系計算機室で実施します．

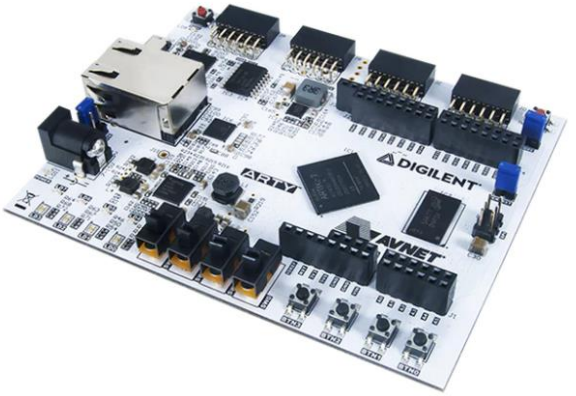
コンピュータ論理設計
Computer Logic Design

1. コンピュータシステムの基本構成
Basic Structure of Computer Systems

吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
講義：月曜日 10:45-12:25, 木曜日 10:45-12:25

Ver. 2023-04-08a

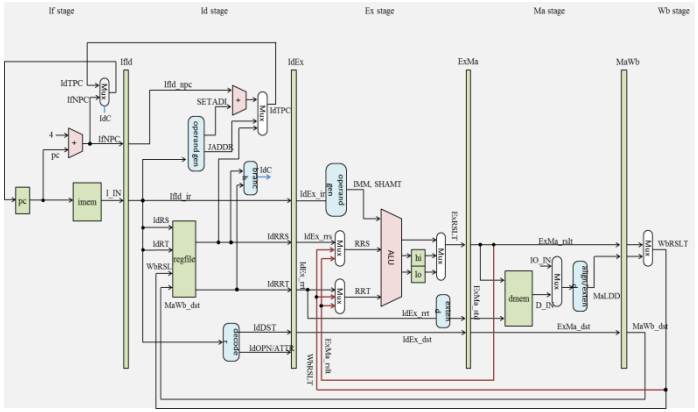
コンピュータ論理設計の特徴



- 講義2単位，演習1単位．
- 1人1台のFPGA (Field-Programmable Gate Array) ボードを用いた演習．
- 4人程度を1グループとした共同作業と問題解決．
- 教科書で説明されるプロセッサのRISC-V版を ハードウェア記述言語Verilog HDLで記述し，FPGAボードに実装する．
- グループとしてプロセッサの高速化に取り組み，コンテスト形式で成果を競う．
- 3Q開講のコンピュータアーキテクチャ（CSC.T363）のための準備．

```
module main (clk, led);
  input  wire clk;
  output wire led;

  reg [26:0] cnt=0;
  always @(posedge clk) cnt <= cnt + 1;

  assign led = cnt[26];
endmodule
```

Verilog HDL code



「2023年度」のコンピュータ論理設計の注意点

- 2023年度の講義と演習は、学術国際情報センター3階 情報工学系計算機室で実施します.
- また，Slack も活用します.
- アダプティブコンピューティング研究推進体 (ACRi) が提供する FPGA の利用環境である ACRi ルームを利用します.
  - https://gw.acri.c.titech.ac.jp/wp/manual/welcome
- 期末試験（ペーパーテスト）を実施します．演習 30%, 設計コンテスト 20%, 期末試験 50% として評価します.
  - 演習，設計コンテストには出席点も含まれるので，毎回，参加しましょう.
- 講義日程や資料などはサポートページを参照してください.
  - www.arch.cs.titech.ac.jp/lecture/CLD/

Syllabus (1/3)

講義の概要とねらい

本講義では、「論理回路理論」の講義で習得した知識をベースに、より実用的なディジタル回路について学ぶ。また、簡単なコンピュータを例題として、コンピュータの基本原理とその論理設計の方法を学習する。
演習では、学んだ組合せ回路と順序回路をVerilog HDL等のハードウェア記述言語で記述し、シミュレーションによる回路の動作検証、FPGAが搭載されたハードウェアボード等に実装して動作確認をおこなう。

到達目標

本講義を履修することによって以下を習得する。
・コンピュータシステムの基本構成
・シングルサイクルプロセッサの論理設計に関する知識
・パイプライン処理をおこなうプロセッサの論理設計に関する知識
・ハードウェア記述言語を用いたシンプルなコンピュータシステムの設計能力

キーワード

コンピュータ，命令セットアーキテクチャ，プロセッサ、パイプライン処理，ハードウェア記述言語，Verilog HDL、FPGA

学生が身につける力

国際的教養力	コミュニケーション力	専門力	課題設定力	実践力または解決力
-	-	✓	-	✓

授業の進め方

原則として、90分×2コマの講義の後、90分×1コマのFPGAボードを用いた演習をおこないます。

FPGAの展望

世界のFPGA市場、2025年に86億米ドル規模へ

2020年04月20日 13時30分 公開　　[馬本隆綱，EE Times Japan]

印刷する　クリップする　21　Share　B!　1

2019年には28nm未満の構成比率が最大に

グローバルインフォメーションは2020年4月、FPGAの世界市場調査レポートを発売した。同レポートによると、FPGA市場は2020年の59億米ドルに対し、2025年は86億米ドルと予測した。年平均成長率（CAGR）は7.6%となる。

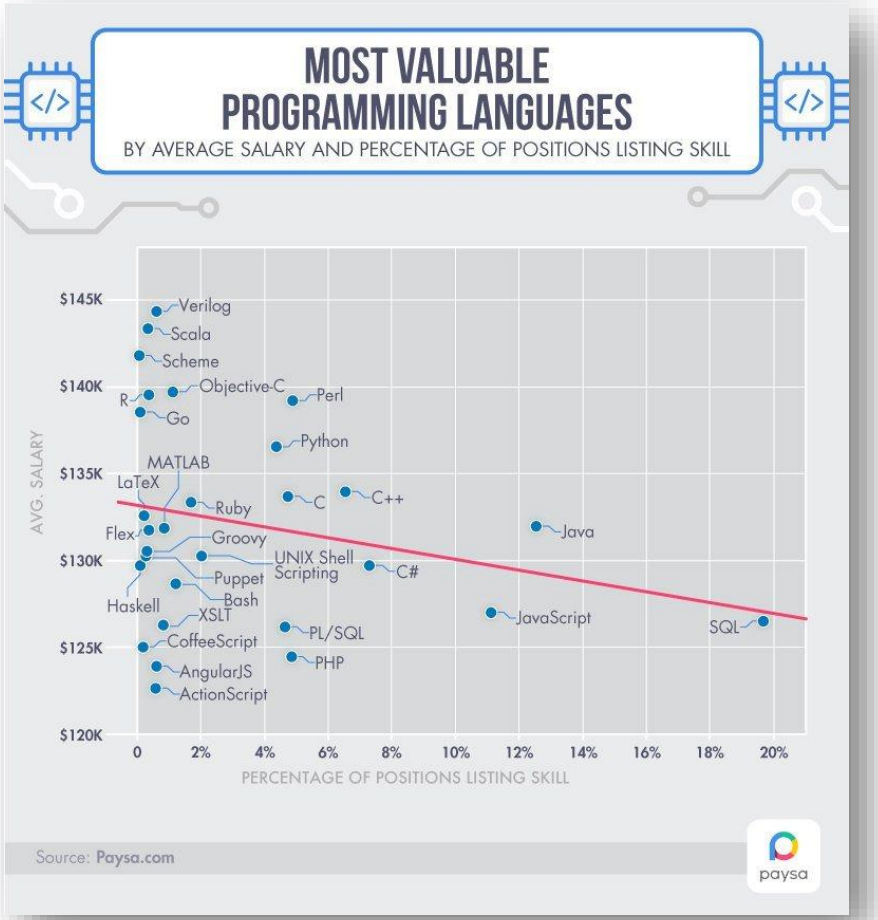
市場調査レポートのタイトルは「FPGAの世界市場：ローエンドFPGA・ミッドレンジFPGA・ハイエンドFPGA・SRAM型・FLASH型FPGA・アンチヒューズ型FPGA-2025年までの予測」（MarketsandMarkets発行）である。

プレスリリースでは詳細な数値を明らかにしていないが、FPGA市場をテクノロジーノードで分類すると、2019年は28nm未満の製品構成比率が最大になったという。低消費電力製品の登場などにより、2025年に向けても28nm未満の製品が高い伸びを示すと予測した。

2020年と2025年のFPGA市場 出典：グローバルインフォメーション

https://eetimes.jp/ee/articles/2004/20/news030.html

SILICON VALLEY'S MOST VALUABLE SKILLS



2016-09
https://www.paysa.com/blog/silicon-valleys-most-valuable-skills/

Syllabus (2/3)

授業計画・課題		
	授業計画	課題
第1回	コンピュータシステムの基本構成	コンピュータシステムの基本構成について理解する。
第2回	論理設計演習(1)	論理設計演習(1)
第3回	ハードウェア記述言語：組合せ回路	組合せ回路の記述を理解する。
第4回	ハードウェア記述言語：順序回路	順序回路の記述を理解する。
第5回	論理設計演習(2)	論理設計演習(2)
第6回	ハードウェア記述言語：よく使われる回路	よく使われる回路の記述を理解する。
第7回	リコンフィギャラブルシステム	リコンフィギャラブルシステムとFPGAボードについて理解する。
第8回	論理設計演習(3)	論理設計演習(3)
第9回	命令セットアーキテクチャ：データ表現とアドレス指定形式	ISAにおけるデータ表現とアドレス指定形式について理解する。
第10回	命令セットアーキテクチャ：算術論理演算命令	ISAにおける算術論理演算命令について理解する。
第11回	論理設計演習(4)	論理設計演習(4)
第12回	命令セットアーキテクチャ：ロードストア命令と分岐命令	ISAにおけるロードストア命令と分岐命令について理解する。
第13回	プロセッサの基本構成要素：算術論理演算ユニット	算術論理演算ユニットについて理解する。
第14回	論理設計演習(5)	論理設計演習(5)
第15回	プロセッサの基本構成要素：レジスタファイルとメモリ	レジスタファイルとメモリについて理解する。
第16回	シングルサイクルプロセッサのデータパス	シングルサイクルプロセッサのデータパスについて理解する。
第17回	論理設計演習(6)	論理設計演習(6)
第18回	シングルサイクルプロセッサの制御	シングルサイクルプロセッサの制御について理解する。
第19回	パイプライン処理	パイプライン処理について理解する。
第20回	論理設計演習(7)	論理設計演習(7)
第21回	パイプラインハザードとデータフォワーディング	パイプラインハザードとデータフォワーディングについて理解する。
第22回	論理設計演習(8)	論理設計演習(8)

Syllabus (3/3)

教科書

デイビッド・A. パターソン、ジョン・L. ヘネシー (著)、成田光彰 (翻訳)『コンピュータの構成と設計 第5版 上/下』日経BP社

参考書、講義資料等

無し。

成績評価の基準及び方法

講義で扱うコンピュータ論理設計に関する理解、ハードウェア記述言語を用いたコンピュータシステム実装への応用力を評価する。演習（30%）と期末

演習 30%, 設計コンテスト 20%, 期末試験 50%

関連する科目

CSC.T252 ： 論理回路理論
CSC.T262 ： アセンブリ言語
CSC.T372 ： コンパイラ構成
CSC.T363 ： コンピュータアーキテクチャ
CSC.T433 ： 先端コンピュータアーキテクチャ

履修の条件(知識・技能・履修済科目等)

履修条件は特に設けないが、関連する科目の論理回路理論を履修していることが望ましい。

連絡先（メール、電話番号）　※"[at]"を"@"(半角)に変換してください。

吉瀬謙二: kise[at]c.titech.ac.jp

オフィスアワー

メールで事前予約すること。

講義のサポートページ
https://www.arch.cs.titech.ac.jp/lecture/CLD/

Lecture Slides and Materials

2023年度の講義と演習は**「学術国際情報センター３階 情報工学系計算機室」**で実施します。

- 2023-04-10 (10:45): Lecture01: Basic Structure of Computer Systems
- 2023-04-13 (08:50): Exercise (1):
- 2023-04-13 (10:45): Lecture02: Hardware Description Language: Combinational Circuit
- 2023-04-17 (10:45): Lecture03: Hardware Description Language: Sequential Circuit
- 2023-04-20 (08:50): Exercise (2):
- 2023-04-20 (10:45): Lecture04: Hardware Description Language: Typical Circuits
- 2023-04-24 (10:45): Lecture05: VLSI and Reconfigurable Systems
- 2023-04-27 (08:50): Exercise (3):
- 2023-04-27 (10:45): Lecture06: Instruction Set Architecture: Data Representation and Addressing
- 2023-05-01 (10:45): Lecture07: Instruction Set Architecture: Arithmetic and Logic Instructions
- 2023-05-08 (08:50): Exercise (4):
- 2023-05-08 (10:45): Lecture08: Instruction Set Architecture: Load/Store and Branch Instructions
- 2023-05-11 (08:50): Exercise (5):
- 2023-05-11 (10:45): Lecture09: Design and Implementation of a Single Cycle Processor (1)
- 2023-05-15 (10:45): Lecture10: Design and Implementation of a Single Cycle Processor (2)
- 2023-05-18 (08:50): Exercise (6):
- 2023-05-18 (10:45): Lecture11: Design and Implementation of a Multi-cycle Processor
- 2023-05-12 (10:45): Lecture12: Pipelining and Hazards (1)
- 2023-05-25 (08:50): Exercise (7):
- 2023-05-25 (10:45): Lecture13: Pipelining and Hazards (2)
- 2023-05-29 (10:45): Lecture14: Preparing for the design contest (group work)
- 2023-06-01 (08:50-12:25): Design Contest
- 2023-06-05 (10:45): **Final Examination**

参考書

- 4ビットカウンタとシリアル通信ではじめるFPGA開発 ACRiブログシリーズ Kindle版
- ハードウェア記述言語を用いたFPGA開発の入門書です。
  FPGAボードを購入することなく、無料で体験できるACRiルームの利用方法を説明しているので、かんたんにFPGAのためのハードウェア設計を始めることができます！

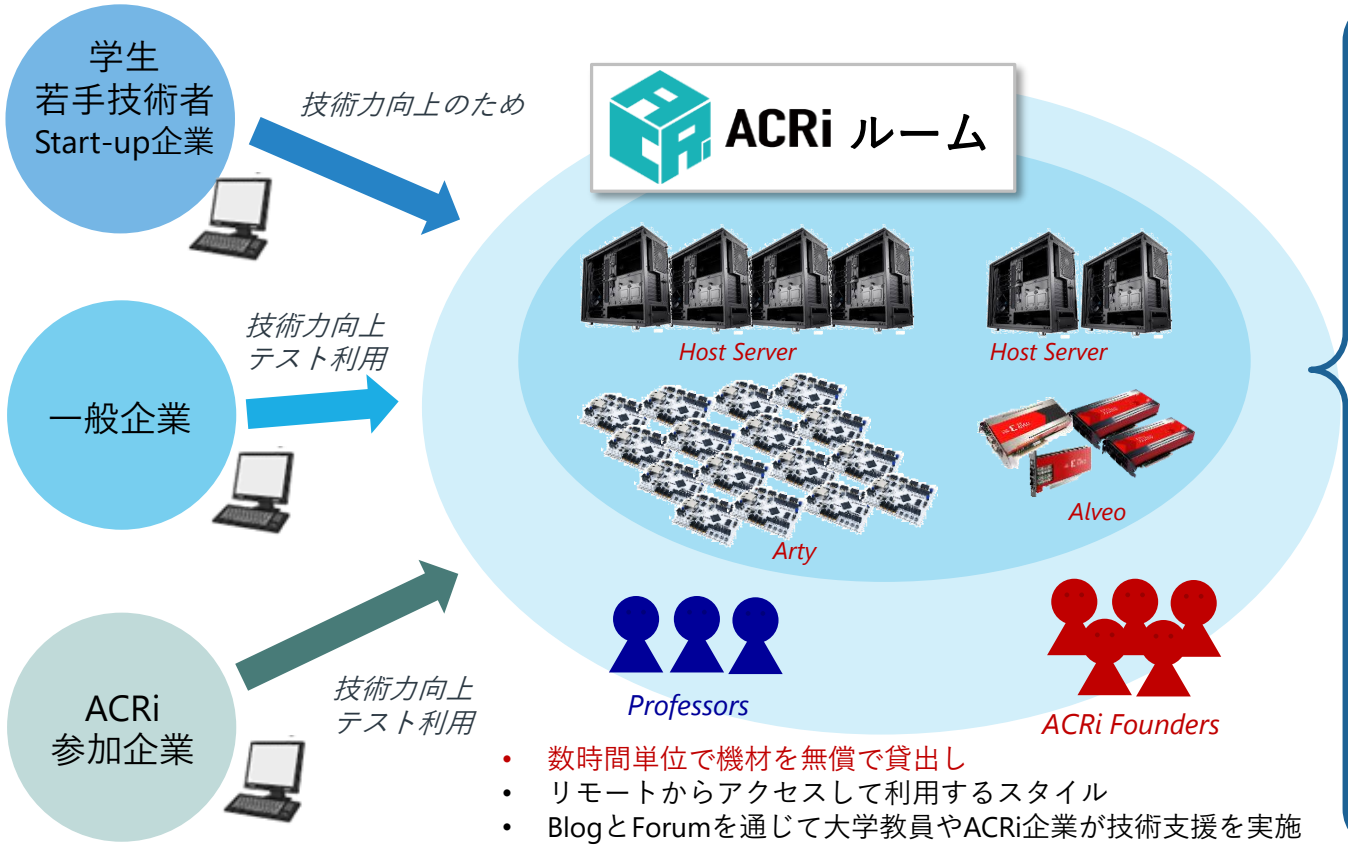
教科書

- コンピュータの構成と設計 第5版、パターソン＆ヘネシー（成田光彰　訳）、日経BP社
  - 1. コンピュータの抽象化とテクノロジ
  - 2. 命令:コンピュータの言葉
  - 3. コンピュータにおける算術演算
  - 4. プロセッサ
  - A. アセンブラ，リンカ，SPIMシミュレータ
  - B. 論理設計の基礎

ACRiルーム (FPGA利用環境)

www.acri.c.titech.ac.jp

－ 日本初、産学連携でFPGA検証環境と学習機会を無償で提供 －



- 数時間単位で機材を無償で貸出し
- リモートからアクセスして利用するスタイル
- BlogとForumを通じて大学教員やACRi企業が技術支援を実施

FPGA Server

- CPU: Core i9 (8 core /16 thread)
- メモリ: DDR4 128GB (32GB x 4)
- ストレージ: SSD M.2 1TB x 2

Arty A7-35T

- 1サーバにArtyを15枚設置
- Digilent Arty A7-35T
- ユーザ毎にVMを割り当て
- 3時間でVMを再起動

Alveo

- 1サーバにAlveoを1枚設置
- Xilinx Alveo
  U50 / U200 / U250 / U280

アダプティブコンピューティング研究推進体
Adaptive Computing Research Initiative (ACRi)

ACRiルームのユーザー登録

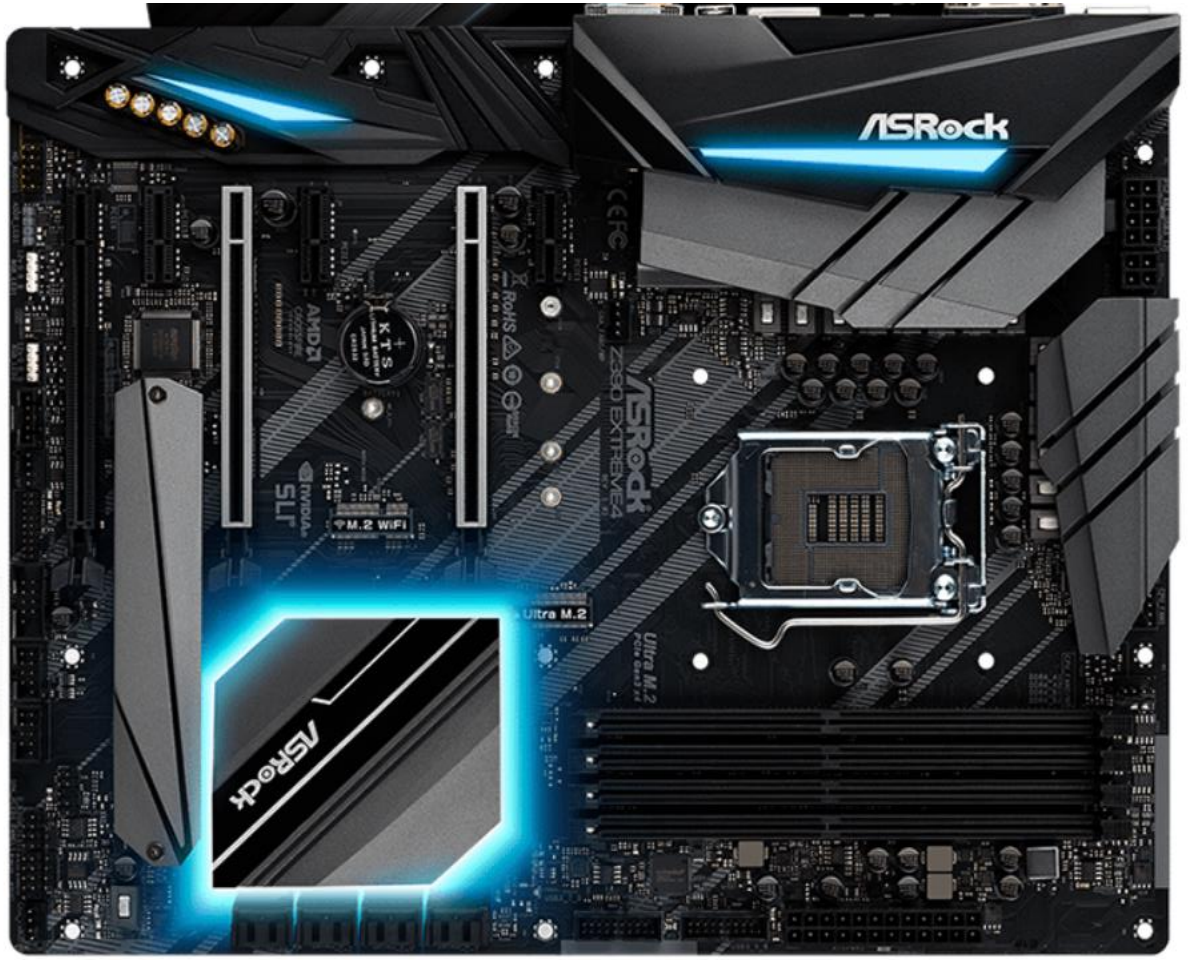
- 論理回路理論でACRiルームのアカウントを持っていれば，必要ありません．
- そうでなければ，今日中に，次のサイトで新規ユーザーの登録をしてください．
  - https://gw.acri.c.titech.ac.jp/wp/
- 登録時のメールアドレスには m.titech.ac.jp を用いること．

ACRiルームのサーバーコンピュータとFPGAボード

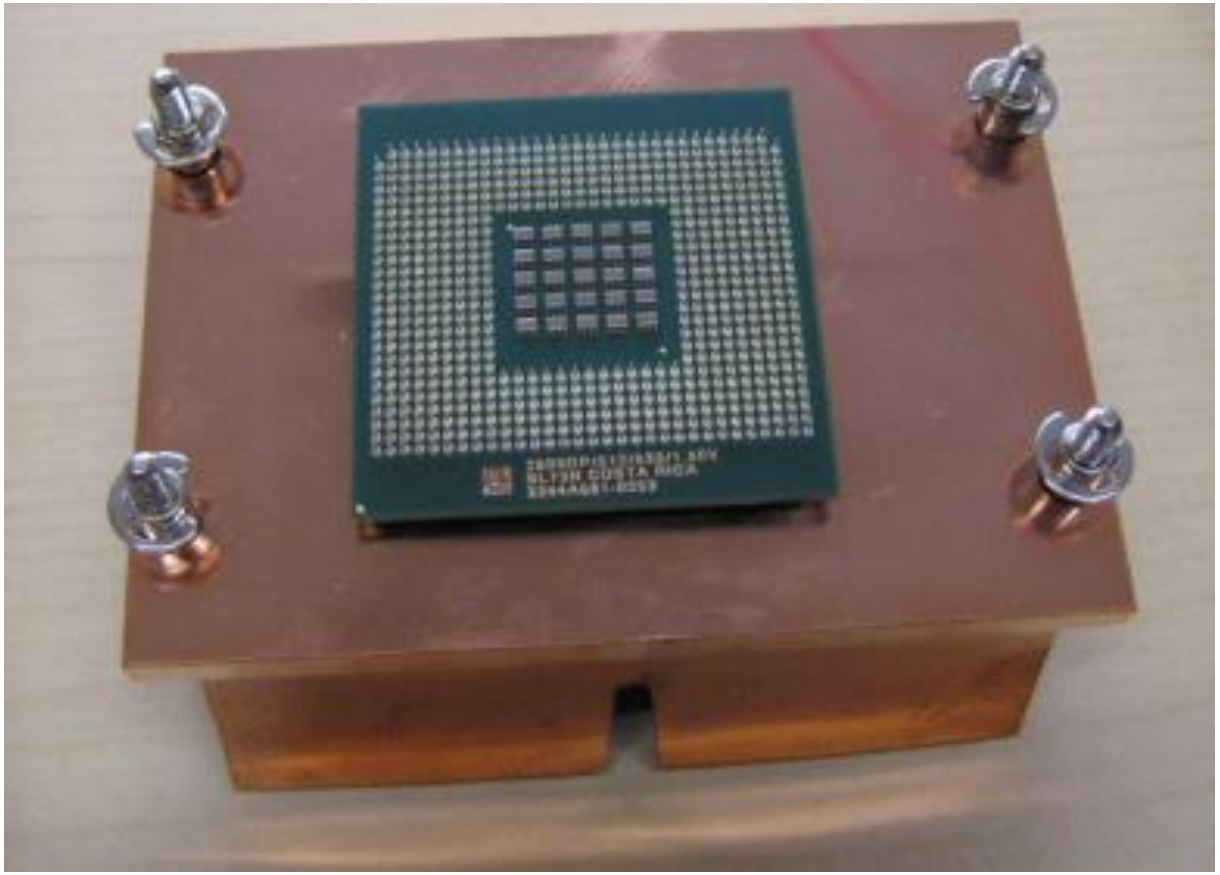
Arty A7-35T

- 1サーバにArtyを15枚設置
- Digilent Arty A7-35T
- ユーザ毎にVMを割り当て
- 3時間でVMを再起動

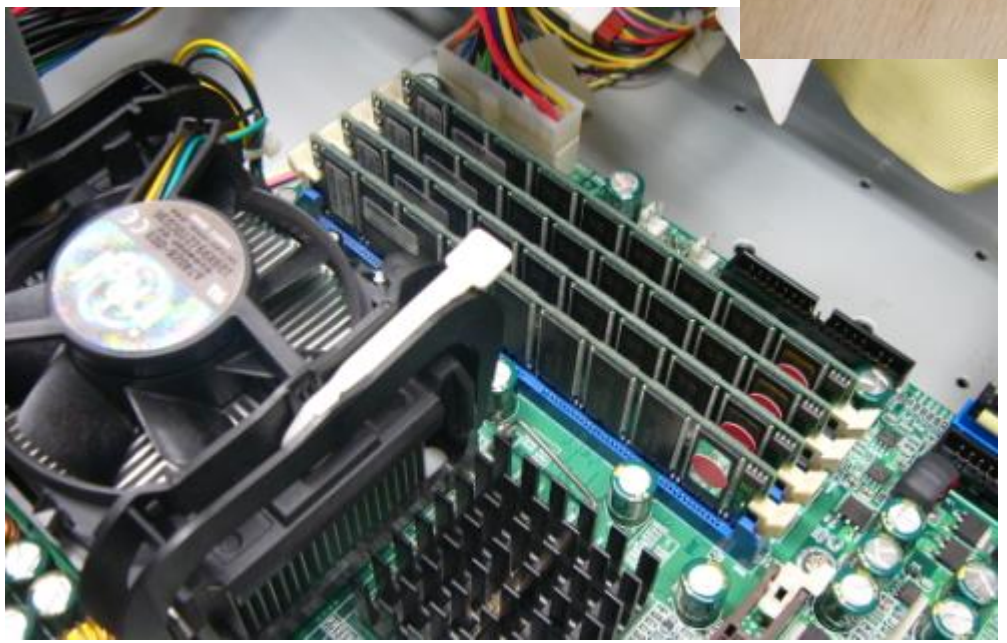
マザーボード



マイクロプロセッサ，CPU (Central Processing Unit)



メモリ，記憶
DRAM (dynamic random access memory)



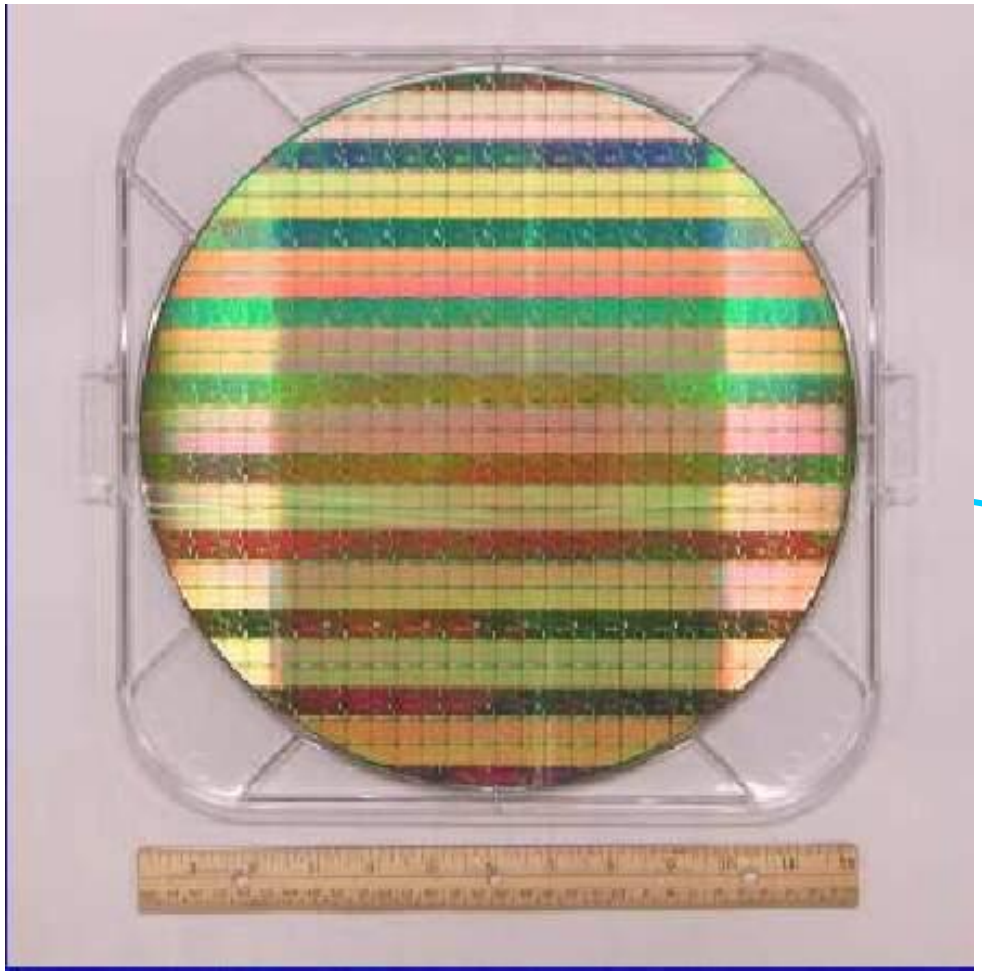
ディスク（SSD, 磁気ディスク）

グラフィックカード, GPU

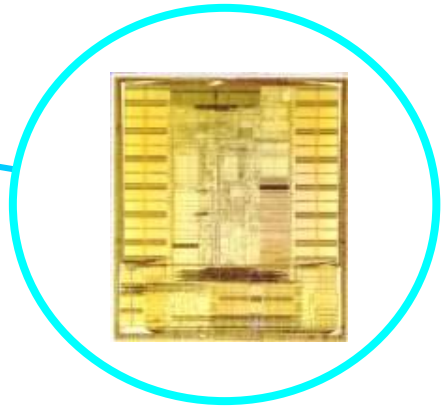
ネットワークカード

VLSIチップの製造：インゴット，ウエーハ

VLSIチップの製造：ウエーハとダイ



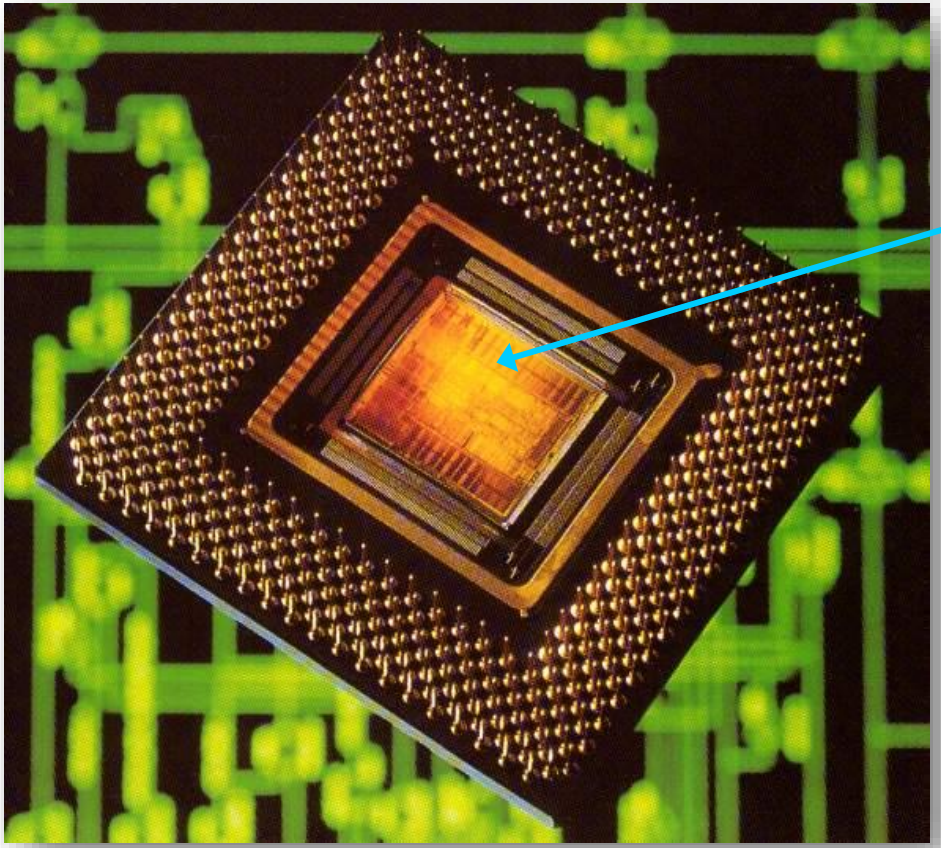
30cmのウエーハ(wafer)
厚さは数ミリで、直径が30cm
大きなCDのような形をしている。



ダイ (die)
（ウエーハから切り出した
個々のチップ）

Intel社, Industry-Leading Transistor Performance Demonstrated on Intel's 90-nanometer Logic Process

プロセッサの実装：ダイのパッケージ化

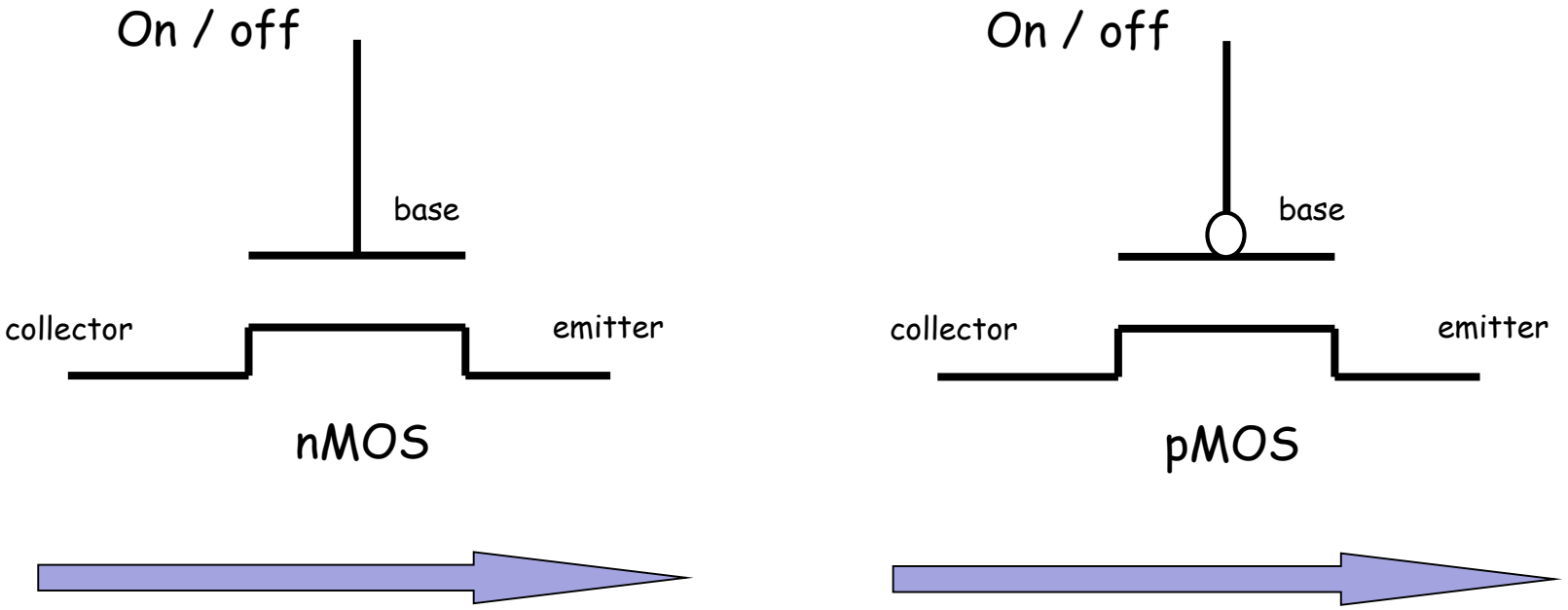


die (dice)：さいころ

Richard L. Sites, Alpha AXP Architecture Reference Manual SECOND EDITION

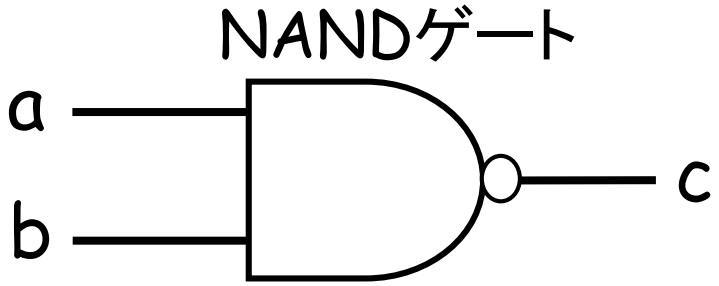
Transistor

- トランジスタは電気的なオン／オフ動作をするスイッチと捉えることができる.

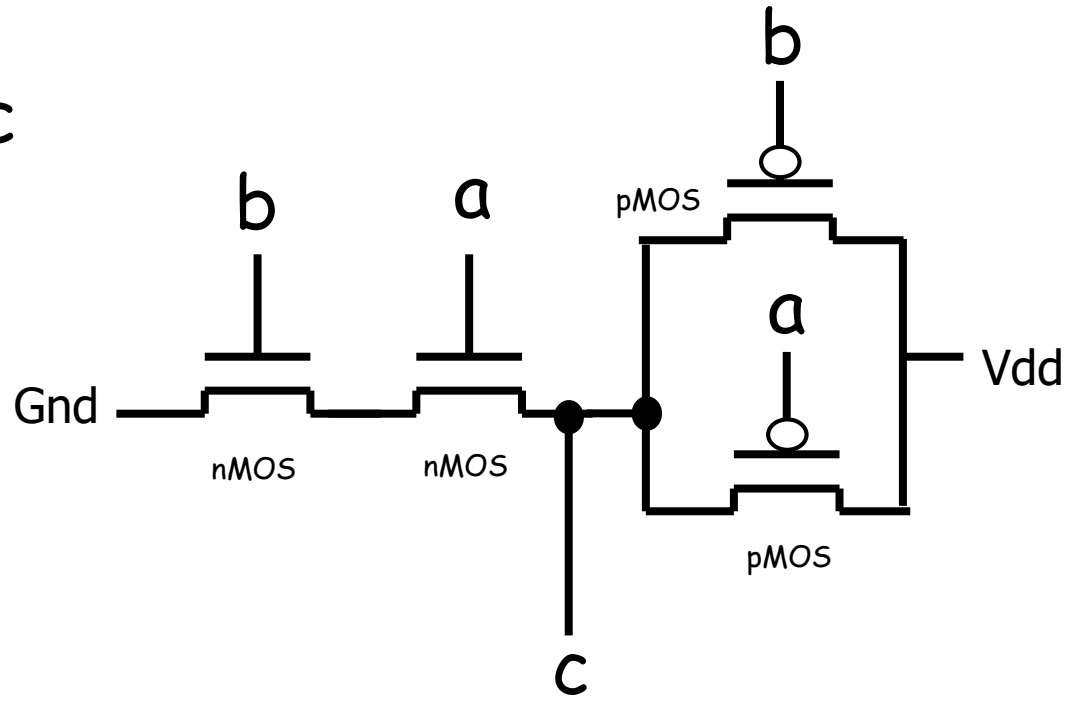


Transistor and Gate

- トランジスタは電気的なオン／オフ動作をするスイッチ
- 幾つかのトランジスタで，少し機能の高いゲートを構成

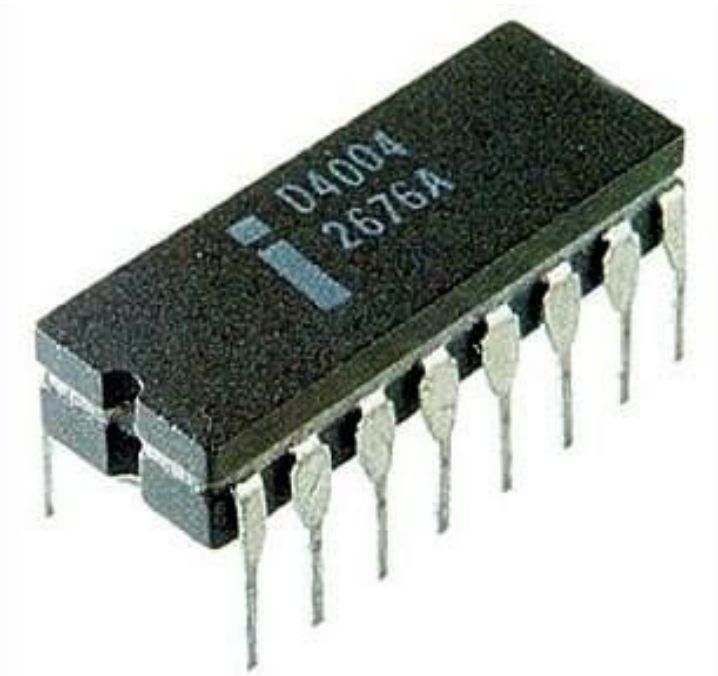


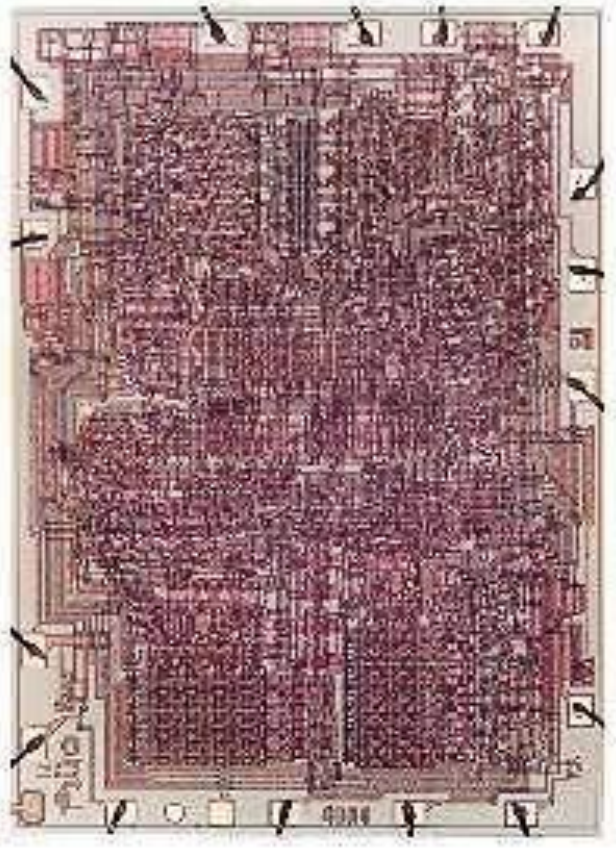
a	b	c
0	0	1
1	0	1
0	1	1
1	1	0



The first commercially available microprocessor

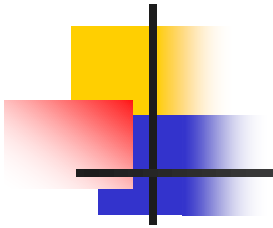
1971年: 4004 マイクロプロセッサ





プロセッサ	出荷年	トランジスタ数
4004	1971	2,250

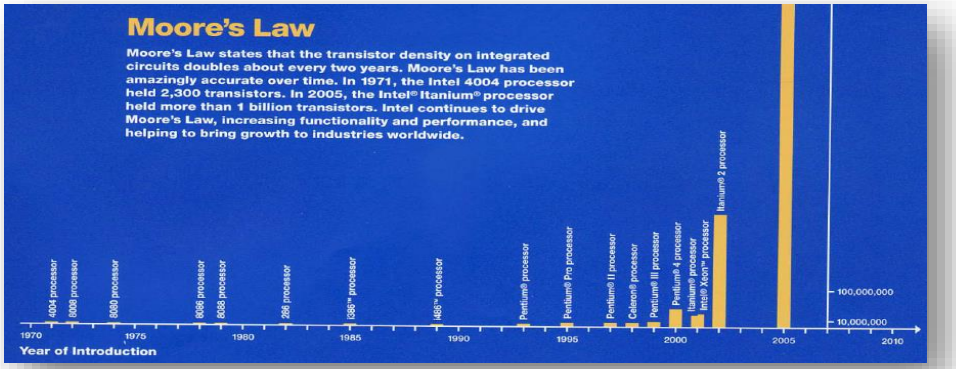
From Wikipedia, Intelミュージアム



Moore’s law

- Moore's law is the observation that the number of transistors in a dense integrated circuit doubles approximately every two years.

Moore's Law



VISUALIZING PROGRESS

If transistors were people

If the transistors in a microprocessor were represented by people, the following timeline gives an idea of the pace of Moore's Law.

2,300
Average music hall capacity

134,000
Large stadium capacity

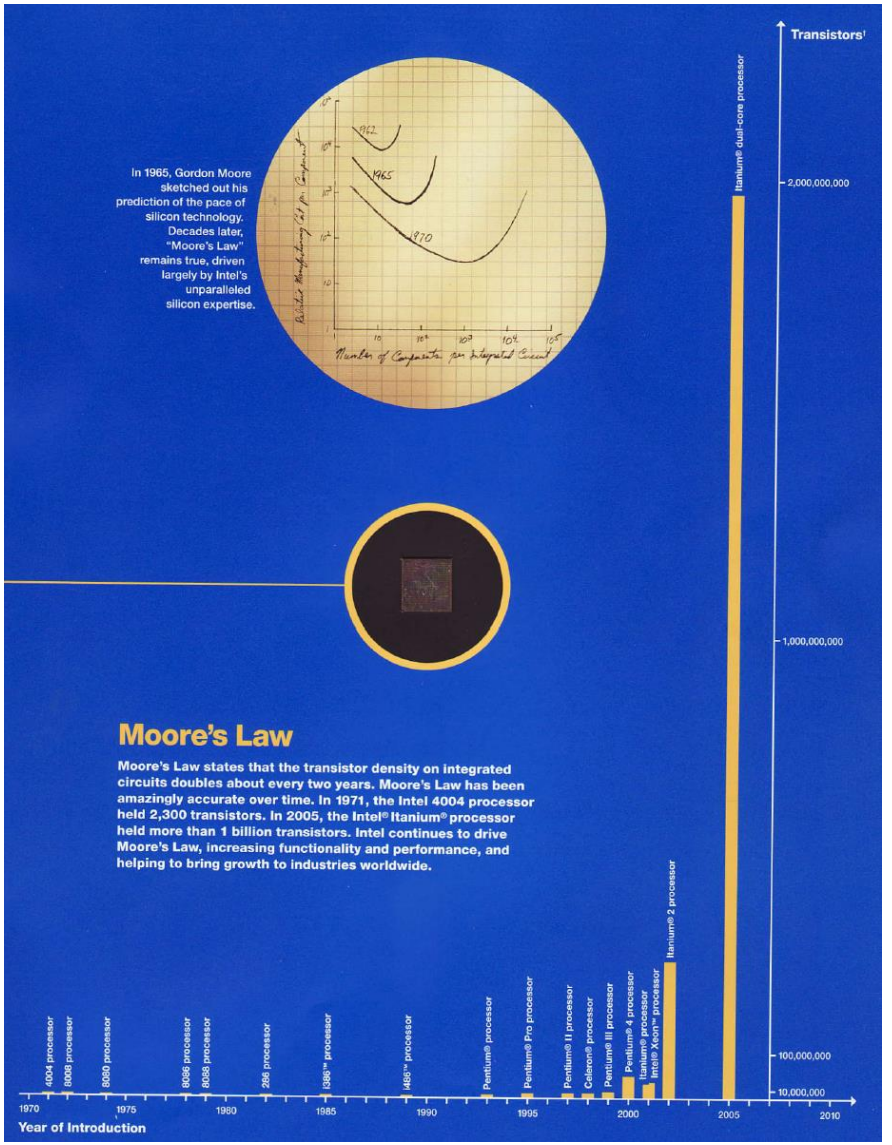
32 Million
Population of Tokyo

1.3 Billion
Population of China

1970	1980	1990	2000	2011
Intel 4004	Intel 286		Pentium III	Core i7 Extreme Edition

Now imagine that those 1.3 billion people could fit onstage in the original music hall. That's the scale of Moore's Law.

Moore's Law


In 1965, Gordon Moore sketched out his prediction of the pace of silicon technology. Decades later, "Moore's Law" remains true, driven largely by Intel's unparalleled silicon expertise.
Moore's Law
Moore's Law states that the transistor density on integrated circuits doubles about every two years. Moore's Law has been amazingly accurate over time. In 1971, the Intel 4004 processor held 2,300 transistors. In 2005, the Intel® Itanium® processor held more than 1 billion transistors. Intel continues to drive Moore's Law, increasing functionality and performance, and helping to bring growth to industries worldwide.
Transistors†
2,000,000,000
1,000,000,000
100,000,000
10,000,000
4004 processor
8008 processor
8080 processor
8086 processor
8088 processor
286 processor
386™ processor
486™ processor
Pentium® processor
Pentium® Pro processor
Pentium® II processor
Celeron® processor
Pentium® III processor
Pentium® 4 processor
Itanium® processor
Intel® Xeon™ processor
Itanium® 2 processor
Itanium® dual-core processor
1970
1975
1980
1985
1990
1995
2000
2005
2010
Year of Introduction

Moore's Law

The experts look ahead

Cramming more components onto integrated circuits

With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65,000 components on a single silicon chip

By Gordon E. Moore

Director, Research and Development Laboratories, Fairchild Semiconductor division of Fairchild Camera and Instrument Corp.

The future of integrated electronics is the future of electronics itself. The advantages of integration will bring about a proliferation of electronics, pushing this science into many new areas.

Integrated circuits will lead to such wonders as home computers—or at least terminals connected to a central computer—automatic controls for automobiles, and personal portable communications equipment. The electronic wristwatch needs only a display to be feasible today.

But the biggest potential lies in the production of large systems. In telephone communications, integrated circuits in digital filters will separate channels on multiplex equipment. Integrated circuits will also switch telephone circuits and perform data processing.

Computers will be more powerful, and will be organized in completely different ways. For example, memories built of integrated electronics may be distributed throughout the machine instead of being concentrated in a central unit. In addition, the improved reliability made possible by integrated circuits will allow the construction of larger processing units. Machines similar to those in existence today will be built at lower costs and with faster turn-around.

Present and future

By integrated electronics, I mean all the various technologies which are referred to as microelectronics today as well as any additional ones that result in electronics functions supplied to the user as irreducible units. These technologies were first investigated in the late 1950's. The object was to miniaturize electronics equipment to include increasingly complex electronic functions in limited space with minimum weight. Several approaches evolved, including microassembly techniques for individual components, thin-film structures and semiconductor integrated circuits.

Each approach evolved rapidly and converged so that each borrowed techniques from another. Many researchers believe the way of the future to be a combination of the various approaches.

The advocates of semiconductor integrated circuitry are already using the improved characteristics of thin-film resistors by applying such films directly to an active semiconductor substrate. Those advocating a technology based upon films are developing sophisticated techniques for the attachment of active semiconductor devices to the passive film arrays.

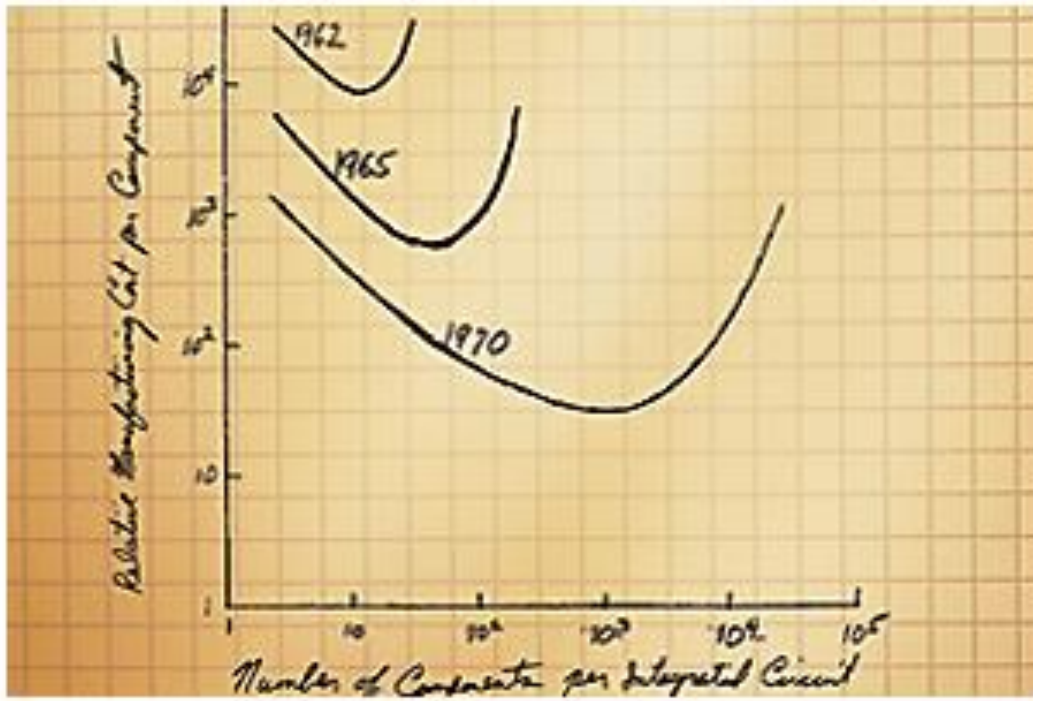
Both approaches have worked well and are being used in equipment today.

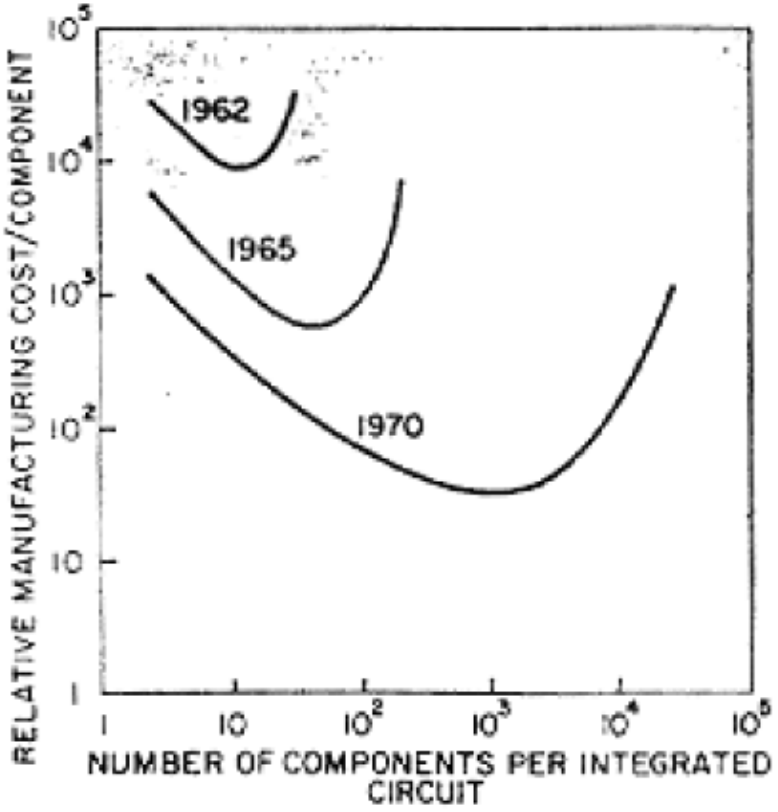
The author

Dr. Gordon E. Moore is one of the new breed of electronic engineers, schooled in the physical sciences rather than in electronics. He earned a B.S. degree in chemistry from the University of California and a Ph.D. degree in physical chemistry from the California Institute of Technology. He was one of the founders of Fairchild Semiconductor and has been director of the research and development laboratories since 1959.

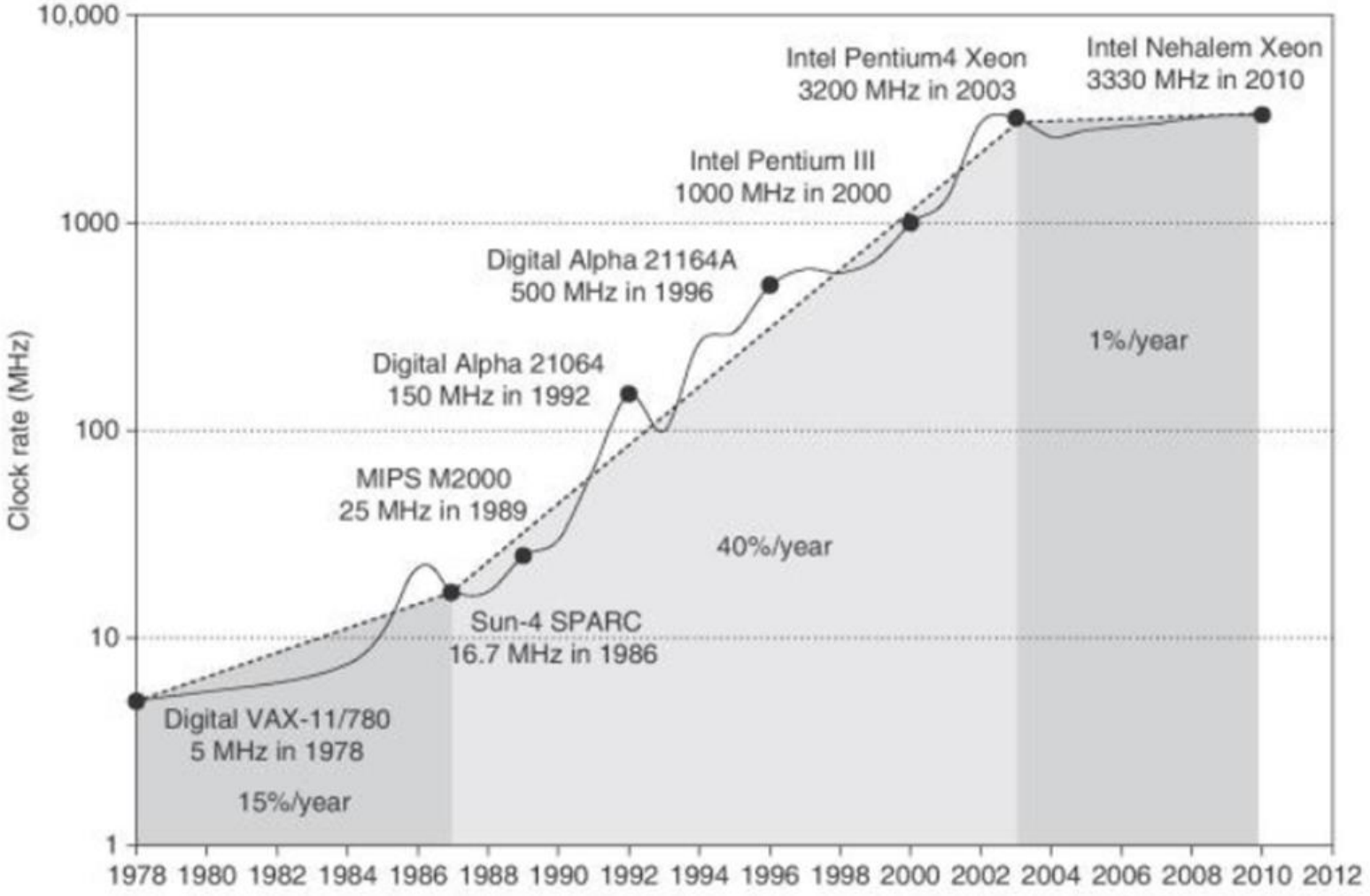
Electronics, Volume 38, Number 8, April 19, 1965

Moore's Law


1962
1965
1970


RELATIVE MANUFACTURING COST/COMPONENT
1962
1965
1970
NUMBER OF COMPONENTS PER INTEGRATED CIRCUIT

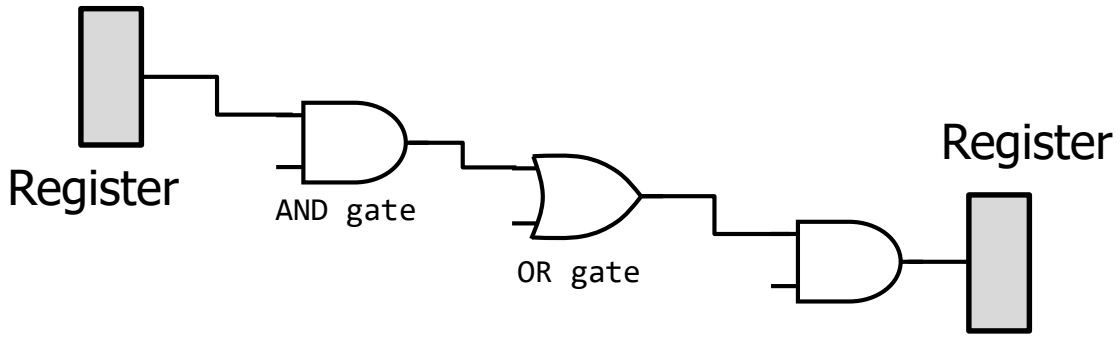
Growth in clock rate of microprocessors



From CAQA 5th edition

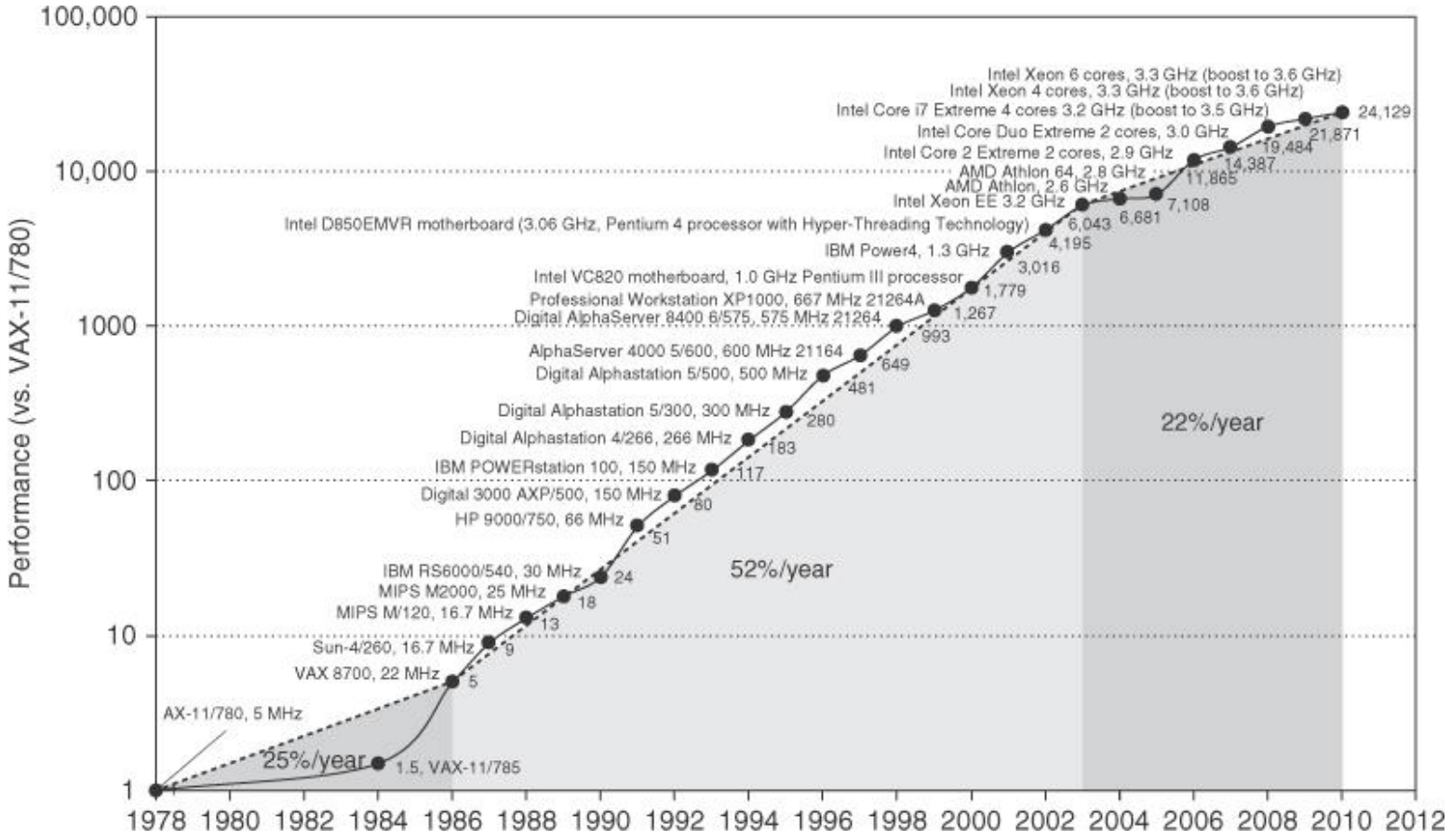
Clock rate is mainly determined by

- Switching speed of gates (transistors)
- The number of levels of gates
  - The maximum number of gates cascaded in series in any combinational logics.
  - In this example, the number of levels of gates is 3.
- Wiring delay and fanout



Growth in processor performance

- Performance = f x IPC
  - f: frequency (clock rate)
  - IPC: retired machine Instructions Per Cycle



From CAQA 5th edition

Code057.v

- w_clk は 100MHz のクロック信号
- 32ビットのレジスタ r_cnt を，毎サイクル、インクリメントする．ただし，値が99,999,999 の時には0に初期化される．（つまり，1秒毎に初期化される）
- 100,000,000 サイクル毎（1秒）に，1ビットのレジスタ r_out の値を反転する．

```
/*********************************************************************/
/* code057.v                          For CSC.T341 CLD Archlab TOKYO TECH */
/*********************************************************************/
module m_main (w_clk, w_led);
  input  wire w_clk;
  output wire [3:0]  w_led;

  reg        r_out = 0;
  reg [31:0] r_cnt = 0;
  always@(posedge w_clk) begin
    r_cnt <= (r_cnt==99999999) ? 0 : r_cnt +1;
    r_out <= (r_cnt==0) ? ~r_out : r_out;
  end
  assign w_led = {r_out, r_out, r_out, r_out};
  // vio_0 vio_00(w_clk, w_led[3], w_led[2], w_led[1], w_led[0]);
endmodule
```

```
module m_main (w_clk, w_led);
  input  wire w_clk;
  output wire [3:0]  w_led;

  reg        r_out = 0;
  reg [31:0] r_cnt = 0;
  always@(posedge w_clk) begin
    r_cnt <= (r_cnt==99999999) ? 0 : r_cnt +1;
    r_out <= (r_cnt==0) ? ~r_out : r_out;
  end
  assign w_led[0] = r_out;
  assign w_led[1] = r_out;
  assign w_led[2] = r_out;
  assign w_led[3] = r_out;
  // vio_0 vio_00(w_clk, w_led[3], w_led[2], w_led[1], w_led[0]);
endmodule
```

ビット連結 { } を用いた左のコードと，ビット毎に接続 assign する右のコードは等価．

補足：Code057.v

- w_clk は **100MHz** のクロック信号
- 32ビットのレジスタ r_cnt を，毎サイクル、インクリメントする．ただし，値が99,999,999の時には0に初期化される．（つまり，1秒毎に初期化される）
- **100,000,000 サイクル毎（1秒）に**，1ビットのレジスタ r_out の値を反転する．

- 1KHz のクロック信号は 1,000 Hz と同じ．
- 1KByte のメモリは 1024 Byte と同じ．

- 1MHz のクロック信号は 1000 x 1000 = 1,000,000 Hz と同じ．
- 1MByte のメモリは 1024 x 1024 = 1,048,576 Byte と同じ．

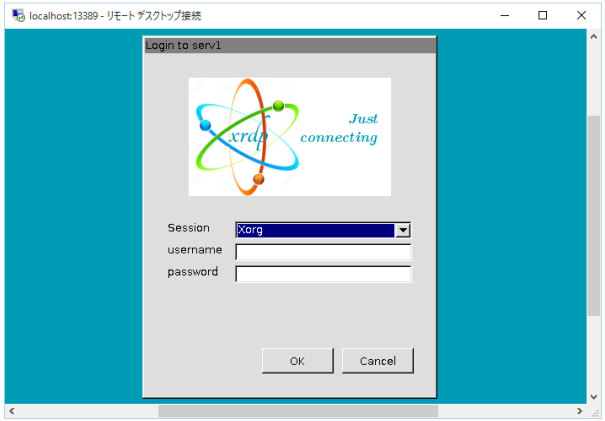
- 100MHz のクロック信号は 100 x 1000 x 1000 = 100,000,000 Hz と同じ．

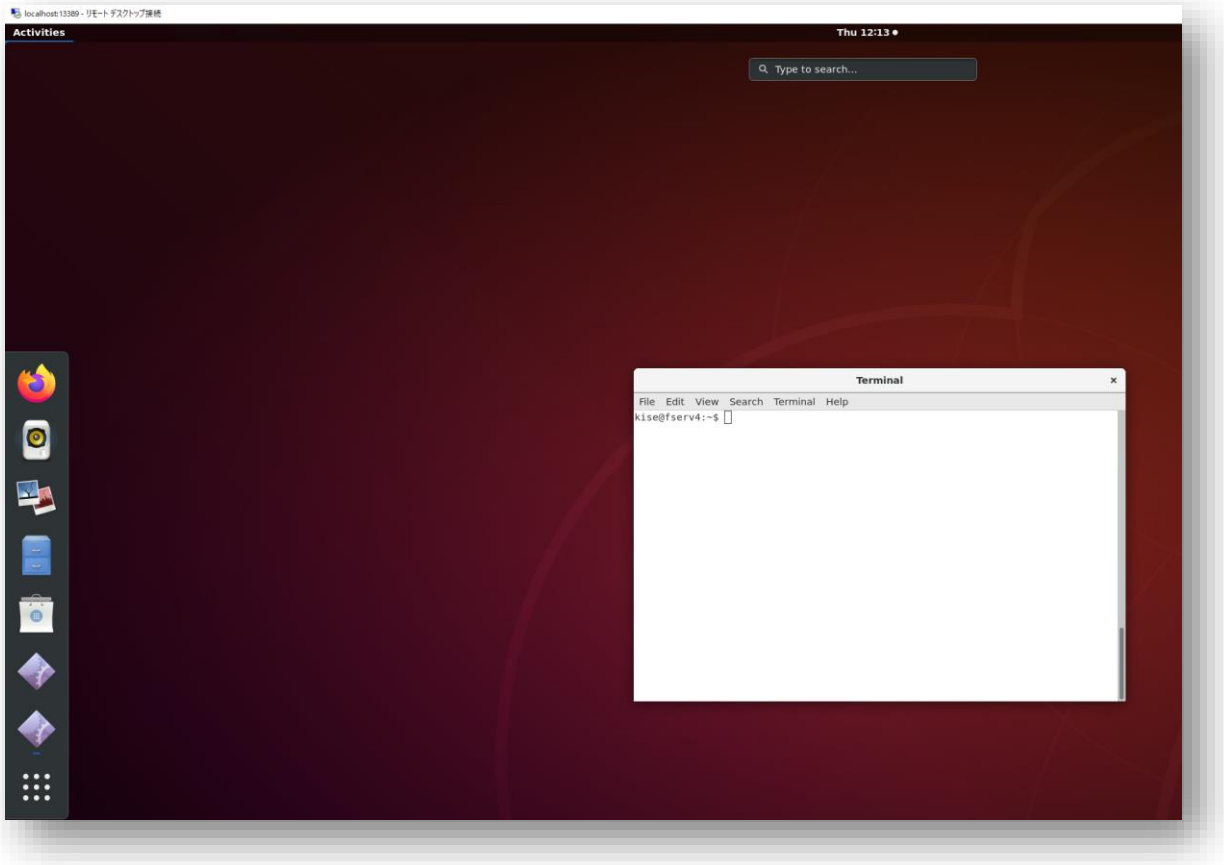
ACRiルームのサーバー計算機

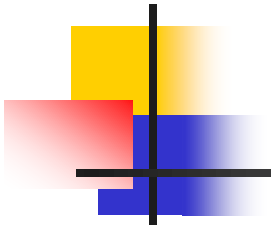
各マシンの負荷を下げるために，仮想マシンの名前の最後の2文字が12～15は使わない．
具体的には vs001～vs011, vs101～vs111, vs201～vs211, vs301～vs311, vs401～vs411, vs501～vs511, vs601～vs610 から選ぶこと．

ACRiルームのデモンストレーション

- Vivado で FPGA をコンフィギュレーションして動作を確認する.
- code057.v と main11.xdc を用いて，演習(1) の最初の部分をやってみる.







コンピュータ論理設計 演習

- 最初の演習は，４月１３日 8:50開始です．
  8:45までに，学術国際情報センター3階 情報工学系計算機室 に集まってください．
- また Slack で情報を提供します．Slack も活用しましょう．

Department of Computer Science
Course number: CSC.T341

2023年度の講義と演習は，学術国際情報センター3階 情報工学系計算機室で実施します．

コンピュータ論理設計
Computer Logic Design

2. ハードウェア記述言語：組合せ回路 (1)
Hardware Description Language: Combinational Circuit (1)

吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
講義：月曜日 10:45-12:25, 木曜日 10:45-12:25

Verilog HDL

- ハードウェア記述言語 (Hardware Description Language)
  - IEEE 1364 として標準化
- C言語に近い（C言語を参考にしている）

Sample Verilog HDL code

- ACRi Room のサーバーにリモートデスクトップでログインする.
  - /home/tu_kise/cld/lec2/ にサンプルのコードがあるので，Ubuntu のターミナルで次のコマンドを入力して，自分のディレクトリにコピーする.
  - /home/tu_kise は automount のディレクトリなので，アクセスしないとファイルが見えない．tabキーによる補完がうまく動作しないことがあるので注意する.

```
$ cd
$ mkdir cld
$ cd cld
$ cp /home/tu_kise/cld/lec2/* .
```

- code001.v をシミュレーションするためには.
  - リモートデスクトップで接続した Ubuntu のターミナルで次のコマンドを入力する.
  - コマンド iverilog でコンパイルして，生成される a.out を実行する.

```
$ iverilog code001.v
$ ./a.out
```

code001.v モジュールの定義と文字列の表示

```
$ iverilog code001.v
$ ./a.out
```

- code001.v をシミュレーションして，その表示を確認すること．
- モジュールの定義はキーワードmoduleからキーワードendmoduleまで．
- moduleの後にモジュール名を書く．この例ではmainがモジュール名．
- モジュール名の後の括弧内に入出力の端子名を列挙する．ここでは端子は何も定義していない．
- セミコロン(;)で，モジュール名と端子の列挙を終える．
- キーワードinitialにより，シミュレーション開始時（時刻0）から処理を始めることを指定する．
- $display または $write はシステムタスクの1つで，メッセージを出力する． $write では改行されない．書式はC言語のprintfと同様．

Verilog HDL code (code001.v)

```
module main ();
  initial $display("hello, world");
endmodule
```

Simulation output

```
hello, world
```

Verilog HDLのコードは青色で，シミュレーションの出力は黄色で示す．

スライドPDFからコピーすると正しく動作しないことがあるので，コードは /home/tu_kise/ からコピーしたものを使うこと．

code002.v ブロックの指定とコンパイルエラーの対処

- code002.v をシミュレーションして，その表示を確認すること．
- 2つのシステムタスク$displayを用いた出力の例．2つのシステムタスクをブロックとしてまとめている．
- ブロックはキーワードbeginで始まり，キーワードendで終わる．C言語の { } に対応．
- code002_ng1.vは2番目の$displayがinitialブロックに含まれないので文法エラーとなる．

code002.v

```
module main ();
  initial begin
    $display("hello, world");
    $display("in Verilog HDL");
  end
endmodule
```

```
hello, world
in Verilog HDL
```

code002_ng.v

```
module main ();
  initial $display("hello, world");
  $display("in Verilog HDL");
endmodule
```

スライドPDFからコピーすると正しく動作しないことがあるので，コードは /home/tu_kise/ からコピーしたものを使うこと．

code003.v 複数のinitialの利用

- code003.v をシミュレーションして，その表示を確認すること．
- モジュール内で複数の initial を用いても良い．
  code002.v と code003.v の出力は同じ．

code002.v

```
module main ();
  initial begin
    $display("hello, world");
    $display("in Verilog HDL");
  end
endmodule
```

```
hello, world
in Verilog HDL
```

code003.v

```
module main ();
  initial $display("hello, world");
  initial $display("in Verilog HDL");
endmodule
```

```
hello, world
in Verilog HDL
```

スライドPDFからコピーすると正しく動作しないことがあるので，コードは /home/tu_kise/ からコピーしたものを使うこと．

code005.v 指定した時間が経過するまで待たせる命令#

- code005.v をシミュレーションして，その表示を確認すること．
- 指定した時間が経過するまで待たせる命令# を用いた例．
- #200 により，ここではシミュレーション開始時（時刻0）から200の時間が経過した時刻200に hello, world を表示する．
- #100 により，ここではシミュレーション開始時（時刻0）から100の時間が経過した時刻100に in Verilog HDL を表示する．
- 1行目はコメント，Verilog HDLのコメントはC, C++と同様．
- 時間の単位は nsec とする．#300 は 300nsec の時間経過を表す．

code005.v

```
/* sample Verilog code */
module main ();
  initial #200 $display("hello, world");
  initial #100 $display("in Verilog HDL");
endmodule
```

```
in Verilog HDL
hello, world
```

スライドPDFからコピーすると正しく動作しないことがあるので，コードは /home/tu_kise/ からコピーしたものを使うこと．

code006.v 複数のinitialを用いたコードの時間制御の例

- code006.v をシミュレーションして，その表示を確認すること．
- $displayによる出力の順番はどうなるか？

code006.v

```
module main ();
  initial #200 $display("hello, world");
  initial begin
    #100 $display("in Verilog HDL");
    #150 $display("When am I displayed?");
  end
endmodule
```

スライドPDFからコピーすると正しく動作しないことがあるので，コードは /home/tu_kise/ からコピーしたものを使うこと．

code007.v シミュレーションはいつ終わるのか

- code007.v をシミュレーションして，その表示を確認すること．
- 出力はどうなるか？

- Vivadoを用いてシミュレーションする場合，デフォルトの設定では1000nsしかシミュレーションしないので Verilog is easy? は出力されない．

code007.v

```
module main ();
  initial #200 $display("hello, world");
  initial begin
    #100 $display("in Verilog HDL");
    #150 $display("When am I displayed?");
    #1000 $display("Verilog is easy?");
  end
endmodule
```

スライドPDFからコピーすると正しく動作しないことがあるので，コードは /home/tu_kise/ からコピーしたものを使うこと．

code008.v システムタスク$time

- code008.v をシミュレーションして，その表示を確認すること．
- システムタスク$timeは，64ビットのシミュレーション時刻を返す．
- このコードでは，それぞれの $display が表示する時刻を表示する．
- 複雑な回路のシミュレーションでは，どの出力がどの時刻に出力されたのかわかりにくい場合がある．その場合，この例のように時刻を出力すると良い．

code008.v

```
module main ();
  initial #200 $display("%3d hello, world", $time);
  initial begin
    #100 $display("%3d in Verilog HDL", $time);
    #150 $display("%3d When am I displayed?", $time);
  end
endmodule
```

```
100 in Verilog HDL
200 hello, world
250 When am I displayed?
```

スライドPDFからコピーすると正しく動作しないことがあるので，コードは /home/tu_kise/ からコピーしたものを使うこと．

code009.v システムタスク$finish

- code009.v をシミュレーションして，その表示を確認すること．
- システムタスク$finishは，シミュレーションを終了させる．
- このコードでは時刻210でシミュレーションが終了する．
- Vivadoのデフォルトの設定では1000nsシミュレーションするが，それより短い時間のシミュレーションや，ある条件でシミュレーションを終了させたい場合に用いると良い．

code009.v

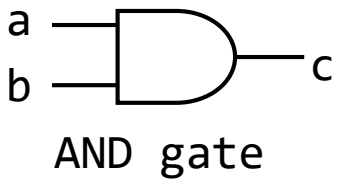
```
module main ();
  initial #200 $display("%3d hello, world", $time);
  initial begin
    #100 $display("%3d in Verilog HDL", $time);
    #150 $display("%3d When am I displayed?", $time);
  end
  initial #210 $finish;
endmodule
```

```
100 in Verilog HDL
200 hello, world
```

スライドPDFからコピーすると正しく動作しないことがあるので，コードは /home/tu_kise/ からコピーしたものを使うこと．

code011.v ANDゲートと重要な記述の幾つか

- code011.v をシミュレーションして，その表示を確認すること．
- reg型の信号a, bを宣言する．reg型はC言語の変数に相当する．
- wire型の信号cを宣言する．wire型はハードウェア記述言語に固有のもの．
- 継続的代入assign は，wire型の信号cをa & bに接続する．initialブロックやalways@ブロックの外に記述する．
- & はANDの論理演算子．
- always@ブロックは，@以降に書かれた事象が発生するたびに繰り返し実行される．always@(*) では，何らかの入力が変化した事象となる．
- initialブロックの中の <= はノンブロッキング代入と呼ばれ，reg型の信号への代入を表す． a <= 0; はreg型の信号aに値0を代入する．wire型にノンブロッキング代入は使えない．



AND gate

code011.v

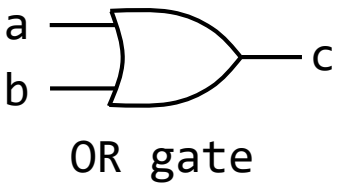
```
module main ();
  reg a, b;
  wire c;
  assign c = a & b;

  initial begin
    #10 a <= 0; b <= 0;
    #10 a <= 0; b <= 1;
    #10 a <= 1; b <= 0;
    #10 a <= 1; b <= 1;
  end
  always@(*) #1 $display("%2d: %d %d -> %d", $time, a, b, c);
endmodule
```

Simulation output

```
11: 0 0 -> 0
21: 0 1 -> 0
31: 1 0 -> 0
41: 1 1 -> 1
```

code012.v ORゲート，論理演算子，算術演算子

- code012.v をシミュレーションして，その表示を確認すること．
- | はORの論理演算子．
- 論理演算子には，単項演算子の ~ (NOT), 2項演算子として & (AND), | (OR), ^ (EXOR)がある．
- 算術演算子には，+ (加算), - (減算), * (乗算), / (除算), % (剰余)がある．
- これらの論理演算子，算術演算子はC言語と同じ．



OR gate

code012.v

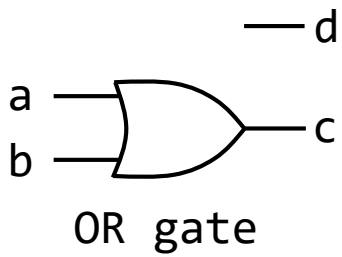
```
module main ();
  reg a, b;
  wire c;
  assign c = a | b;

  initial begin
    #10 a <= 0; b <= 0;
    #10 a <= 0; b <= 1;
    #10 a <= 1; b <= 0;
    #10 a <= 1; b <= 1;
  end
  always@(*) #1 $display("%2d: %d %d -> %d", $time, a, b, c);
endmodule
```

Simulation output

```
11: 0 0 -> 0
21: 0 1 -> 1
31: 1 0 -> 1
41: 1 1 -> 1
```

code013.v ORゲート，不定値とハイインピーダンス

- code013.v をシミュレーションして，その表示を確認すること．
- wire dはどこにも接続されていない．シミュレーション結果は？
- 信号線の取り得る値には，0, 1, x, z がある．xは不定値を，zはハイインピーダンスを表す．
- どこにも接続されていないwire，あるいは明示的にハイインピーダンスに設定したwireはzとなる．
- 初期化されていないreg型の信号や，不定値を用いた演算結果などはxとなる．
- 意図的にx, zとしていないのにx, zが出力される場合，コードに記述ミスがあることが多いので，コードの記述を見直すと良い．



code013.v

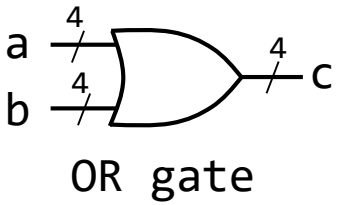
```
module main ();
  reg a, b;
  wire c, d;
  assign c = a | b;

  initial begin
    #10 a <= 0;
    #10 a <= 0; b <= 1;
    #10 a <= 1; b <= 0;
    #10 a <= 1; b <= 1;
  end
  always@(*) #1 $display("%2d: %d %d -> %d %d", $time, a, b, c, d);
endmodule
```

Simulation output

```
11: 0 x -> x z
21: 0 1 -> 1 z
31: 1 0 -> 1 z
41: 1 1 -> 1 z
```

code014.v 複数本の信号線(バス), 数値の表現

- code014.v をシミュレーションして, その表示を確認すること.
- Verilog HDLでは, 2本以上の信号線の束をバス(bus)と呼ぶ.
- reg型, wire型の信号線をバスとして宣言するには, reg, wire の後に [3:0] の様に本数を指定する. 例えば reg [3:0] a は, a[3], a[2], a[1], a[0]の4本から成るバスを宣言する.
- code014.v では, 4ビット幅のバスとしてreg型a, bを, 4ビット幅のバスとしてwire型cを宣言する.
- 数値を表現するためには, '(シングルクォーテーション)より前の数字がビット幅を表し, 'の後のbが2進法であることを表す(その他, 16進法h, 10進法d, 8進法oがある). 例えば, 4'b1010 は2進法で示された4ビットの1010となる. 数値の表現では大文字, 小文字は区別されない. 4'b1010 と 4'B1010 と 4'hAは同じ値となる.
  ビット幅を省略すると32ビットとなる. 基数を指定しないと10進法となる.
- システムタスク$displayでは, 2進法で表示するための %b を利用できる.



OR gate

code014.v

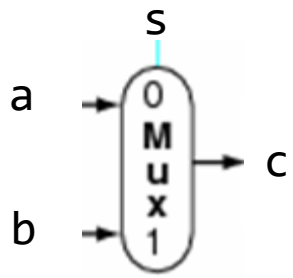
```
module main ();
  reg  [3:0] a, b;
  wire [3:0] c;
  assign c = a | b;

  initial begin
    #10 a <= 4'b1010; b <= 4'b1100;
  end
  always@(*) #1 $display("%2d: %b %b -> %b", $time, a, b, c);
endmodule
```

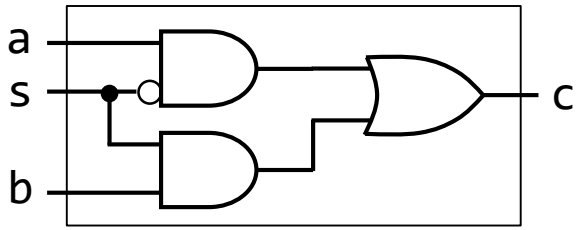
Simulation output

```
11: 1010 1100 -> 1110
```

code015.v マルチプレクサ

- code015.v をシミュレーションして，その表示を確認すること．
- Multiplexer（マルチプレクサ）のVerilog HDL記述を考える．
  - s が 0 であれば a を出力として，s が1であれば b を出力とする回路．





code015.v

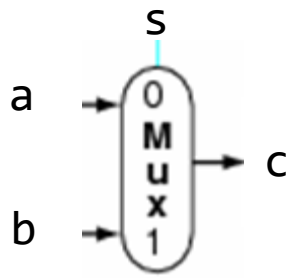
```
module main ();
  reg  a, b, s;
  wire c;
  assign c = (a & ~s) | (s & b);
  initial begin
    #10 s <= 0; a <= 0; b <= 0;
    #10 s <= 0; a <= 0; b <= 1;
    #10 s <= 0; a <= 1; b <= 0;
    #10 s <= 0; a <= 1; b <= 1;
    #10 s <= 1; a <= 0; b <= 0;
    #10 s <= 1; a <= 0; b <= 1;
    #10 s <= 1; a <= 1; b <= 0;
    #10 s <= 1; a <= 1; b <= 1;
  end
  always@(*) #1 $display("%2d: %d %d %d -> %b", $time, s, a, b, c);
endmodule
```

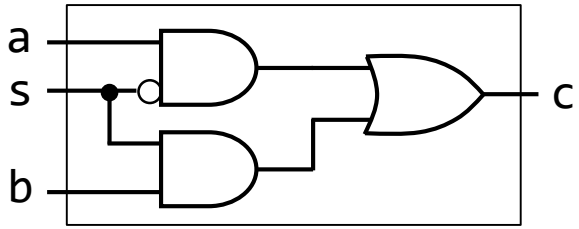
Simulation output

```
      s, a, b    c
11:  0  0  0 -> 0
21:  0  0  1 -> 0
31:  0  1  0 -> 1
41:  0  1  1 -> 1
51:  1  0  0 -> 0
61:  1  0  1 -> 1
71:  1  1  0 -> 0
81:  1  1  1 -> 1
```

code016.v 3項演算子とマルチプレクサ

- code016.v をシミュレーションして，その表示を確認すること．
- マルチプレクサのVerilog HDL記述を考える．
- 3項演算子の条件演算子 (? :) を使っても同じ結果になる．この記述の方が簡潔でわかりやすい．





code016.v

```
module main ();
  reg  a, b, s;
  wire c;
  assign c = s ? b : a;
  initial begin
    #10 s <= 0; a <= 0; b <= 0;
    #10 s <= 0; a <= 0; b <= 1;
    #10 s <= 0; a <= 1; b <= 0;
    #10 s <= 0; a <= 1; b <= 1;
    #10 s <= 1; a <= 0; b <= 0;
    #10 s <= 1; a <= 0; b <= 1;
    #10 s <= 1; a <= 1; b <= 0;
    #10 s <= 1; a <= 1; b <= 1;
  end
  always@(*) #1 $display("%2d: %d %d %d -> %b", $time, s, a, b, c);
endmodule
```

Simulation output

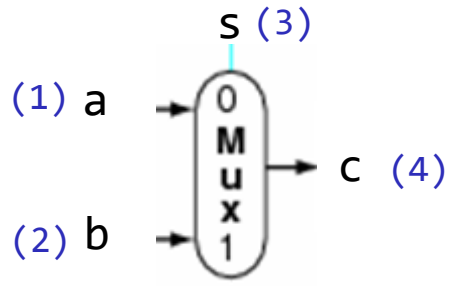
```
     s, a, b    c
11:  0  0  0 -> 0
21:  0  0  1 -> 0
31:  0  1  0 -> 1
41:  0  1  1 -> 1
51:  1  0  0 -> 0
61:  1  0  1 -> 1
71:  1  1  0 -> 0
81:  1  1  1 -> 1
```

code017.v モジュールのインスタンス化

- code017.v をシミュレーションして，その表示を確認すること．
- モジュール名とインスタンス名を記述して，入出力端子名を列挙する．列挙した順序で配線が接続される．
- C言語の関数呼び出しに似ている．この例では m_mux というモジュール名のインスタンス m_mux0 を生成し，m_topモジュールの a, b, s, c をインスタンス m_mux0 の a, b, s, c に接続している．

code017.v

```
module m_top ();
  reg  a, b, s;
  wire c;
  initial begin
    #10 s <= 0; a <= 0; b <= 0;
    #10 s <= 0; a <= 0; b <= 1;
    #10 s <= 0; a <= 1; b <= 0;
    #10 s <= 0; a <= 1; b <= 1;
    #10 s <= 1; a <= 0; b <= 0;
    #10 s <= 1; a <= 0; b <= 1;
    #10 s <= 1; a <= 1; b <= 0;
    #10 s <= 1; a <= 1; b <= 1;
  end
  always@(*) #1 $display("%2d: %d %d %d -> %b", $time, s, a, b, c);
  m_mux m_mux0 (a, b, s, c);
endmodule
            (1) (2) (3) (4)
module m_mux (a, b, s, c);
  input  wire a, b, s;
  output wire c;
  assign c = s ? b : a;
endmodule
```



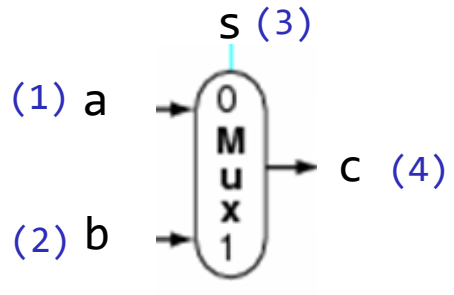
Simulation output

```
      s, a, b    c
11:  0  0  0 -> 0
21:  0  0  1 -> 0
31:  0  1  0 -> 1
41:  0  1  1 -> 1
51:  1  0  0 -> 0
61:  1  0  1 -> 1
71:  1  1  0 -> 0
81:  1  1  1 -> 1
```

モジュールのインスタンス化の補足

- モジュール名とインスタンス名を記述して，入出力端子名を列挙する．列挙した順序で配線が接続される．
- この例では m_mux というモジュール名のインスタンス m_mux0 を生成し，m_topモジュールの a, b, s, c をインスタンス m_mux0 の w_a, w_b, w_s, w_c に順番に接続している．
- インスタンス化するモジュールの入出力端子名とそれを利用するモジュールの配線の名前は一致しなくても良い．

```
module m_top ();
  reg  a, b, s;
  wire c;
  initial begin
    #10 s <= 0; a <= 0; b <= 0;
    #10 s <= 0; a <= 0; b <= 1;
    #10 s <= 0; a <= 1; b <= 0;
    #10 s <= 0; a <= 1; b <= 1;
    #10 s <= 1; a <= 0; b <= 0;
    #10 s <= 1; a <= 0; b <= 1;
    #10 s <= 1; a <= 1; b <= 0;
    #10 s <= 1; a <= 1; b <= 1;
  end
  always@(*) #1 $display("%2d: %d %d %d -> %b", $time, s, a, b, c);
  m_mux m_mux0 (a, b, s, c);
endmodule
                (1)  (2)   (3)   (4)
module m_mux (w_a, w_b, w_s, w_c);
  input  wire w_a, w_b, w_s;
  output wire w_c;
  assign w_c = w_s ? w_b : w_a;
endmodule
```



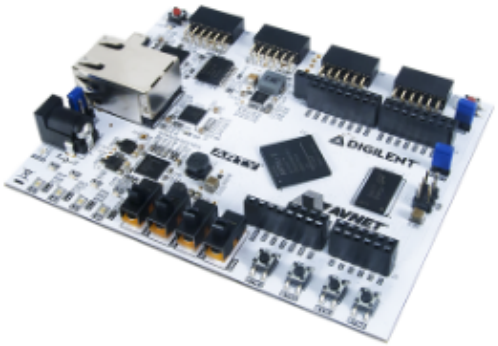
Simulation output

```
     s, a, b    c
11:  0  0  0 -> 0
21:  0  0  1 -> 0
31:  0  1  0 -> 1
41:  0  1  1 -> 1
51:  1  0  0 -> 0
61:  1  0  1 -> 1
71:  1  1  0 -> 0
81:  1  1  1 -> 1
```

ACRiルームで利用するFPGAボード Digilent Arty A7-35T

Arty A7

The Arty A7, formerly known as the Arty, is a ready-to-use development platform designed around the Artix-7™ Field Programmable Gate Array (FPGA) from Xilinx. It was designed specifically for use as a MicroBlaze Soft Processing System. When used in this context, the Arty A7 becomes the most flexible processing platform you could hope to add to your collection, capable of adapting to whatever your project requires. Unlike other Single Board Computers, the Arty A7 isn't bound to a single set of processing peripherals: One moment it's a communication powerhouse chock-full of UARTs, SPIs, IICs, and an Ethernet MAC, and the next it's a meticulous timekeeper with a dozen 32-bit timers.



Store

Reference Manual | Technical Support

Arty A7

Artix-7 FPGA Development Board

Features

- Programmable over JTAG and Quad-SPI Flash
- On-chip analog-to-digital converter

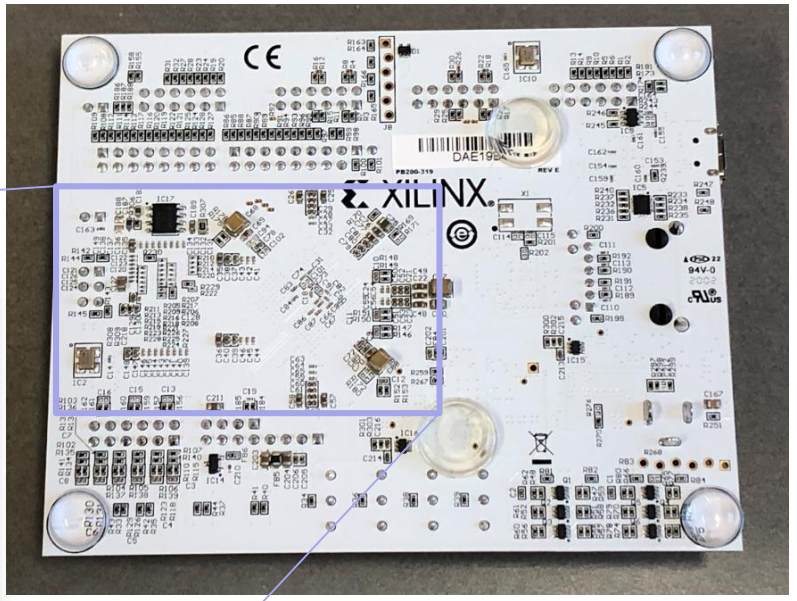
Key Specifications

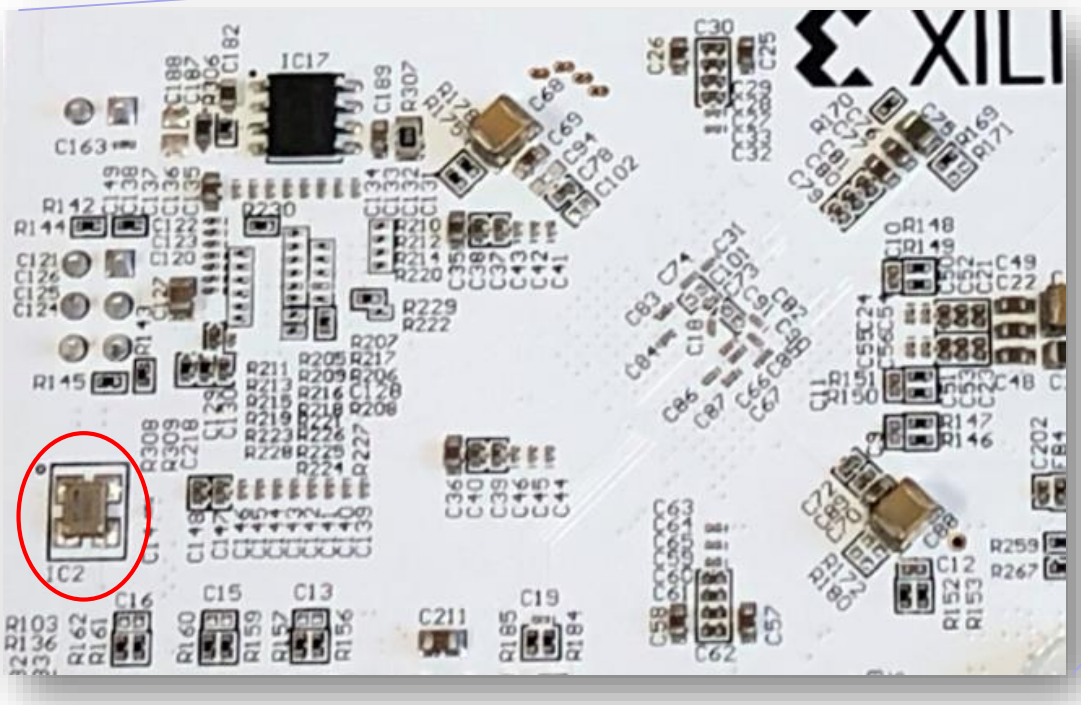
FPGA Part #	XC7A35TICSG324-1L (XC7A100TCSG324-1*)
Logic Slices	5,200 (15,850*)
Block RAM	1,800 Kbits (4,860* Kbits)
DSP Slices	90 (240*)
DDR3	256 MB @ 667 MHz
Internal clock	450 MHz+
Quad-SPI Flash	16 MB
Ethernet	10/100 Mbps

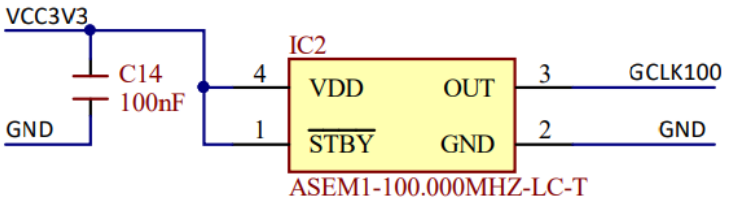
https://reference.digilentinc.com/reference/programmable-logic/arty-a7/start

ASEM1-100.000MHZ-LC-T 100MHz 発振器

- FPGA ボードに実装されている発振器が 100MHz のクロック信号を生成し，それが FPGA の E3 という名前のピンに接続される．（FPGA の入力信号となる．）







FPGA constraint file, XDC (Xilinx Design Constraints)

- main11.xdc の内容を Ubuntu のターミナルあるいは Vivado で確認すること.
- 拡張子が xdc のファイルは，制約 (constraint) を与えるために利用する.
- 制約ファイル main11.xdc の1行目では，w_clk という信号を E3 というピン（100MHzのクロック信号）に割り当てる制約を追加する.
  - w_clk は論理合成のためのトップモジュール m_main としてVerilog HDL記述で列挙した信号名
- 信号をピンを割り当てる制約が無い場合，その信号はVivadoによって自動的に適切なピンに割り当てられる.
- 2行目で、入力ピン w_clk が，10.00ns (100MHz)のクロックであることを指定する.
- このピンを LVCMOS33 (low voltage CMOS 3.3V) とする制約を追加している. この制約について，本演習では詳細を理解する必要はない.

main11.xdcの最初の2行

```
set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33} [get_ports { w_clk }];
create_clock -add -name sys_clk -period 10.00 [get_ports {w_clk}];
```

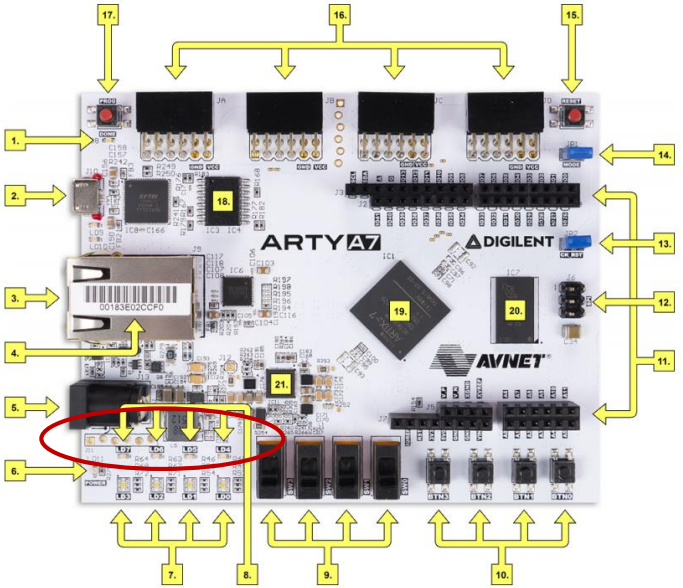
```
module m_main (w_clk);
  input  wire w_clk;
  .
  .
  .
```

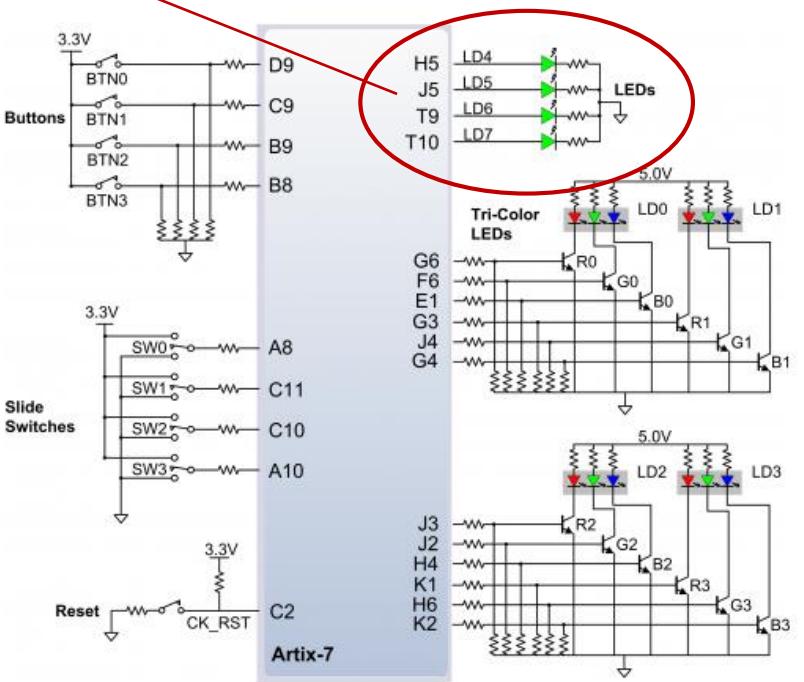
FPGA constraint file, XDC (Xilinx Design Constraints)

- main11.xdc の2行目以降では，w_led[0] の信号を H5 のピンに割り当てる制約を追加する．
  同様に，w_led[1], w_led[2], w_led[3] に，J5, T9, T10 のピンを割り当てる．

main11.xdc

```
set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33} [get_ports { w_clk }];
create_clock -add -name sys_clk -period 10.00 [get_ports {w_clk}];

set_property -dict { PACKAGE_PIN H5  IOSTANDARD LVCMOS33} [get_ports { w_led[0] }];
set_property -dict { PACKAGE_PIN J5  IOSTANDARD LVCMOS33} [get_ports { w_led[1] }];
set_property -dict { PACKAGE_PIN T9  IOSTANDARD LVCMOS33} [get_ports { w_led[2] }];
set_property -dict { PACKAGE_PIN T10 IOSTANDARD LVCMOS33} [get_ports { w_led[3] }];
```





	Description	Callout	Description	Callout	Description
	FPGA programming DONE LED	8	User RGB LEDs	15	chipKIT processor reset

ACRi ルームで使う2種類のパスワード

- ACRi ルームに関するよくある質問(FAQ)
  - https://gw.acri.c.titech.ac.jp/wp/manual/faq
- 現在のACRiルームでは，次の2種類のアカウントを用いて運用

  [1] ACRiルームのWeb上の予約システムのアカウント

  [2] ACRiルームのLinuxサーバにログインするためのアカウント

  混乱を避けるために，[1] と [2] のパスワードを同じものに設定すると良い．

> **パスワードを忘れてしまいました。**
>
> 予約システムのパスワードを忘れた場合には、ログイン画面の「パスワードをお忘れですか？」のリンクから再発行の手続きを行ってください。
>
> サーバーのパスワードを忘れた場合には、acri-room at acri dot c.titech.ac.jp (at, dot は適切に置き換え) まで問い合わせてください。

Department of Computer Science
Course number: CSC.T341

2023年度の講義と演習は，学術国際情報センター3階 情報工学系計算機室で実施します．

コンピュータ論理設計
Computer Logic Design

3. ハードウェア記述言語：組合せ回路 (2)
Hardware Description Language: Combinational Circuit (2)

吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
講義：月曜日 10:45-12:25, 木曜日 10:45-12:25

Sample Verilog HDL code

- ACRi Room のサーバーにリモートデスクトップでログインする.
  - /home/tu_kise/cld/lec3/ にサンプルのコードがあるので，Ubuntu のターミナルで次のコマンドを入力して，自分のディレクトリにコピーする.
  - /home/tu_kise は automount のディレクトリなので，アクセスしないとファイルが見えない．tabキーによる補完がうまく動作しないことがあるので注意する.

```
$ cd
$ mkdir cld
$ cd cld
$ cp /home/tu_kise/cld/lec3/* .
```

- code010.v をシミュレーションするためには.
  - リモートデスクトップで接続した Ubuntu のターミナルで次のコマンドを入力する.
  - コマンド iverilog でコンパイルして，生成される a.out を実行する.

```
$ iverilog code010.v
$ ./a.out
```

code010.v C言語で書いたかもしれないコード

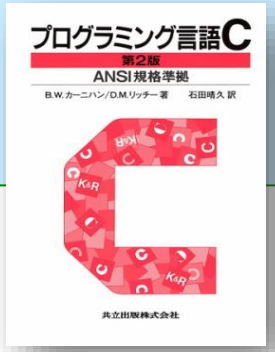
- code010.v をシミュレーションして，その表示を確認すること．
- 整数integerとforループを用いた温度変換プログラムの例．
- 整数型のfahr, celsiusを定義．
- C言語の様に演算子++は使えない．fahr++ という記述はエラーとなるので注意．

```
$ iverilog code010.v
$ ./a.out
```

code010.v

```
module main ();
  integer fahr, celsius;
  initial begin
    for (fahr = 0; fahr <= 300; fahr = fahr + 20) begin
      celsius = 5*(fahr-32) / 9;
      $display("%3d %6d", fahr, celsius);
    end
  end
endmodule
```

```
  0    -17
 20     -6
 40      4
 60     15
 80     26
100     37
120     48
140     60
160     71
180     82
200     93
220    104
240    115
260    126
280    137
300    148
```



スライドPDFからコピーすると正しく動作しないことがあるので，コードは /home/tu_kise/ からコピーしたものを使うこと．

Some rules for following lectures and exercises

- モジュールの名前には m_ から始まる名前を使う. wire型の信号の名前には w_ から始まる名前を使う. reg型の名前には r_ から始まる名前を使う.
- シミュレーションの最上位のモジュール(トップモジュール)には m_top という名前を使う.
  - $display などのシステムタスクは m_top の中でしか用いてはいけない.
- 論理合成のトップモジュールには m_main という名前を使う.
- Nameという名前のモジュールのインスタンス名には Nameに数字を付加した名前を使う.

ルールを適用したVerilog HDL記述の例

```
module m_top ();
  reg  r_a, r_b, r_s;
  wire w_c;
  initial begin
    #10 r_s <= 0; r_a <= 0; r_b <= 0;
    #10 r_s <= 0; r_a <= 0; r_b <= 1;
  end
  always@(*) #1 $display("%2d: %d %d %d -> %b", $time, r_s, r_a, r_b, w_c);
  m_mux m_mux0 (r_a, r_b, r_s, w_c);
endmodule

module m_mux (w_a, w_b, w_s, w_c);
  input  wire w_a, w_b, w_s;
  output wire w_c;
  assign w_c = w_s ? w_b : w_a;
endmodule
```

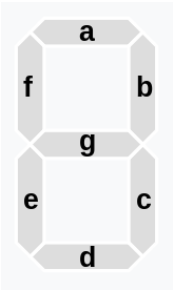
code018.v case文を用いたLEDデコーダ

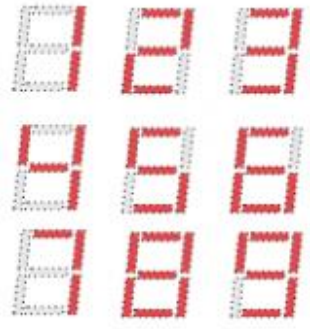
- code018.v をシミュレーションして，その表示を確認すること．
- 0～9を表示する seven-segment LED decoder の例を示す．
- 場合分けの処理を記述するための case文 がある．記述はC言語と同様．
- モジュールm_7segledでは，入力の値により，点灯させるLEDのビットを1とする．
  - r_led の MSBから，LEDのabcdefgのセグメントを割り当てる．

```
module m_7segled (w_in, r_led);
  input  wire [3:0] w_in;
  output reg  [6:0] r_led;
  always @(*) begin
    case (w_in)
      4'd0   : r_led <= 7'b1111110;
      4'd1   : r_led <= 7'b0110000;
      4'd2   : r_led <= 7'b1101101;
      4'd3   : r_led <= 7'b1111001;
      4'd4   : r_led <= 7'b0110011;
      4'd5   : r_led <= 7'b1011011;
      4'd6   : r_led <= 7'b1011111;
      4'd7   : r_led <= 7'b1110000;
      4'd8   : r_led <= 7'b1111111;
      4'd9   : r_led <= 7'b1111011;
      default: r_led <= 7'b0000000;
    endcase
  end
endmodule
```

```
module m_top ();
  reg  [3:0] r_in;
  wire [6:0] w_led;
  integer i;
  initial
    for (i=0; i<=15; i=i+1) begin r_in <= i; #10; end
  initial       $display("      abcdefg");
  always@(*) #1 $display(" %x -> %b", r_in, w_led);

  m_7segled m_7segled0 (r_in, w_led);
endmodule
```

```
      abcdefg
0 -> 1111110
1 -> 0110000
2 -> 1101101
3 -> 1111001
4 -> 0110011
5 -> 1011011
6 -> 1011111
7 -> 1110000
8 -> 1111111
9 -> 1111011
a -> 0000000
b -> 0000000
c -> 0000000
d -> 0000000
e -> 0000000
f -> 0000000
```





code018.v

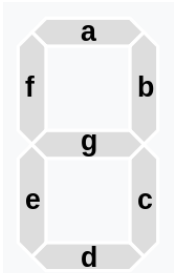
code019.v 3項演算子を用いたLEDデコーダ

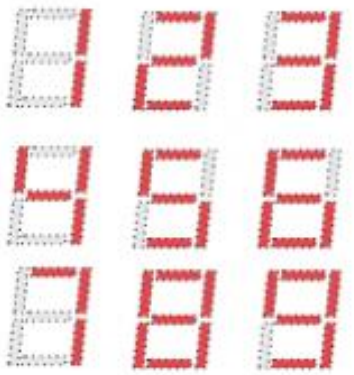
- code019.v をシミュレーションして，その表示を確認すること．
- Seven-segment LED decoder の別の例を示す．
- 関係演算子(==)は等しい時に1'b1となり，そうでなければ1'b0となる．
- code019.v ではreg型は使っていない．このため，m_7segled からは組合せ回路が合成される．
- code018.v の m_7segled から，組合せ回路が合成されるか？順序回路が合成されるか？
  - reg型の信号が常にレジスタに合成されるという訳ではない．

code019.v （m_topの記述はcode18.vと同じ）

```
module m_7segled (w_in, w_led);
  input  wire [3:0] w_in;
  output wire [6:0] w_led;

  assign w_led = (w_in==4'd0) ? 7'b1111110 :
                 (w_in==4'd1) ? 7'b0110000 :
                 (w_in==4'd2) ? 7'b1101101 :
                 (w_in==4'd3) ? 7'b1111001 :
                 (w_in==4'd4) ? 7'b0110011 :
                 (w_in==4'd5) ? 7'b1011011 :
                 (w_in==4'd6) ? 7'b1011111 :
                 (w_in==4'd7) ? 7'b1110000 :
                 (w_in==4'd8) ? 7'b1111111 :
                 (w_in==4'd9) ? 7'b1111011 :
                 7'b0000000;
endmodule
```

```
      abcdefg
0 -> 1111110
1 -> 0110000
2 -> 1101101
3 -> 1111001
4 -> 0110011
5 -> 1011011
6 -> 1011111
7 -> 1110000
8 -> 1111111
9 -> 1111011
a -> 0000000
b -> 0000000
c -> 0000000
d -> 0000000
e -> 0000000
f -> 0000000
```





code020.v case文ではすべての入力を定義する

- code020.v をシミュレーションして，その表示を確認すること．
- Seven-segment LED decoder の別の例を示す．
- code020.v の m_7segled から，組合せ回路が合成されるか？順序回路が合成されるか？
  - w_in が 4'ha の時に，どうして 7'b1111011 が出力されるのか？

code018.v

```
module m_7segled (w_in, r_led);
  input  wire [3:0] w_in;
  output reg  [6:0] r_led;
  always @(*) begin
    case (w_in)
      4'd0   : r_led <= 7'b1111110;
      4'd1   : r_led <= 7'b0110000;
      4'd2   : r_led <= 7'b1101101;
      4'd3   : r_led <= 7'b1111001;
      4'd4   : r_led <= 7'b0110011;
      4'd5   : r_led <= 7'b1011011;
      4'd6   : r_led <= 7'b1011111;
      4'd7   : r_led <= 7'b1110000;
      4'd8   : r_led <= 7'b1111111;
      4'd9   : r_led <= 7'b1111011;
      default: r_led <= 7'b0000000;
    endcase
  end
endmodule
```

```
      abcdefg
0 -> 1111110
1 -> 0110000
2 -> 1101101
3 -> 1111001
4 -> 0110011
5 -> 1011011
6 -> 1011111
7 -> 1110000
8 -> 1111111
9 -> 1111011
a -> 0000000
b -> 0000000
c -> 0000000
d -> 0000000
e -> 0000000
f -> 0000000
```

code020.v

```
module m_7segled (w_in, r_led);
  input  wire [3:0] w_in;
  output reg  [6:0] r_led;
  always @(*) begin
    case (w_in)
      4'd0   : r_led <= 7'b1111110;
      4'd1   : r_led <= 7'b0110000;
      4'd2   : r_led <= 7'b1101101;
      4'd3   : r_led <= 7'b1111001;
      4'd4   : r_led <= 7'b0110011;
      4'd5   : r_led <= 7'b1011011;
      4'd6   : r_led <= 7'b1011111;
      4'd7   : r_led <= 7'b1110000;
      4'd8   : r_led <= 7'b1111111;
      4'd9   : r_led <= 7'b1111011;
    endcase
  end
endmodule
```

```
      abcdefg
0 -> 1111110
1 -> 0110000
2 -> 1101101
3 -> 1111001
4 -> 0110011
5 -> 1011011
6 -> 1011111
7 -> 1110000
8 -> 1111111
9 -> 1111011
a -> 1111011
b -> 1111011
c -> 1111011
d -> 1111011
e -> 1111011
f -> 1111011
```

code021.v if文を用いたLEDデコーダ

- code021.v をシミュレーションして，その表示を確認すること．
- Seven-segment LED decoder の別の例を示す．
- case文ではなく，if文 (if else) を用いて記述することもできる．
  - code18.v と code21.v は同じ出力となる．

code018.v

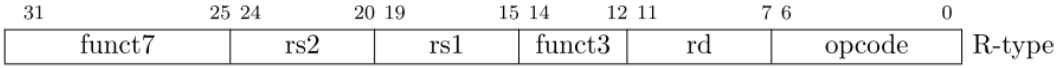
```
module m_7segled (w_in, r_led);
  input  wire [3:0] w_in;
  output reg  [6:0] r_led;
  always @(*) begin
    case (w_in)
      4'd0   : r_led <= 7'b1111110;
      4'd1   : r_led <= 7'b0110000;
      4'd2   : r_led <= 7'b1101101;
      4'd3   : r_led <= 7'b1111001;
      4'd4   : r_led <= 7'b0110011;
      4'd5   : r_led <= 7'b1011011;
      4'd6   : r_led <= 7'b1011111;
      4'd7   : r_led <= 7'b1110000;
      4'd8   : r_led <= 7'b1111111;
      4'd9   : r_led <= 7'b1111011;
      default: r_led <= 7'b0000000;
    endcase
  end
endmodule
```

code021.v

```
module m_7segled (w_in, r_led);
  input  wire [3:0] w_in;
  output reg  [6:0] r_led;
  always @(*) begin
    if      (w_in==4'd0) r_led <= 7'b1111110;
    else if (w_in==4'd1) r_led <= 7'b0110000;
    else if (w_in==4'd2) r_led <= 7'b1101101;
    else if (w_in==4'd3) r_led <= 7'b1111001;
    else if (w_in==4'd4) r_led <= 7'b0110011;
    else if (w_in==4'd5) r_led <= 7'b1011011;
    else if (w_in==4'd6) r_led <= 7'b1011111;
    else if (w_in==4'd7) r_led <= 7'b1110000;
    else if (w_in==4'd8) r_led <= 7'b1111111;
    else if (w_in==4'd9) r_led <= 7'b1111011;
    else                 r_led <= 7'b0000000;
  end
endmodule
```

code022.v ビット選択

- code022.v をシミュレーションして，その表示を確認すること．
- ビット選択の例を示す．バスは多ビットの束で表現されるので，バスから選択するビットの範囲を指定する．RISC-Vアーキテクチャの機械命令で用いられるR形式の命令から各フィールドを選択する例．




```
module m_top ();
  reg  [31:0] r_ir = 32'h12345678;
  wire [6:0]  w_fct7, w_op;
  wire [4:0]  w_rs2, w_rs1, w_rd;
  wire [2:0]  w_fct3;
  initial begin #1
    $display(" %x -> %x %x %x %x %x %x", r_ir, w_fct7, w_rs2, w_rs1, w_fct3, w_rd, w_op);
    $display(" %x -> %d %d %d %d %d %d", r_ir, w_fct7, w_rs2, w_rs1, w_fct3, w_rd, w_op);
  end
  m_decode m_d0 (r_ir, w_fct7, w_rs2, w_rs1, w_fct3, w_rd, w_op);
endmodule

module m_decode (w_ir, w_fct7, w_rs2, w_rs1, w_fct3, w_rd, w_op);
  input  wire [31:0] w_ir;
  output wire [6:0]  w_fct7, w_op;
  output wire [4:0]  w_rs2, w_rs1, w_rd;
  output wire [2:0]  w_fct3;
  assign w_fct7 = w_ir[31:25];
  assign w_rs2  = w_ir[24:20];
  assign w_rs1  = w_ir[19:15];
  assign w_fct3 = w_ir[14:12];
  assign w_rd   = w_ir[11:7];
  assign w_op   = w_ir[6:0];
endmodule
```

```
12345678 -> 09 03 08 5 0c 78
12345678 ->   9  3  8 5 12 120
```

Simulation output

code023.v ビットの連結と複製

- code023.v をシミュレーションして，その表示を確認すること．
- ビットの連結 (concatenation) の例を示す．
- 連結演算子 ({ } ，波括弧 curly brackets) は，幾つかの信号を連結してビット長の大きい1つのバスにできる．4ビットの信号 w_a, w_b を連結するには {w_a, w_b} と記述する．4ビットの信号 w_a, w_b, w_c を連結するには {w_a, w_b, w_c} と記述する．
- ある信号を複製してビット長の大きい1つのバスにできる．例えば，4ビットの信号 w_a を3回複製して連結するには {3{w_a}} と記述する．例えば，{4{w_a}} と {w_a, w_a, w_a, w_a} は同じビット列となる．
- 最後の例で示した下位ビットのMSBを複製して上位ビットを補填する操作は，2の補数で表現された符号付きの整数を符号拡張する際に用いられる．後の講義で解説する．

code023.v

```
module m_top ();
  reg [3:0] r_a = 4'b1001;
  reg [3:0] r_b = 4'b0101;
  reg [3:0] r_c = 4'b1111;
  initial #1 begin
    $display("%b", {r_a, r_b});
    $display("%b", {r_a, r_b, r_c});
    $display("%b", {2{r_a}});
    $display("%b", {3{r_a}});
    $display("%b", {4{r_a}});
    $display("%b", {{4{r_a[3]}}, r_a});
    $display("%b", {{4{r_b[3]}}, r_b});
  end
endmodule
```

Simulation output

```
10010101
100101011111
10011001
100110011001
1001100110011001
11111001
00000101
```

w_bの値を赤色で強調した．

code024.v 関係演算子

- code024.v をシミュレーションして，その表示を確認すること．
- 関係演算子 (>, <, >=, <=, ==, !=) の例を示す．
- 例えば，w_a >= w_b は，w_a の値が w_b の値以上であれば 1'b1，そうでなければ 1'b0 となる．
- C言語と同様．
- ノンブロッキング代入の演算子 <= と 関係演算子 <= は同じ記述だが，文法的に区別できる．
  この演習では (w_a >= w_b) の様に，関係演算子の比較の前後に ( ) を追加して明示的に区別する．

code024.v

```
module m_top ();
  reg [3:0] r_a = 4'd7;
  reg [3:0] r_b = 4'd8;
  initial #1 begin
    $display("%b", (r_a> r_b));
    $display("%b", (r_a< r_b));
    $display("%b", (r_a>=r_b));
    $display("%b", (r_a<=r_b));
    $display("%b", (r_a==r_b));
    $display("%b", (r_a!=r_b));
  end
endmodule
```

Simulation output

```
0
1
0
1
0
1
```

code025.v 論理シフト演算

- code025.v をシミュレーションして，その表示を確認すること．
- 論理シフト演算 (>>, <<) の例を示す．
- C言語と同様．
- 例えば，w_a << 3 は，w_a の値を左に3ビット移動させ，下位の3ビットは0となる．同様に，w_b >> 2 では，w_bの値を右に2ビット移動させ，上位の2ビットは0となる．
- 論理シフト演算では，シフトさせるビット数としてワイヤ型やレジスタ型の信号を用いてもよい．

code025.v

```
module m_top ();
  reg [7:0] r_a = 8'b11110101;
  reg [2:0] r_s = 3'd3;
  initial #1 begin
    $display("%b", (r_a>>0));
    $display("%b", (r_a>>1));
    $display("%b", (r_a<<1));
    $display("%b", (r_a>>r_s));
    $display("%b", (r_a<<r_s));
  end
endmodule
```

Simulation output

```
11110101
01111010
11101010
00011110
10101000
```

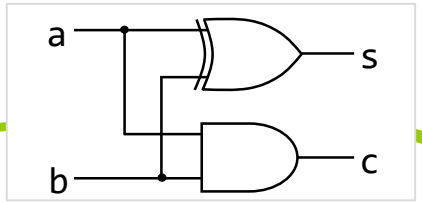
code026.v リダクション演算子

- code026.v をシミュレーションして，その表示を確認すること．
- リダクション演算子(&, |, ^) の例．
  例えば ^ はバスの全てのビットの排他的論理和となる．コメントを参照．

code026.v

```
module m_top ();
  reg  [4:0]  r_btn;
  wire [2:0]  w_led;
  initial begin
    #10 r_btn <= 5'b00000;
    #10 r_btn <= 5'b11111;
    #10 r_btn <= 5'b00010;
  end
  always@(*) #1 $display(" %b -> %b", r_btn, w_led);
  m_main m_main0 (r_btn, w_led);
endmodule

module m_main (w_btn, w_led);
  input  wire [4:0] w_btn;
  output wire [2:0] w_led;
  assign w_led[0]  = &w_btn; // same as w_btn[0] & w_btn[1] & w_btn[2] & w_btn[3] & w_btn[4]
  assign w_led[1]  = |w_btn; // same as w_btn[0] | w_btn[1] | w_btn[2] | w_btn[3] | w_btn[4]
  assign w_led[2]  = ^w_btn; // same as w_btn[0] ^ w_btn[1] ^ w_btn[2] ^ w_btn[3] ^ w_btn[4]
endmodule
```

code071.v default_nettype の追加



- code071.v と code072.v を iverilog でコンパイルすること.
- 入力ミスで，定義していない信号 M_s を用いている．定義していない信号を使うと，1ビットのwireとして扱われる．定義していない信号の使用をエラーにするにはソースコードの最初に `default_nettype none を追加すれば良い.
- code072.v ではエラーとなる．今後，すべてのソースコードに `default_nettype none を追加すること.

code071.v

```
module m_top ();
  reg  r_a, r_b;
  wire w_c, w_s;
  initial begin
    #10 r_a <= 0; r_b <= 0;
    #10 r_a <= 0; r_b <= 1;
    #10 r_a <= 1; r_b <= 0;
    #10 r_a <= 1; r_b <= 1;
  end
  always@(*) #1
    $write("%2d: %d %d -> %b %b¥n",
           $time, r_a, r_b, w_c, w_s);
  m_HA m_HA0 (r_a, r_b, w_c, w_s);
endmodule

module m_HA (w_a, w_b, w_c, w_s);
  input  wire w_a, w_b;
  output wire w_c, w_s;
  assign w_c = w_a & w_b;
  assign M_s = w_a ^ w_b;
endmodule
```

```
 1: x x -> x z
11: 0 0 -> 0 z
21: 0 1 -> 0 z
31: 1 0 -> 0 z
41: 1 1 -> 1 z
```

code072.v

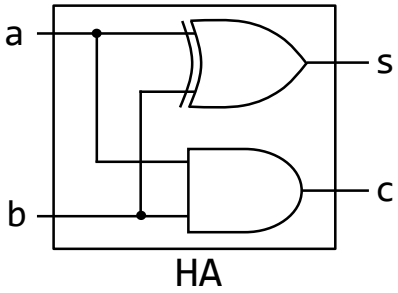
```
`default_nettype none

module m_top ();
  reg  r_a, r_b;
  wire w_c, w_s;
  initial begin
    #10 r_a <= 0; r_b <= 0;
    #10 r_a <= 0; r_b <= 1;
    #10 r_a <= 1; r_b <= 0;
    #10 r_a <= 1; r_b <= 1;
  end
  always@(*) #1
    $write("%2d: %d %d -> %b %b¥n",
           $time, r_a, r_b, w_c, w_s);
  m_HA m_HA0 (r_a, r_b, w_c, w_s);
endmodule

module m_HA (w_a, w_b, w_c, w_s);
  input  wire w_a, w_b;
  output wire w_c, w_s;
  assign w_c = w_a & w_b;
  assign M_s = w_a ^ w_b;
endmodule
```

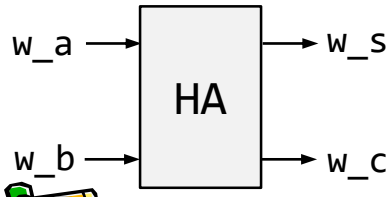
code073.v 半加算器(Half Adder)

- code073.v をシミュレーションして，その表示を確認すること.
- Half Adder, HA (半加算器)の回路とその記述の例を示す.
  - 1ビットの入力 a, b の加算をおこなう回路.
  - 入力 a, b と出力 c (carry out), s (sum) とするtruth table(真理値表)を table073 に示す.

table073

a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0





code073.v

```
`default_nettype none

module m_top ();
  reg  r_a, r_b;
  wire w_c, w_s;
  initial begin
    #10 r_a <= 0; r_b <= 0;
    #10 r_a <= 0; r_b <= 1;
    #10 r_a <= 1; r_b <= 0;
    #10 r_a <= 1; r_b <= 1;
  end
  always@(*) #1
    $write("%2d: %d %d -> %b %b¥n", $time, r_a, r_b, w_c, w_s);
  m_HA m_HA0 (r_a, r_b, w_s, w_c);
endmodule

module m_HA (w_a, w_b, w_s, w_c);
  input  wire w_a, w_b;
  output wire w_s, w_c;
  assign w_c = w_a & w_b;
  assign w_s = w_a ^ w_b;
endmodule
```

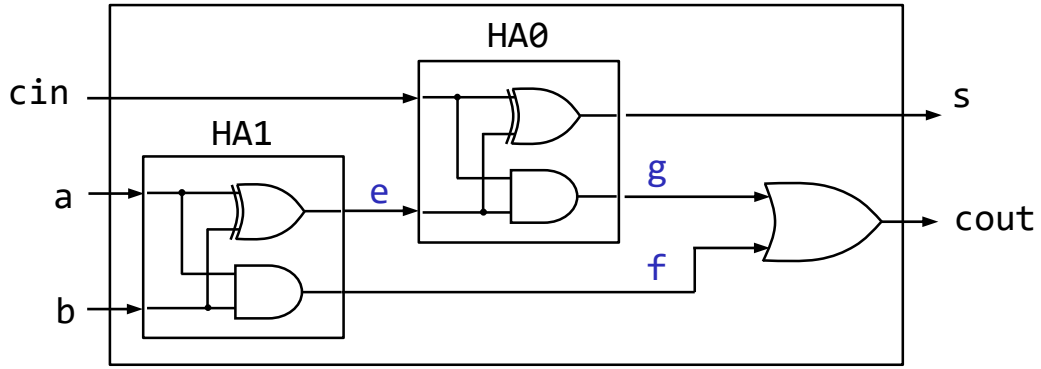
```
11: 0 0 -> 0 0
21: 0 1 -> 0 1
31: 1 0 -> 0 1
41: 1 1 -> 1 0
```

code074.v 全加算器(Full Adder)

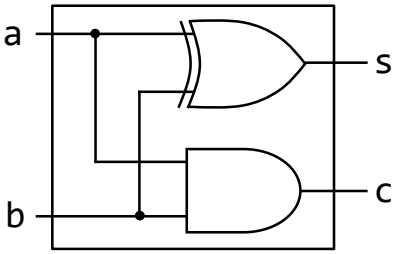
- Full Adder, FA (全加算器)の真理値表と回路を示す.
  - 1ビットの入力 a, b, cin の加算をおこなう回路.
  - 入力 a, b, cin (carry in), 出力 cout (carry out), s (sum) とする真理値表 (truth table) を示す.

table074

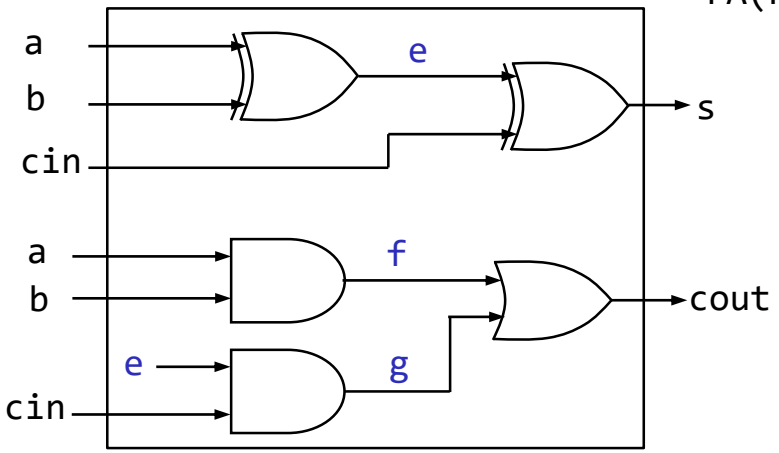
a	b	cin	cout	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



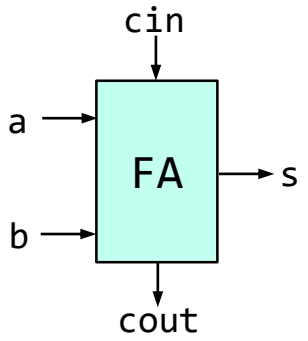
FA(Full Adder)



HA



FA(Full Adder)

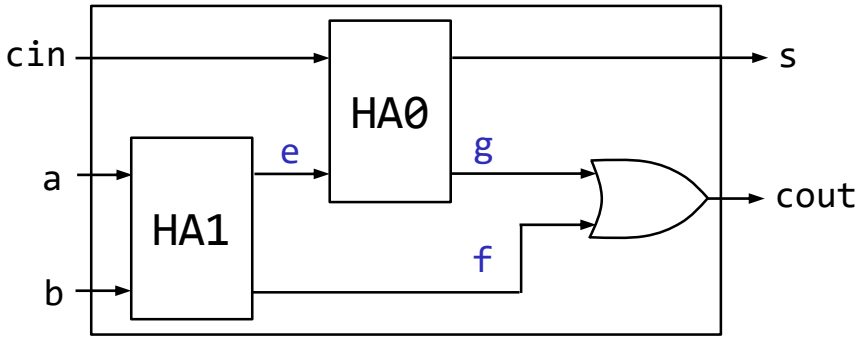


code074.v 全加算器(Full Adder)

- 全加算器として動作するように code074.v の青色の部分を変更し，シミュレーションで確認すること．
- Full Adder, FA (全加算器) の回路とその記述の一部を示す．
- 次のスライドにヒントあり．

table074

a	b	cin	cout	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

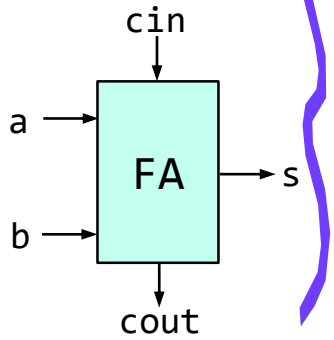


FA(Full Adder)

code074.v

```
module m_top ();
  reg  r_a, r_b, r_cin;
  wire w_s, w_cout;
  initial begin
    #10 r_a <= 0; r_b <= 0;  r_cin <= 0;
    #10 r_a <= 0; r_b <= 0;  r_cin <= 1;
    #10 r_a <= 0; r_b <= 1;  r_cin <= 0;
    #10 r_a <= 0; r_b <= 1;  r_cin <= 1;
    #10 r_a <= 1; r_b <= 0;  r_cin <= 0;
    #10 r_a <= 1; r_b <= 0;  r_cin <= 1;
    #10 r_a <= 1; r_b <= 1;  r_cin <= 0;
    #10 r_a <= 1; r_b <= 1;  r_cin <= 1;
  end
  always@(*) #1 $write("%d %d %d -> %b %b¥n",
                  r_a, r_b, r_cin, w_cout, w_s);
  m_FA m_FA0 (r_a, r_b, r_cin, w_s, w_cout);
endmodule

module m_FA (w_a, w_b, w_cin, w_s, w_cout);
  /* Please describe here by yourself */
endmodule

module m_HA (w_a, w_b, w_s, w_c);
  input  wire w_a, w_b;
  output wire w_s, w_c;
  assign w_c = w_a & w_b;
  assign w_s = w_a ^ w_b;
endmodule
```

```
0 0 0 -> 0 0
0 0 1 -> 0 1
0 1 0 -> 0 1
0 1 1 -> 1 0
1 0 0 -> 0 1
1 0 1 -> 1 0
1 1 0 -> 1 0
1 1 1 -> 1 1
```



ヒント code074.v 全加算器(Full Adder)

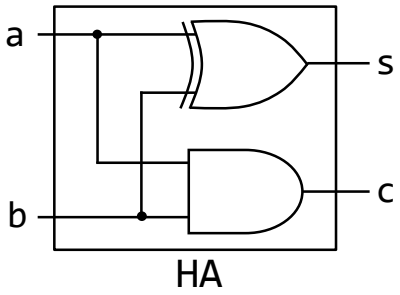
- 全加算器として動作するように code074.v の青色の部分を変更し，シミュレーションで確認すること．
- Full Adder, FA (全加算器) の回路とその記述の一部を示す．

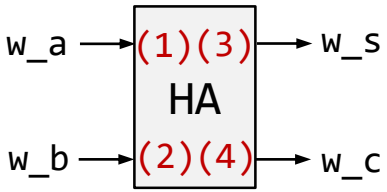
code074.v の一部

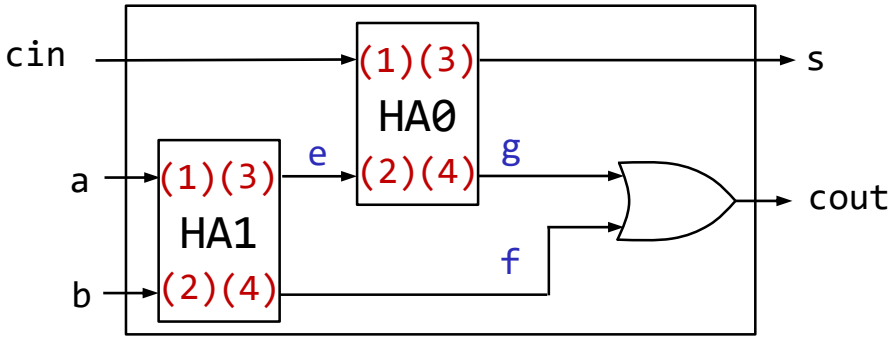
```
module m_FA (w_a, w_b, w_cin, w_s, w_cout);
  /* Please describe here by yourself */
endmodule
            (1) (2) (3) (4)
module m_HA (w_a, w_b, w_s, w_c);
  input  wire w_a, w_b;
  output wire w_s, w_c;
  assign w_c = w_a & w_b;
  assign w_s = w_a ^ w_b;
endmodule
```

ヒント: 少し記述を追加した code074.v の一部

```
module m_FA (w_a, w_b, w_cin, w_s, w_cout);
   input wire w_a, w_b, w_cin;
   output wire w_s, w_cout;
   wire w_e, w_f, w_g;
   m_HA HA0 ( /* connect wires here */ );
   m_HA HA1 ( /* connect wires here */ );
   assign w_cout = w_f | w_g;
endmodule

module m_HA (w_a, w_b, w_s, w_c);
  input  wire w_a, w_b;
  output wire w_s, w_c;
  assign w_c = w_a & w_b;
  assign w_s = w_a ^ w_b;
endmodule
```

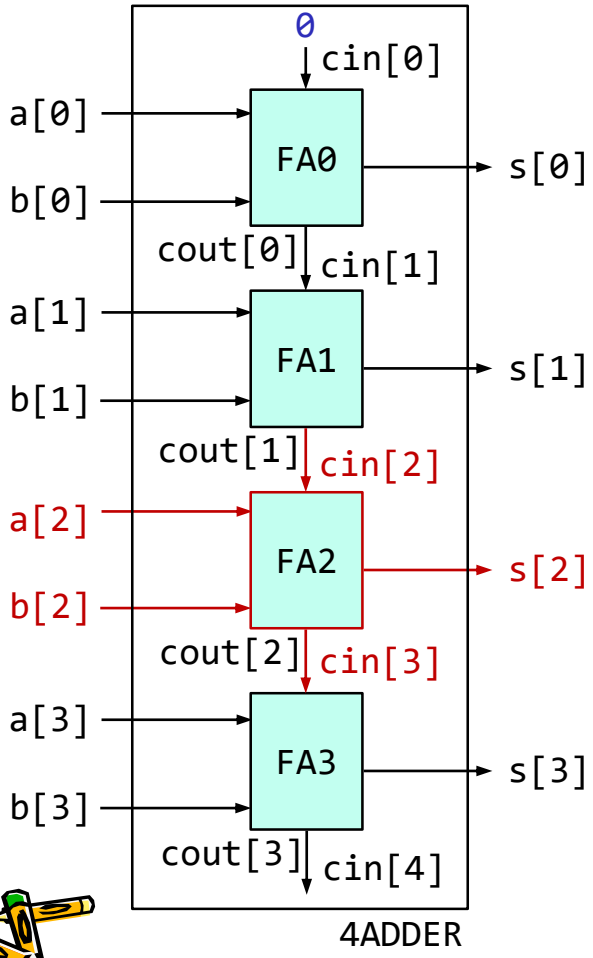






code075.v 4-bit Ripple Carry Adder

- code075.v をシミュレーションして，その表示を確認すること．
- 4-bit Adderの回路とその記述の例を示す．この構成の加算器は順次桁上げ加算器 (Ripple Carry Adder) と呼ばれる．



code075.v

```
module m_top ();
  reg  [3:0] r_a, r_b;
  wire [3:0] w_s;
  initial begin
    #10 r_a <= 3; r_b <= 4;
    #10 r_a <= 1; r_b <= 9;
    #10 r_a <= 8; r_b <= 9;
  end
  always@(*) #1 $write("%2d %2d -> %2d¥n", r_a, r_b, w_s);
  m_4ADDER m_4ADDER0 (r_a, r_b, w_s);
endmodule

module m_4ADDER (w_a, w_b, w_s);
  input  wire [3:0] w_a, w_b;
  output wire [3:0] w_s;
  wire        [4:0] w_cin;
  assign w_cin[0] = 0;
  m_FA FA0(w_a[0], w_b[0], w_cin[0], w_s[0], w_cin[1]);
  m_FA FA1(w_a[1], w_b[1], w_cin[1], w_s[1], w_cin[2]);
  m_FA FA2(w_a[2], w_b[2], w_cin[2], w_s[2], w_cin[3]);
  m_FA FA3(w_a[3], w_b[3], w_cin[3], w_s[3], w_cin[4]);
endmodule

module m_FA (w_a, w_b, w_cin, w_s, w_cout);
  input wire w_a, w_b, w_cin;
  output wire w_s, w_cout;
  wire [1:0] w_sum = w_a + w_b + w_cin;
  assign w_s = w_sum[0];
  assign w_cout = w_sum[1];
endmodule
```

```
 3  4 ->  7
 1  9 -> 10
 8  9 ->  1
```

code076.v 32-bit Ripple Carry Adder

- code076.v をシミュレーションして，その表示を確認すること．
- 32-bit Adderの記述の例を示す．
  - generate を使うことで，ループによる複数モジュールのインスタンス化や接続ができる．
  - for の最後の：Gen で，インスタンス名を Gen に指定する．
  - この例では，Gen[0].m_FA0, Gen[1].m_FA0, Gen[2].m_FA0, … となる．

code076.v

```
module m_top ();
  reg  [31:0] r_a, r_b;
  wire [31:0] w_s;
  initial begin
    #10 r_a <=  321; r_b <= 4444;
    #10 r_a <= 1024; r_b <= 2048;
  end
  always@(*) #1 $write("%4d %4d -> %4d¥n", r_a, r_b, w_s);
  m_ADDER m_ADDER0 (r_a, r_b, w_s);
endmodule

module m_ADDER (w_a, w_b, w_s);
  input  wire [31:0] w_a, w_b;
  output wire [31:0] w_s;
  wire [32:0] w_cin;
  assign w_cin[0] = 0;
  generate genvar g;
    for (g = 0; g < 32; g = g + 1) begin : Gen
      m_FA m_FA0(w_a[g], w_b[g], w_cin[g], w_s[g], w_cin[g+1]);
    end
  endgenerate
endmodule
```

```
 321 4444 -> 4765
1024 2048 -> 3072
```

code077.v n-bit Ripple Carry Adder

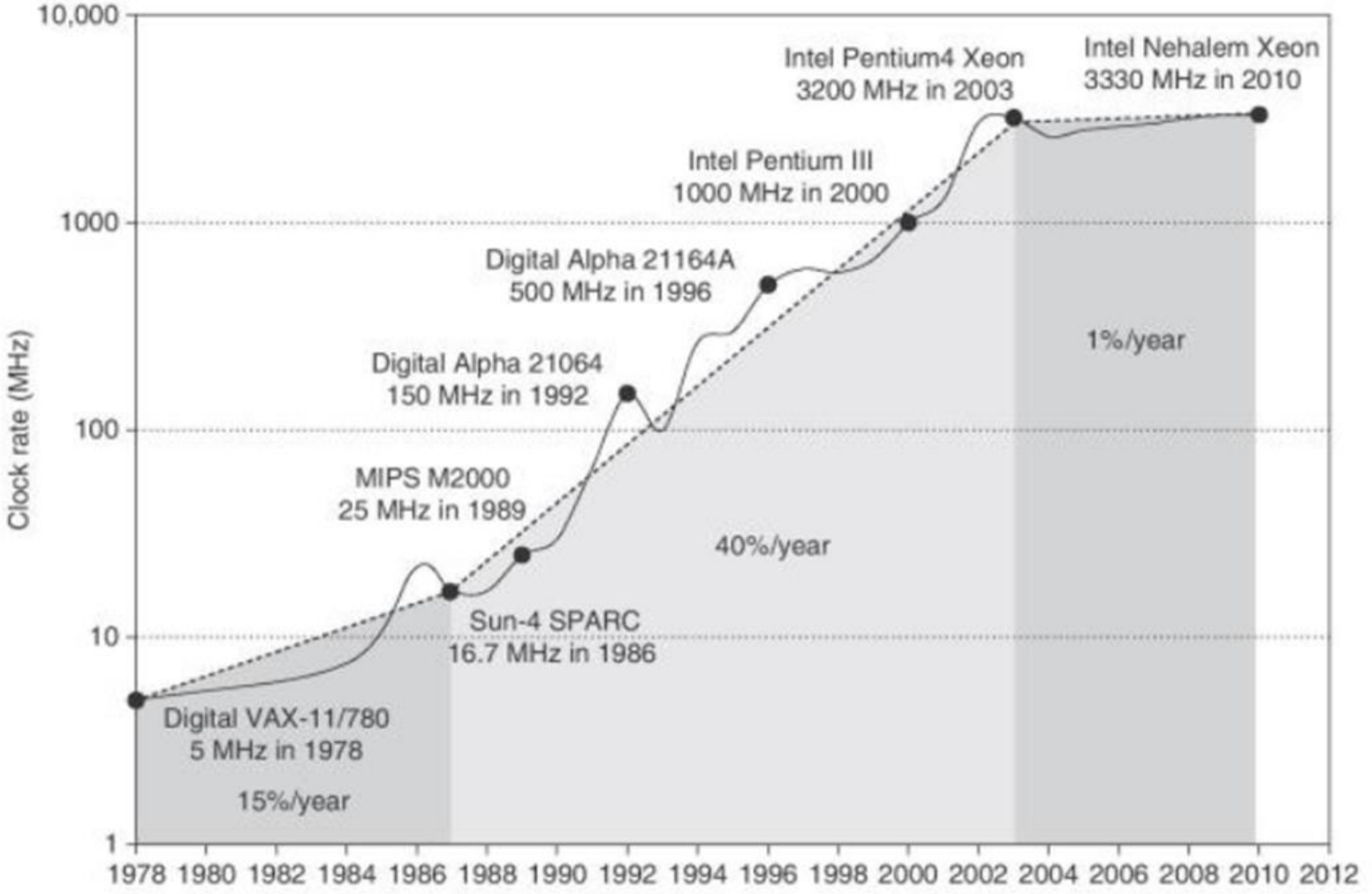
- code077.v をシミュレーションして，その表示を確認すること．
- n-bit Adderの記述の例を示す．
  - define を用いた記述の例．D_N の値を変更するだけで，加算器のビット幅を変更できる．
  - この演習では，defineにより定義される定数の名前は D_ から始まるものとする．．

code077.v

```
`define D_N 5

module m_top ();
  reg  [`D_N-1:0] r_a, r_b;
  wire [`D_N-1:0] w_s;
  initial begin
    #10 r_a <=  321; r_b <= 4444;
    #10 r_a <= 1024; r_b <= 2048;
  end
  always@(*) #1 $write("%4d %4d -> %4d¥n", r_a, r_b, w_s);
  m_ADDER m_ADDER0 (r_a, r_b, w_s);
endmodule

module m_ADDER (w_a, w_b, w_s);
  input  wire [`D_N-1:0] w_a, w_b;
  output wire [`D_N-1:0] w_s;
  wire [`D_N:0] w_cin;
  assign w_cin[0] = 0;
  generate genvar g;
    for (g = 0; g < `D_N; g = g + 1) begin : Gen
      m_FA m_FA0(w_a[g], w_b[g], w_cin[g], w_s[g], w_cin[g+1]);
    end
  endgenerate
endmodule
```

```
   1   28 ->   29
   0    0 ->    0
```

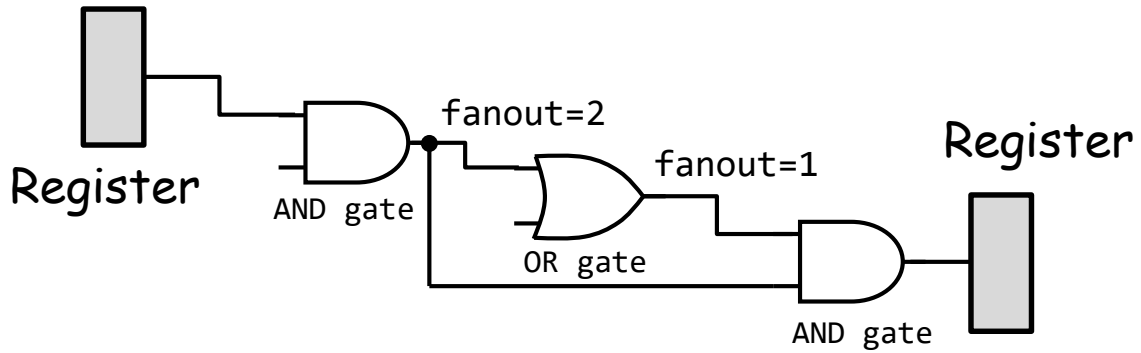
Growth in clock rate of microprocessors



From CAQA 5th edition

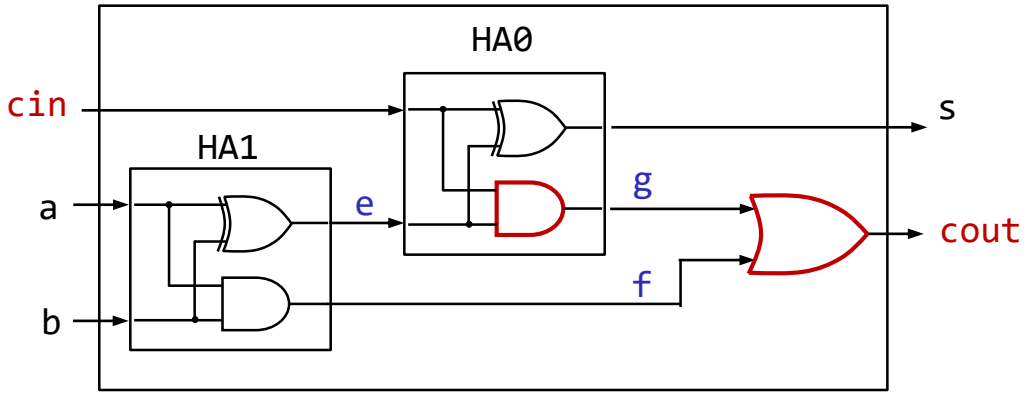
Clock rate is mainly determined by

- Switching speed of gates (transistors)
- The number of levels of gates
  - The maximum number of gates cascaded in series in any combinational logics.
  - In this example, the number of levels of gates is 3.
- Wiring delay and fanout
- The slowest of all paths is called the critical path

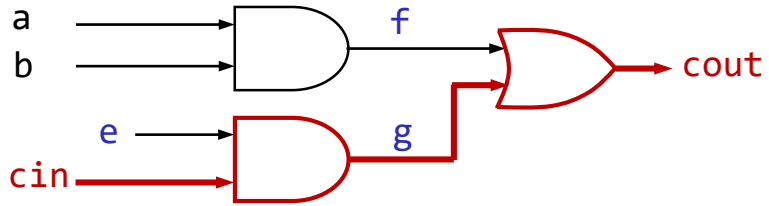


code078.v n-bit Ripple Carry Adder のクリティカルパス

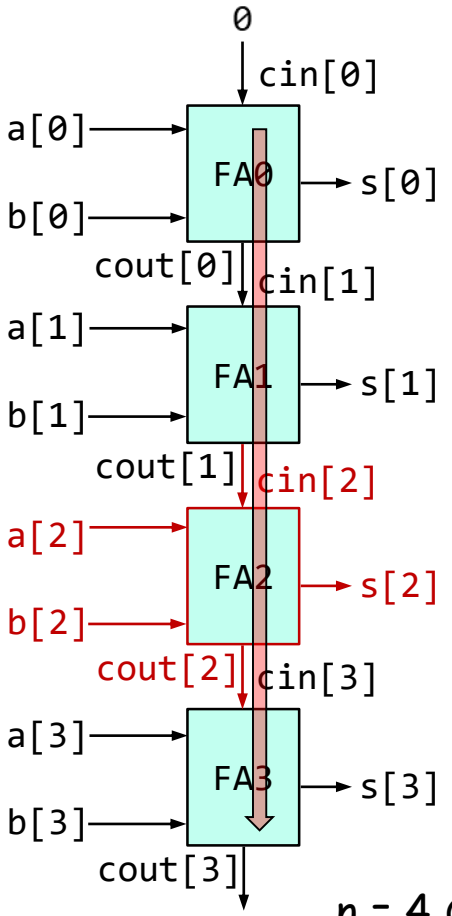
- The carry out signal (w_cout) from the carry in signal (w_cin) takes two gate delays per bit.



FA(Full Adder)

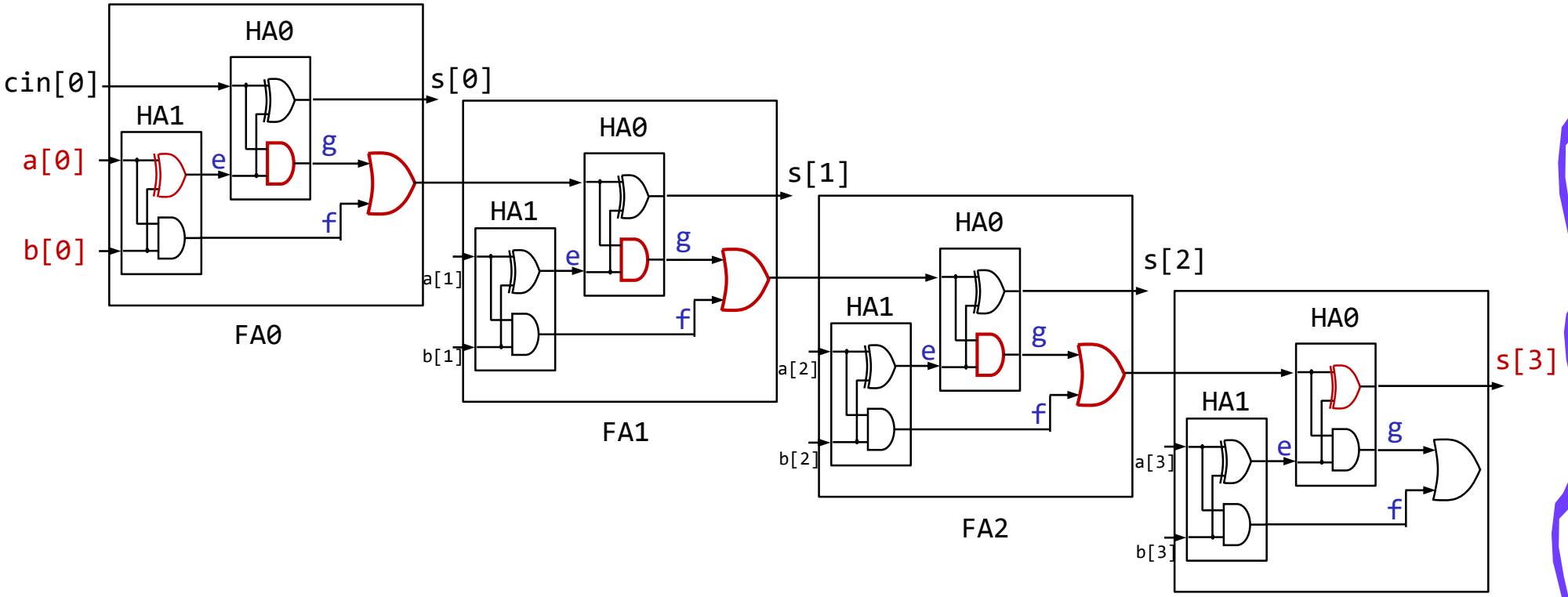


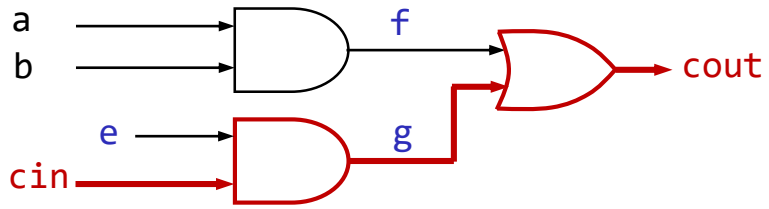
two gate delays



n = 4 の構成

code078.v 4-bit Ripple Carry Adder のクリティカルパス





two gate delays per bit

加算器のクリティカルパスの遅延を計測する

- code078.v の m_FA の青色の部分を，code074.v と同様に変更すること．

code078.v

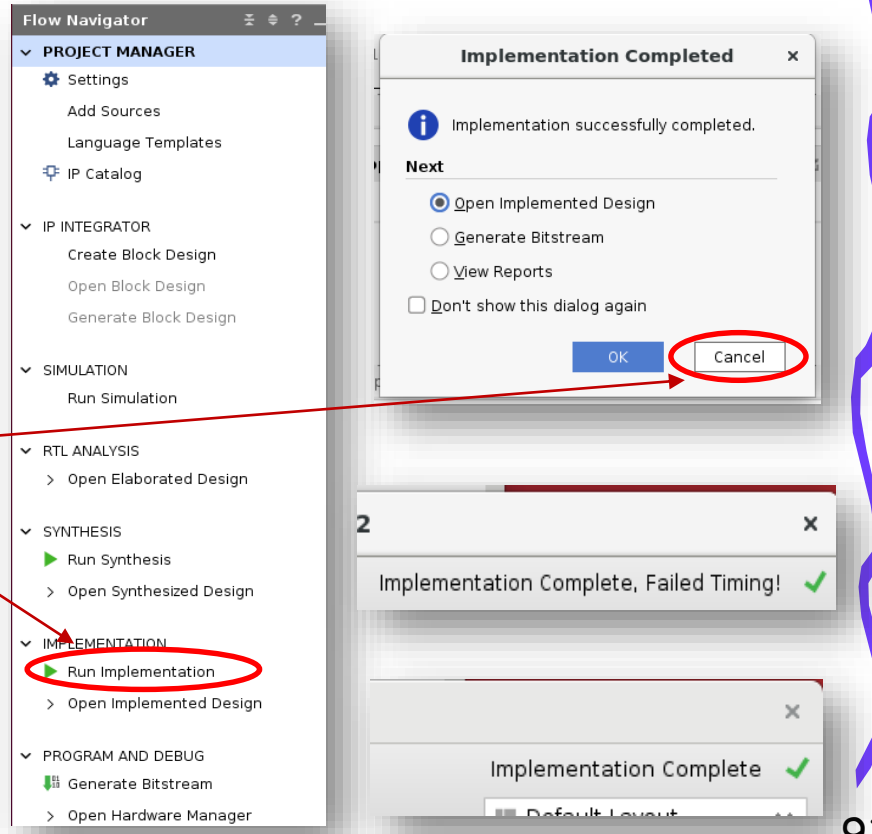
```
`define D_N 32

module m_main (w_clk, w_a, w_b, w_dout);
  input  wire w_clk, w_a, w_b;
  output wire w_dout;
  reg [`D_N-1:0] r_a=0, r_b=0, r_s=0;
  wire [`D_N-1:0] w_s;
  assign w_dout = ^r_s;
  always@(posedge w_clk) begin
    r_a <= {w_a, r_a[`D_N-1:1]};
    r_b <= {w_b, r_b[`D_N-1:1]};
    r_s <= w_s;
  end
  m_ADDER m_ADDER0 (r_a, r_b, w_s);
endmodule

module m_ADDER (w_a, w_b, w_s);
  input  wire [`D_N-1:0] w_a, w_b;
  output wire [`D_N-1:0] w_s;
  wire [`D_N:0] w_cin;
  assign w_cin[0] = 0;
  generate genvar g;
    for (g = 0; g < `D_N; g = g + 1) begin : Gen
      m_FA m_FA0(w_a[g], w_b[g], w_cin[g], w_s[g], w_cin[g+1]);
    end
  endgenerate
endmodule

module m_FA (w_a, w_b, w_cin, w_s, w_cout);
   /* Please describe here by yourself */
endmodule

module m_HA (w_a, w_b, w_s, w_c);
  input  wire w_a, w_b;
  output wire w_s, w_c;
  assign w_c = w_a & w_b;
  assign w_s = w_a ^ w_b;
endmodule
```

加算器のクリティカルパスの遅延を計測する

- 演習で，code078.v を修正して，100MHzの動作周波数の制約を満たす n-bit Adder の最大の n を求めること．ただし，n は5の倍数とする．
  - code078.v を用いて合成する(Run Implementation) ． Bitstreamは生成する必要はない．
  - 1行目の D_N の値を変化させて合成．Failed Timing! と出力された時は制約を満たしていない．
  - 1行目の D_N の値を小さくして合成．Implementation Complete が出力された時は満たしている．

code078.v

```
`define D_N 32

module m_main (w_clk, w_a, w_b, w_dout);
  input  wire w_clk, w_a, w_b;
  output wire w_dout;
  reg [`D_N-1:0] r_a=0, r_b=0, r_s=0;
  wire [`D_N-1:0] w_s;
  assign w_dout = ^r_s;
  always@(posedge w_clk) begin
    r_a <= {w_a, r_a[`D_N-1:1]};
    r_b <= {w_b, r_b[`D_N-1:1]};
    r_s <= w_s;
  end
  m_ADDER m_ADDER0 (r_a, r_b, w_s);
endmodule

module m_ADDER (w_a, w_b, w_s);
  input  wire [`D_N-1:0] w_a, w_b;
  output wire [`D_N-1:0] w_s;
  wire [`D_N:0] w_cin;
  assign w_cin[0] = 0;
  generate genvar g;
    for (g = 0; g < `D_N; g = g + 1) begin : Gen
      m_FA m_FA0(w_a[g], w_b[g], w_cin[g], w_s[g], w_cin[g+1]);
    end
  endgenerate
endmodule

以降は省略
```



Worst Negative Slack (WNS) & Critical Path

- From Vivado menu, select Open Implemented Design
- Design Timing Summary ウィンドウが表示される.
- WNS が正の値であれば，生成された回路は制約を満たしている．また，回路にはその値だけの余裕(slack)があることを示す.
  - 左図の D_N = 32 の例では，クロック周波数が 100MHz で 10 ns の制約に対して WNS は 1.796 ns となっており，これだけの余裕があることを示す．つまり制約を満たしている．この回路のクリティカルパスの遅延は 10 – 1.796 = 8.204 ns となる.
  - 右図の D_N = 80 例では，WNS は -3.527 であり，制約を満たしていない．この回路のクリティカルパスの遅延は 10 + 3.527 = 13.527 ns となる.

Design Timing Summary

Setup		Hold	
Worst Negative Slack (WNS):	1.796 ns	Worst Hold Slack (WHS):	0.166 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	94	Total Number of Endpoints:	94

All user specified timing constraints are met.

D_N 32 とした時のRipple Carry Adderの合成結果

Design Timing Summary

Setup		Hold	
Worst Negative Slack (WNS):	-3.527 ns	Worst Hold Slack (WHS):	0.136 ns
Total Negative Slack (TNS):	-60.012 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	29	Number of Failing Endpoints:	0
Total Number of Endpoints:	238	Total Number of Endpoints:	238

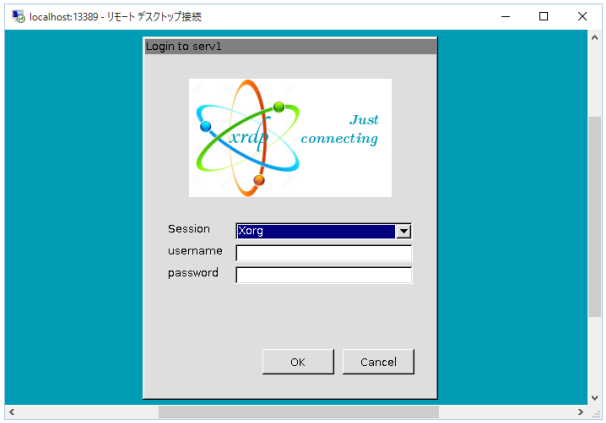
Timing constraints are not met.

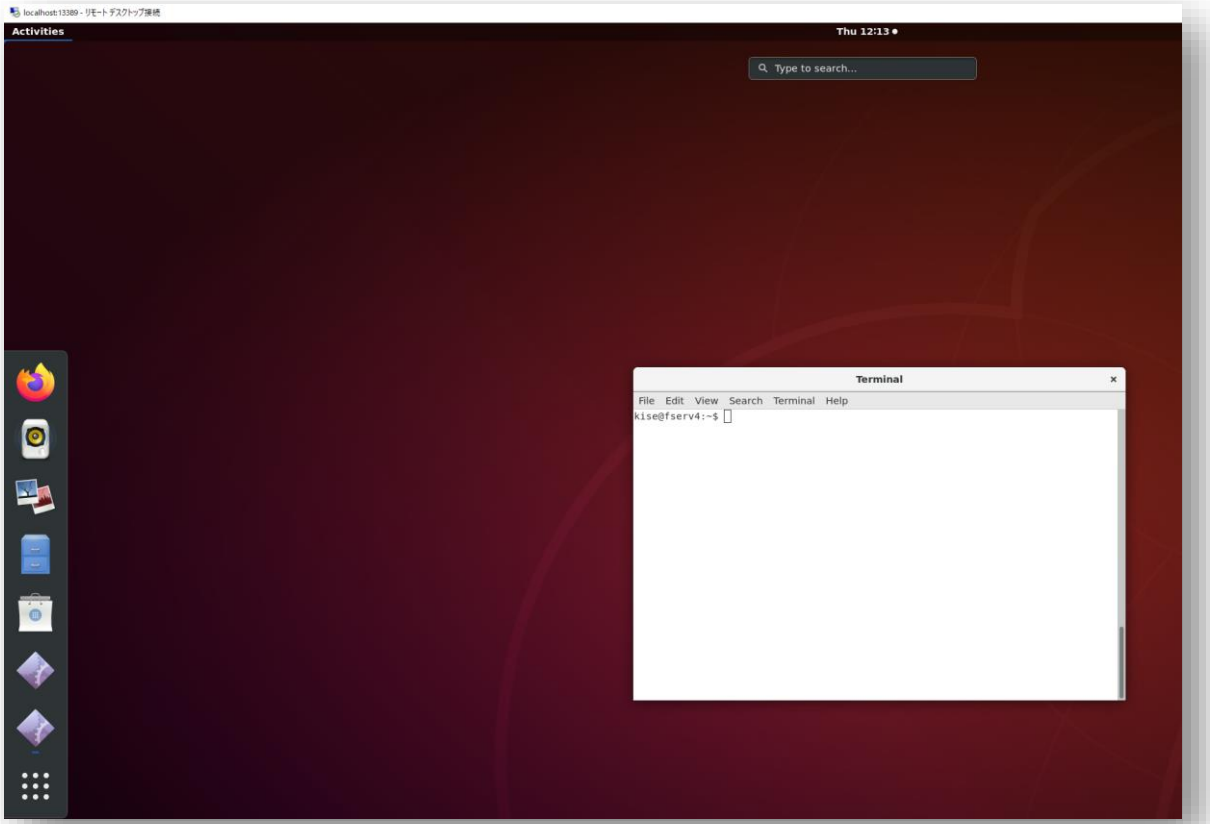
D_N 80とした時のRipple Carry Adderの合成結果

Vivado 2022.2 を利用

ACRiルームのデモンストレーション

- Vivado での VIO の使い方.
- Vivado での HW manager, オープンしたデザイン, ソースの切り替え方法.
- WNS





Department of Computer Science
Course number: CSC.T341

2023年度の講義と演習は，学術国際情報センター3階 情報工学系計算機室で実施します．

コンピュータ論理設計
Computer Logic Design

4. ハードウェア記述言語：順序回路
Hardware Description Language: Sequential Circuit

吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
講義：月曜日 10:45-12:25, 木曜日 10:45-12:25

Sample Verilog HDL code

- ACRi Room のサーバーにリモートデスクトップでログインする.
  - /home/tu_kise/cld/lec4/ にサンプルのコードがあるので，Ubuntu のターミナルで次のコマンドを入力して，自分のディレクトリにコピーする.
  - /home/tu_kise は automount のディレクトリなので，アクセスしないとファイルが見えない．tabキーによる補完がうまく動作しないことがあるので注意する.

```
$ cd
$ mkdir cld
$ cd cld
$ cp /home/tu_kise/cld/lec4/* .
```

- code051.v をシミュレーションするためには.
  - リモートデスクトップで接続した Ubuntu のターミナルで次のコマンドを入力する.
  - コマンド iverilog でコンパイルして，生成される a.out を実行する.

```
$ iverilog code051.v
$ ./a.out
```

code051.v 波形ビューワ GTKwave を使って波形を見る

```
$ iverilog code051.v
$ ./a.out
$ gtkwave main.vcd &
```

- code051.v をシミュレーションして，その表示を確認すること．
- シミュレーションのためのクロック信号の記述を示す．
- forever文は，続くブロックの処理を無限に繰り返す．この例では，reg型の信号r_clkを開始時に0に初期化し，#50の後にr_clkの値の反転を繰り返す．
- GTKwaveで波形を確認すると，10MHz（周期 100ns）のクロックが生成されることがわかる．

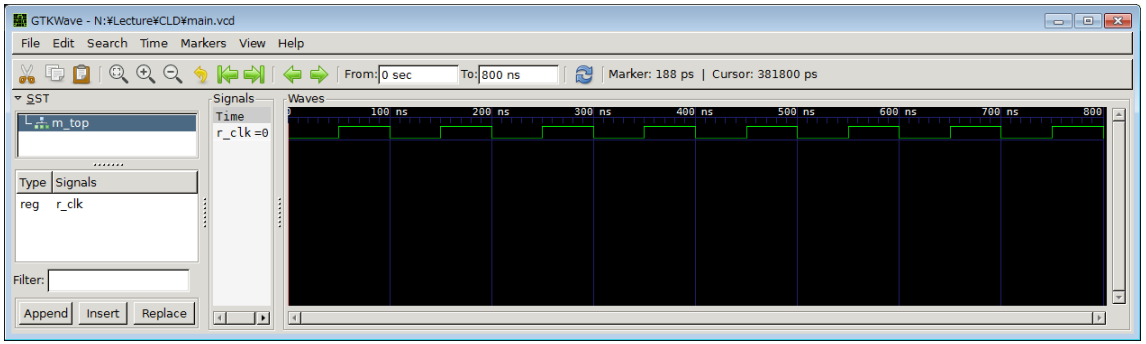
code051.v

```
module m_top ();
  reg r_clk=0;
  initial forever #50 r_clk = ~r_clk;

  always@(*) $write("%3d %d¥n", $time, r_clk);
  initial #800 $finish;

  initial $dumpfile("main.vcd"); /* file name for GTKWave */
  initial $dumpvars(0, m_top);   /* module    for GTKWave */
endmodule
```

Simulation output

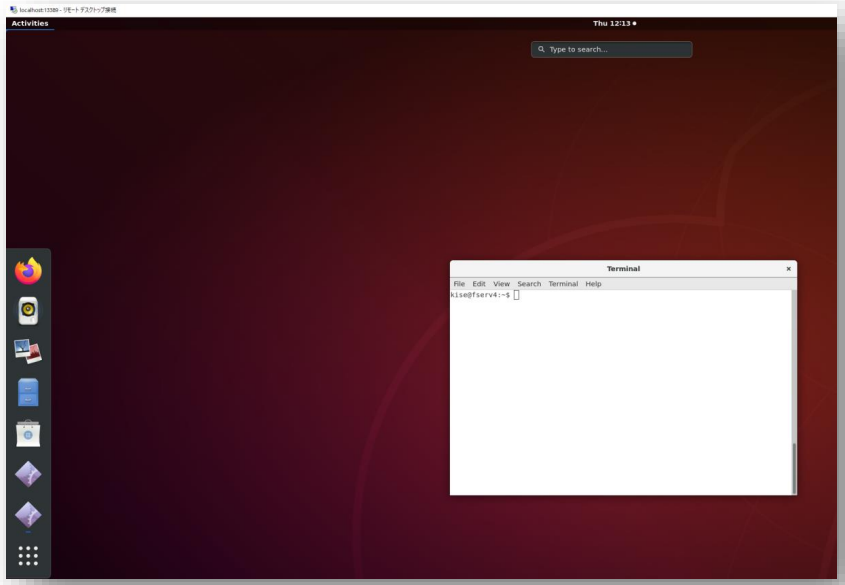
```
  0 0
 50 1
100 0
150 1
200 0
250 1
300 0
350 1
400 0
450 1
500 0
550 1
600 0
650 1
700 0
750 1
800 0
```

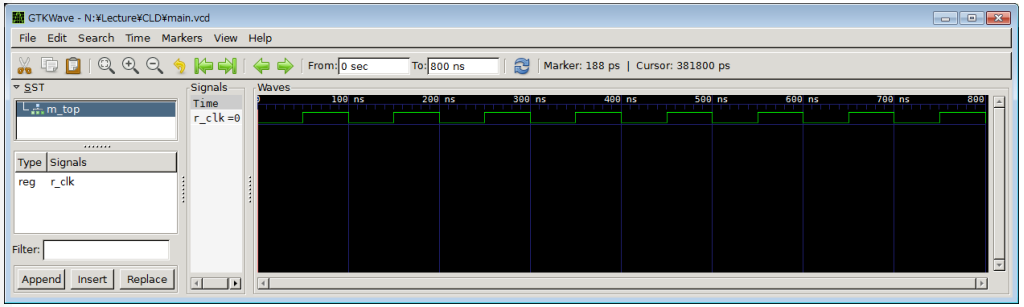
Waveform of GTKwave



GTKwave 波形ビューワのデモンストレーション

- Vivado での GTKwave の使い方.
- ファイルの読み込み, 表示する信号の追加.
- 表示範囲の調整, 2進法, 10進法での表示.





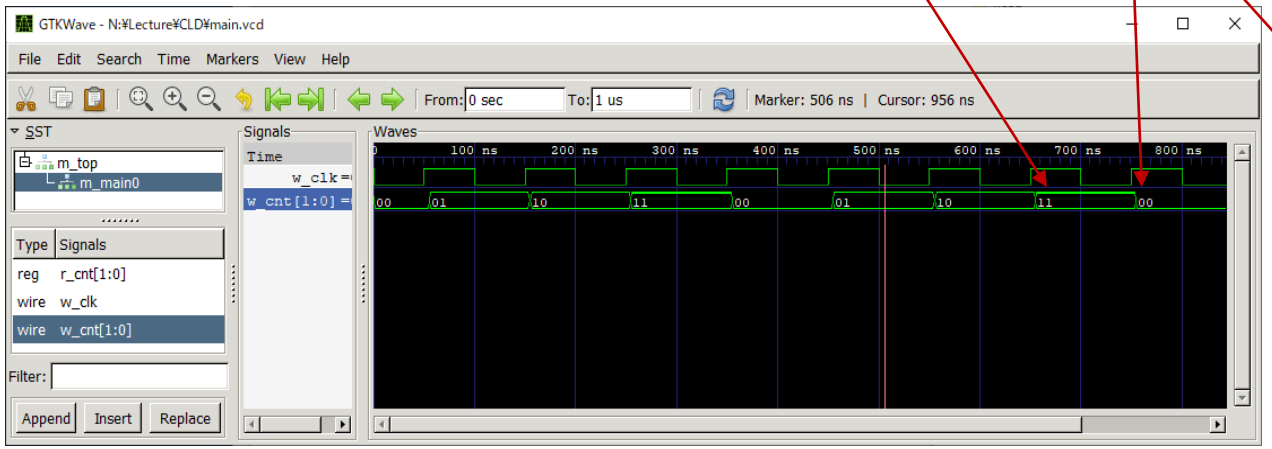
code052.v カウンタ回路を記述して波形を見る

- code052.v をシミュレーションして，その表示を確認すること．
- 単純な2ビットカウンタの記述例を示す．
- クロック信号 w_clk の立ち上がりの時 (posedge w_clk) に，1インクリメントする．ただし，r_cnt の最初の値を 0 とする。
- r_cnt の更新はクロック信号の立ち上がりから #5 だけ経過してから．
- 2ビットカウンタなので，最大値の 3 (2'b11) の次に 0 (2'b00) となる．
- iverilogでシミュレーションし，GTKWaveで波形を確認する．

```
$ iverilog code052.v
$ ./a.out
$ gtakwave a.out &
```

code052.v

```
module m_top ();
  reg r_clk=0;
  initial forever #50 r_clk = ~r_clk;
  wire [1:0] w_cnt;
  m_main m_main0 (r_clk, w_cnt);
  initial $dumpfile("main.vcd");
  initial $dumpvars(0, m_main0);
  initial #1000 $finish;
endmodule


module m_main (w_clk, w_cnt);
  input  wire w_clk;
  output wire [1:0] w_cnt;

  reg [1:0] r_cnt = 0;
  always@(posedge w_clk) begin
      r_cnt <= #5 r_cnt + 1;
  end
  assign w_cnt = r_cnt;
endmodule
```



Waveform

code053.v regの出力はregに接続されたwire

- code053.v をシミュレーションして，その表示を確認すること．
- 単純な2ビットカウンタの別の記述例を示す．
- クロック信号 w_clk の立ち上がりの時 (posedge w_clk) に，1インクリメントする．ただし，r_cnt の最初の値を 0 とする。
- モジュールの出力ポートでは wire と reg を用いることができる．
- `timescale 1ns/100ps という記述の 1ns は #1 が実時間で 1ns であることを示し，次の 100ps でシミュレーションの精度が 100ps 単位であることを指定する．
  - timescale の行はこのまま記述しておいた方が良い．

code053.v

```
`timescale 1ns/100ps
`default_nettype none

module m_top ();
  reg r_clk=0;
  initial forever #50 r_clk = ~r_clk;
  wire [1:0] w_cnt;
  m_main m_main0 (r_clk, w_cnt);
  initial $dumpfile("main.vcd");
  initial $dumpvars(0, m_main0);
  initial #1000 $finish;
endmodule


module m_main (w_clk, r_cnt);
  input  wire w_clk;
  output reg [1:0] r_cnt;

  initial r_cnt = 0;
  always@(posedge w_clk) r_cnt <= #5 r_cnt + 1;
endmodule
```

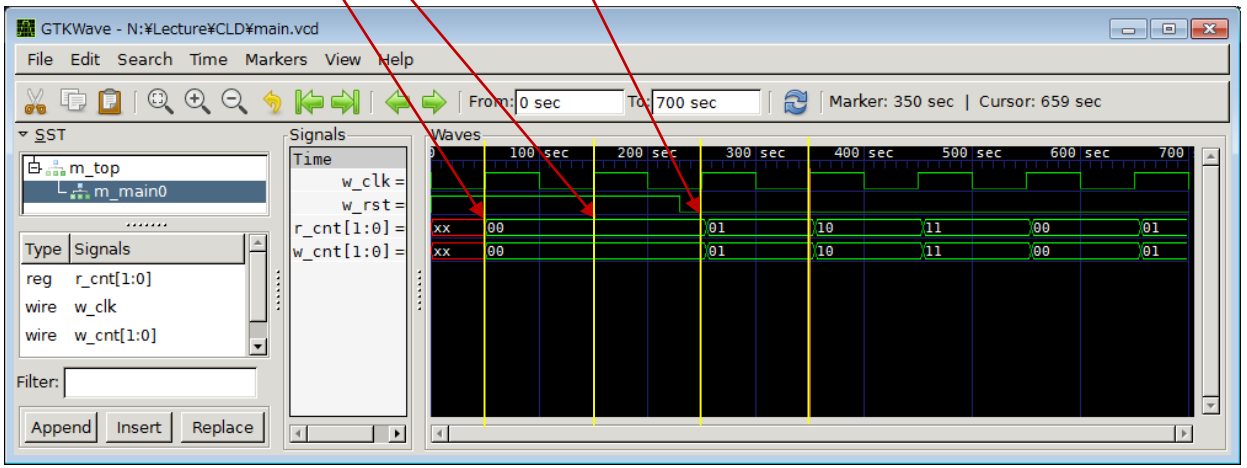
code052.v

```
module m_main (w_clk, w_cnt);
  input  wire w_clk;
  output wire [1:0] w_cnt;

  reg [1:0] r_cnt = 0;
  always@(posedge w_clk) begin
      r_cnt <= #5 r_cnt + 1;
  end
  assign w_cnt = r_cnt;
endmodule
```

code054.v 同期リセット付きのカウンタ回路

- code054.v をシミュレーションして，その表示を確認すること．
- 同期リセット付き2ビットカウンタの記述例を示す．
- クロック信号 w_clkの立ち上がりの時に，リセット信号 r_rst が1でカウンタの値はゼロで初期化され，
  そうでなければ1インクリメントされる．
- iverilogでシミュレーションし，GTKWaveで波形を確認する．

Waveform



code054.v

```
module m_top ();
  reg r_clk=0;
  initial forever #50 r_clk = ~r_clk;
  reg r_rst=1;
  initial #230 r_rst=0;
  wire [1:0] w_cnt;
  m_main m_main0 (r_clk, r_rst, w_cnt);
  initial $dumpfile("main.vcd");
  initial $dumpvars(0, m_main0);
  initial #1000 $finish;
endmodule

module m_main (w_clk, w_rst, w_cnt);
  input  wire w_clk, w_rst;
  output wire [1:0] w_cnt;

  reg [1:0] r_cnt;
  always@(posedge w_clk) begin
    if (w_rst) r_cnt <= 0;
    else r_cnt <= #5 r_cnt + 1;
  end
  assign w_cnt = r_cnt;
endmodule
```

同期リセット，非同期リセット，リセット無し

- 同期リセットの記述（左）では，クロック信号に同期してリセットされる．
- 非同期リセットの記述（中央）では，クロック信号に同期せずに，リセット信号の立ち上がりのタイミングでリセットされる．
- 右の記述はリセット信号を使わない記述．
- 必要がなければ，リセットを使わない記述（右）が良い．
  リセットが必要であれば，同期リセット（左）を使う記述が良い．

code054.v

```
module m_main (w_clk, w_rst, w_cnt);
  input  wire w_clk, w_rst;
  output wire [1:0] w_cnt;

  reg [1:0] r_cnt;
  always@(posedge w_clk) begin
    if (w_rst) r_cnt <= 0;
    else r_cnt <= #5 r_cnt + 1;
  end
  assign w_cnt = r_cnt;
endmodule
```

同期リセットの例

```
module m_main (w_clk, w_rst, w_cnt);
  input  wire w_clk, w_rst;
  output wire [1:0] w_cnt;

  reg [1:0] r_cnt;
  always@(posedge w_clk or posedge w_rst) begin
    if (w_rst) r_cnt <= 0;
    else r_cnt <= #5 r_cnt + 1;
  end
  assign w_cnt = r_cnt;
endmodule
```

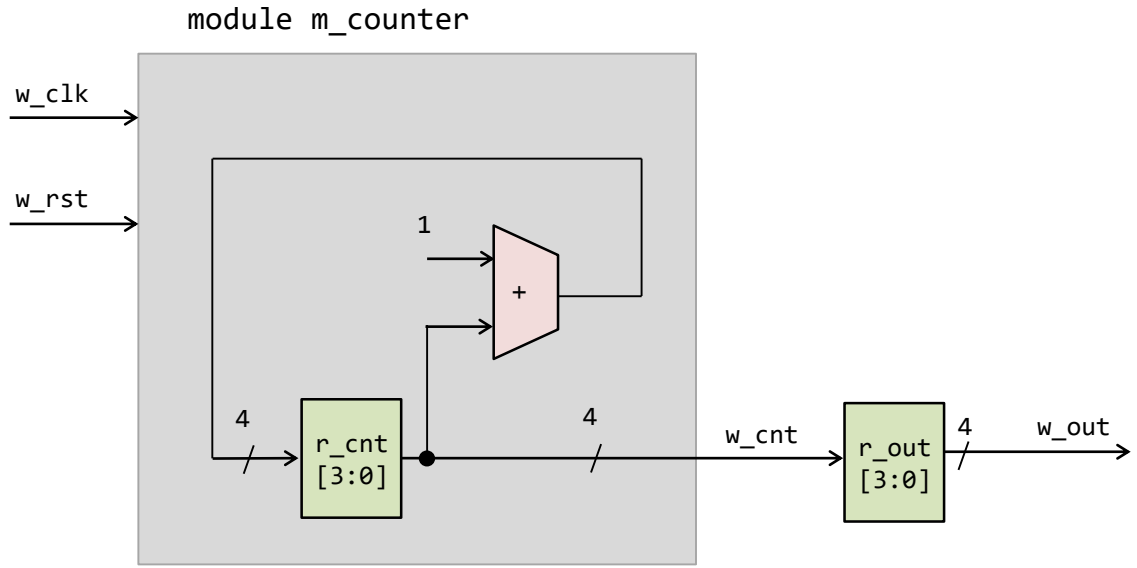
非同期リセットの例

```
module m_main (w_clk, w_cnt);
  input  wire w_clk;
  output wire [1:0] w_cnt;

  reg [1:0] r_cnt = 0;
  always@(posedge w_clk) begin
     r_cnt <= #5 r_cnt + 1;
  end
  assign w_cnt = r_cnt;
endmodule
```

リセットを使わない例

Sequential Circuit（順序回路）とレジスタの動作

- 順序回路では，クロック信号に対するレジスタの動作を正しく理解することが重要
- レジスタは，クロック信号の立ち上がりのタイミングで入力線の値を取得(sampling)して，少し遅れて取得した値を出力線に出力する.
- レジスタの更新では，initial や always でタイミングを指定すること.
- このモジュールの波形を考えよう.



code055.v, code056.v 一定の間隔で生じる動作の記述

- code055.v の 999999 を 4 に変更してシミュレーションして，その表示を確認すること．
- 1MHz のクロック信号を入力として、1秒の間隔で 0, 1, 0, 1 と変化させるハードウェアの記述例を示す．LED を点滅させる場合などに利用する。
- 補足
  - 1MB = 1024 x 1024 B
  - 1MHz = 1000 x 1000 Hz
- code055.v の2か所のalwaysブロックをまとめると，code056.v になる．

code056.v

```
module m_main (w_clk, r_out);
  input  wire w_clk;
  output reg  r_out;

  initial r_out = 0;
  reg [31:0] r_cnt = 0;
  always@(posedge w_clk) begin
    r_cnt <= (r_cnt==999999) ? 0 : r_cnt +1;
    r_out <= (r_cnt==0) ? ~r_out : r_out;
  end
endmodule
```

code055.v

```
`timescale 1ns/100ps
module m_top ();
  reg r_clk=0;
  initial forever #50 r_clk = ~r_clk;
  wire w_out;
  m_main m_main0 (r_clk, w_out);
  initial $dumpfile("main.vcd");
  initial $dumpvars(0, m_main0);
  initial #1000 $finish;
endmodule


module m_main (w_clk, r_out);
  input  wire w_clk;
  output reg  r_out;

  reg [31:0] r_cnt = 0;
  always@(posedge w_clk) begin
    r_cnt <= (r_cnt==999999) ? 0 : r_cnt +1;
  end

  initial r_out = 0;
  always@(posedge w_clk) begin
    r_out <= (r_cnt==0) ? ~r_out : r_out;
  end
endmodule
```

code057.v 演習(1)で使ったハードウェア記述

- 100MHz のクロック信号を入力として、1秒の間隔で 0, 1, 0, 1 と変化させるハードウェアの記述例を示す. LED を点滅させる場合などに利用する.
- 今は理解できる.

code057.v

```
module m_main (w_clk, w_led);
  input  wire w_clk;
  output wire [3:0]  w_led;

  reg        r_out = 0;
  reg [31:0] r_cnt = 0;
  always@(posedge w_clk) begin
    r_cnt <= (r_cnt==99999999) ? 0 : r_cnt +1;
    r_out <= (r_cnt==0) ? ~r_out : r_out;
  end
  assign w_led = {r_out, r_out, r_out, r_out};
  // vio_0 vio_00(w_clk, w_led[3], w_led[2], w_led[1], w_led[0]);
endmodule
```

code061.v シフトレジスタ(右シフト)

- code061.v をシミュレーションして，その表示を確認すること.
- シフトレジスタは，サイクル毎に，その内容を1ビットだけ右(あるいは左)にシフトする．また，最上位ビット(あるいは最下位ビット)に入力された値を格納する.

code061.v

```
module m_top ();
  reg r_clk=0;
  initial forever #50 r_clk = ~r_clk;
  wire [7:0] w_out;
  m_main m_main0 (r_clk, w_out);
  initial $dumpfile("main.vcd");
  initial $dumpvars(0, m_main0);
  initial #1000 $finish;
  always@(*) #1 $display("%3d: %b", $time, w_out);
endmodule


module m_main (w_clk, r_sreg);
  input  wire w_clk;
  output reg [7:0] r_sreg = 8'b10101111;

  wire w_in = 0;
  always@(posedge w_clk) begin
    r_sreg <= {w_in, r_sreg[7:1]};
  end
endmodule
```

右シフト

Simulation output

```
  1: 10101111
 51: 01010111
151: 00101011
251: 00010101
351: 00001010
451: 00000101
551: 00000010
651: 00000001
751: 00000000
```

code062.v シフトレジスタ(左シフト)

- code062.v をシミュレーションして，その表示を確認すること.
- シフトレジスタは，サイクル毎に，その内容を1ビットだけ右(あるいは左)にシフトする．また，最上位ビット(あるいは最下位ビット)に入力された値を格納する.

code062.v

```
module m_top ();
  reg r_clk=0;
  initial forever #50 r_clk = ~r_clk;
  wire [7:0] w_out;
  m_main m_main0 (r_clk, w_out);
  initial $dumpfile("main.vcd");
  initial $dumpvars(0, m_main0);
  initial #1000 $finish;
  always@(*) #1 $display("%3d: %b", $time, w_out);
endmodule


module m_main (w_clk, r_sreg);
  input  wire w_clk;
  output reg [7:0] r_sreg = 8'b10101111;

  wire w_in = 0;
  always@(posedge w_clk) begin
    r_sreg <= {r_sreg[6:0], w_in};
  end
endmodule
```

左シフト

Simulation output

```
  1: 10101111
 51: 01011110
151: 10111100
251: 01111000
351: 11110000
451: 11100000
551: 11000000
651: 10000000
751: 00000000
```

code064.v EXOR（排他的論理和）ゲート

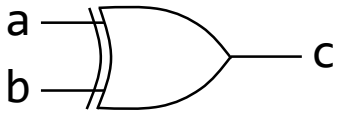
- code064.v をシミュレーションして，その表示を確認すること．
- 論理演算子には，
  単項演算子の ~ (NOT), 2項演算子として & (AND), | (OR), ^ (EXOR)がある．

code064.v

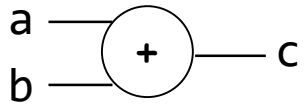
```
module m_top ();
  reg r_a, r_b;
  wire w_c;
  assign w_c = r_a ^ r_b;

  initial begin
    #10 r_a <= 0; r_b <= 0;
    #10 r_a <= 0; r_b <= 1;
    #10 r_a <= 1; r_b <= 0;
    #10 r_a <= 1; r_b <= 1;
  end
  always@(*) #1 $display("%2d: %d %d -> %d", $time, r_a, r_b, w_c);
endmodule
```

Simulation output

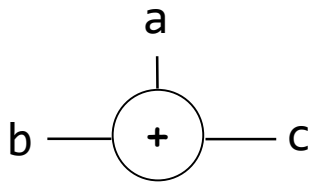
```
11: 0 0 -> 0
21: 0 1 -> 1
31: 1 0 -> 1
41: 1 1 -> 0
```



EXOR gate



1ビットの加算器



code065.v 3入力の EXOR ゲート

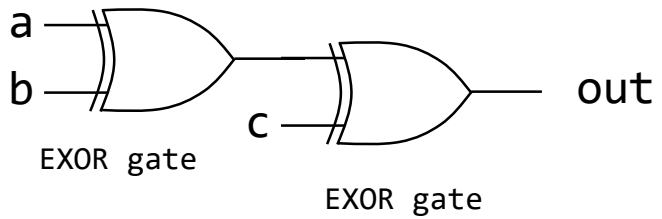
- code065.v をシミュレーションして，その表示を確認すること．
- 論理演算子には，2項演算子として & (AND), | (OR), ^ (EXOR)がある．
- 3入力以上の論理演算子を考えよう．

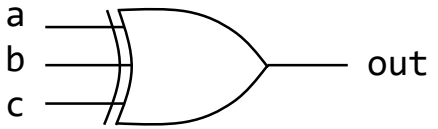
```
module m_top ();
  reg r_a, r_b, r_c;
  wire w_out;
  assign w_out = r_a ^ r_b ^ r_c;

  initial begin
    #10 r_a <= 0; r_b <= 0; r_c <= 0;
    #10 r_a <= 0; r_b <= 0; r_c <= 1;
    #10 r_a <= 0; r_b <= 1; r_c <= 0;
    #10 r_a <= 0; r_b <= 1; r_c <= 1;
    #10 r_a <= 1; r_b <= 0; r_c <= 0;
    #10 r_a <= 1; r_b <= 0; r_c <= 1;
    #10 r_a <= 1; r_b <= 1; r_c <= 0;
    #10 r_a <= 1; r_b <= 1; r_c <= 1;
  end
  always@(*) #1 $display("%2d: %d %d %d -> %d", $time, r_a, r_b, r_c, w_out);
endmodule
```

Simulation output

```
11: 0 0 0 -> 0
21: 0 0 1 -> 1
31: 0 1 0 -> 1
41: 0 1 1 -> 0
51: 1 0 0 -> 1
61: 1 0 1 -> 0
71: 1 1 0 -> 0
81: 1 1 1 -> 1
```

code065.v





code066.v 3入力の EXOR ゲートのバスを用いた記述

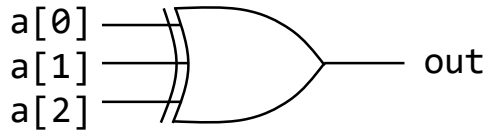
- code066.v をシミュレーションして，その表示を確認すること．

code066.v

```
module m_top ();
  reg [2:0] r_a;
  wire w_out;
  assign w_out = ^r_a;

  initial begin
    #10 r_a <= 3'b000;
    #10 r_a <= 3'b001;
    #10 r_a <= 3'b010;
    #10 r_a <= 3'b011;
    #10 r_a <= 3'b100;
    #10 r_a <= 3'b101;
    #10 r_a <= 3'b110;
    #10 r_a <= 3'b111;
  end
  always@(*) #1 $display("%2d: %b -> %d", $time, r_a, w_out);
endmodule
```

Simulation output

```
11: 000 -> 0
21: 001 -> 1
31: 010 -> 1
41: 011 -> 0
51: 100 -> 1
61: 101 -> 0
71: 110 -> 0
81: 111 -> 1
```

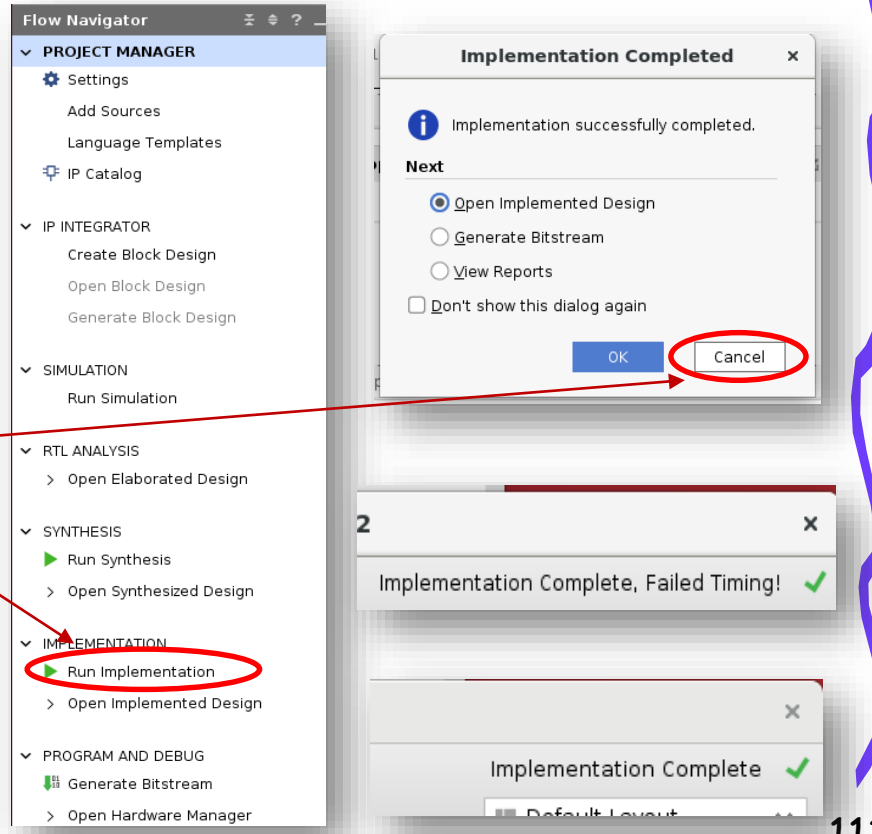


演習(2) 加算器のクリティカルパスの遅延を計測する

- code078.v を修正して，100MHzの動作周波数の制約を満たす n-bit Adder の最大の n を求めること．ただし，n は5の倍数とする．
  - code078.v を用いて合成する(Run Implementation)．Bitstreamは生成する必要はない．
  - 1行目の D_N の値を変化させて合成．Failed Timing! と出力された時は制約を満たしていない．
  - 1行目の D_N の値を小さくして合成．Implementation Complete が出力された時は満たしている．

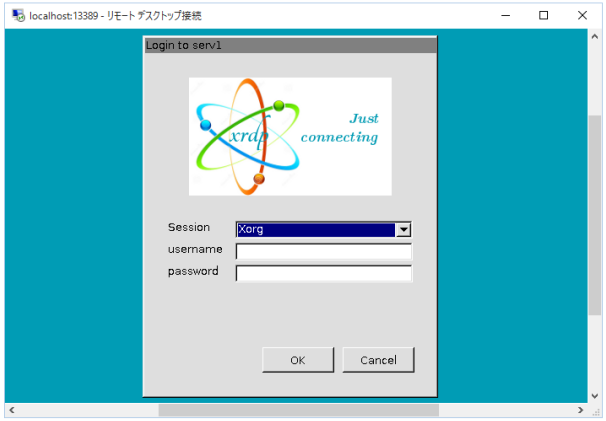
code078.v

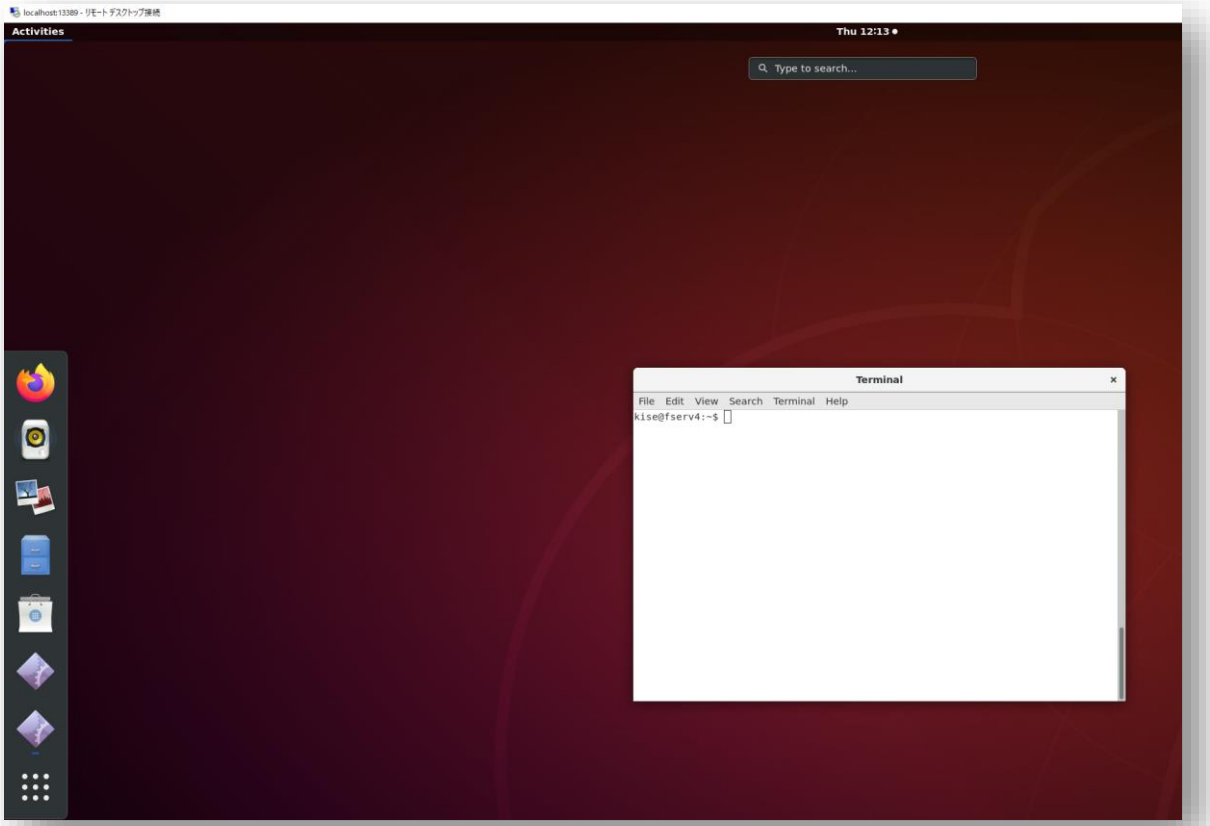
```
`define D_N 32

module m_main (w_clk, w_a, w_b, w_dout);
  input  wire w_clk, w_a, w_b;
  output wire w_dout;
  reg [`D_N-1:0] r_a=0, r_b=0, r_s=0;
  wire [`D_N-1:0] w_s;
  assign w_dout = ^r_s;
  always@(posedge w_clk) begin
    r_a <= {w_a, r_a[`D_N-1:1]};
    r_b <= {w_b, r_b[`D_N-1:1]};
    r_s <= w_s;
  end
  m_ADDER m_ADDER0 (r_a, r_b, w_s);
endmodule

module m_ADDER (w_a, w_b, w_s);
  input  wire [`D_N-1:0] w_a, w_b;
  output wire [`D_N-1:0] w_s;
  wire [`D_N:0] w_cin;
  assign w_cin[0] = 0;
  generate genvar g;
    for (g = 0; g < `D_N; g = g + 1) begin : Gen
      m_FA m_FA0(w_a[g], w_b[g], w_cin[g], w_s[g], w_cin[g+1]);
    end
  endgenerate
endmodule

以降は省略
```



ACRiルームのデモンストレーション

- Vivado での VIO の使い方.
- Vivado での HW manager, オープンしたデザイン, ソースの切り替え方法.
- WNS
- Vivado における回路図(schematic)の表示.





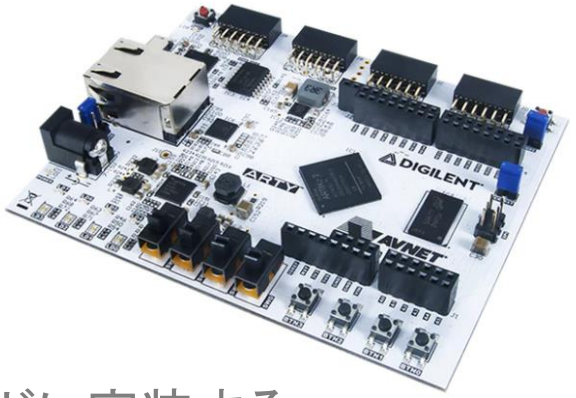
Department of Computer Science
Course number: CSC.T341

コンピュータ論理設計
Computer Logic Design

5. VLSIとリコンフィギャラブルシステム
VLSI and Reconfigurable Systems

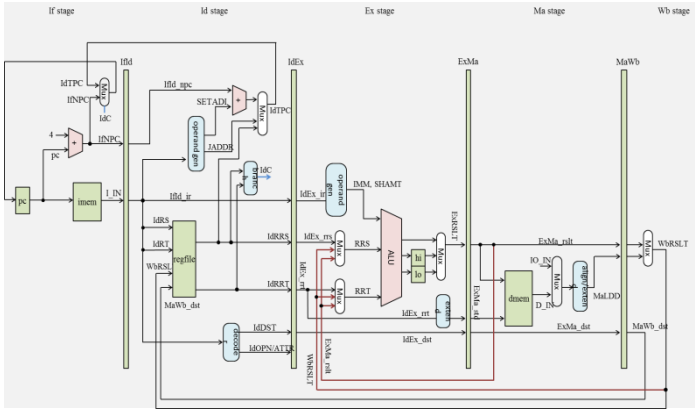
吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
講義：月曜日 10:45-12:25, 木曜日 10:45-12:25

コンピュータ論理設計の特徴



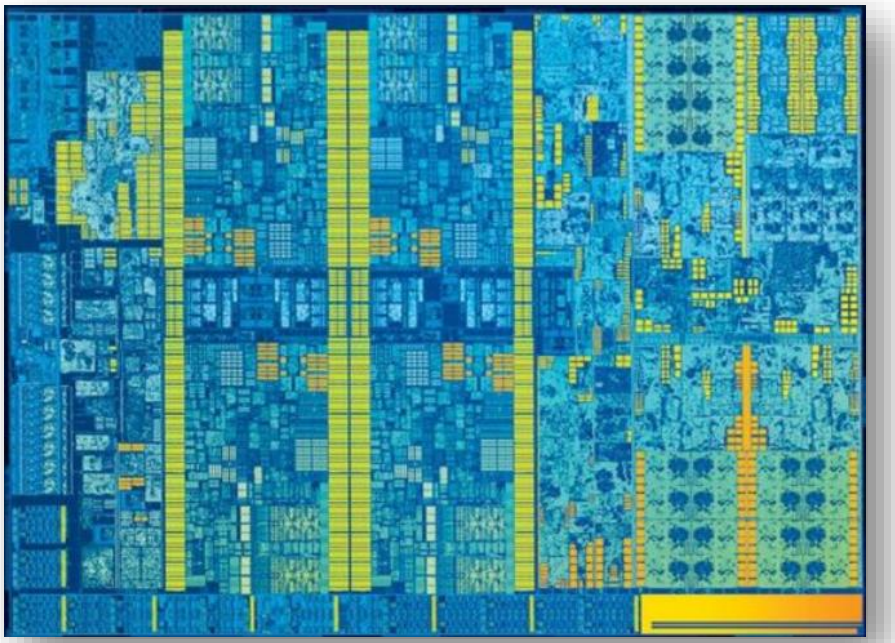
- 講義2単位，演習1単位．
- 1人1台のFPGA (Field-Programmable Gate Array) ボードを用いた演習．
- 4人程度を1グループとした共同作業と問題解決．
- 教科書で説明されるプロセッサのRISC-V版を ハードウェア記述言語Verilog HDLで記述し，FPGAボードに実装する．
- グループとしてプロセッサの高速化に取り組み，コンテスト形式で成果を競う．
- 3Q開講のコンピュータアーキテクチャ（CSC.T363）のための準備．

```
module main (clk, led);
  input  wire clk;
  output wire led;

  reg [26:0] cnt=0;
  always @(posedge clk) cnt <= cnt + 1;

  assign led = cnt[26];
endmodule
```

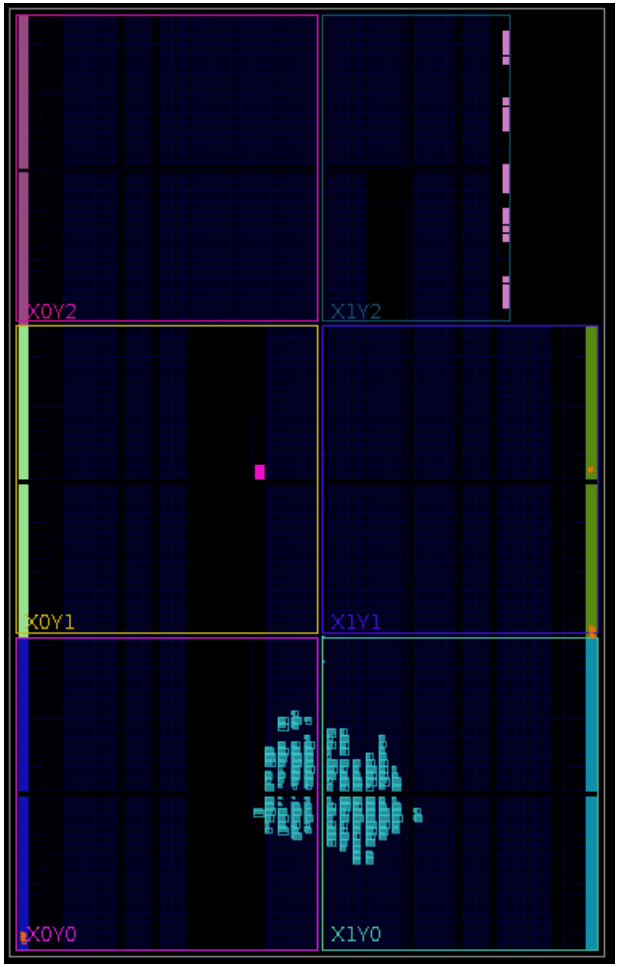


ASIC (Application Specific Integrated Circuit)

- ASIC（特定用途向け集積回路）
  - マイクロプロセッサとしてのASIC
  - FPGAとしてのASIC
- FPGAとしてのASICに実現されるマイクロプロセッサは？



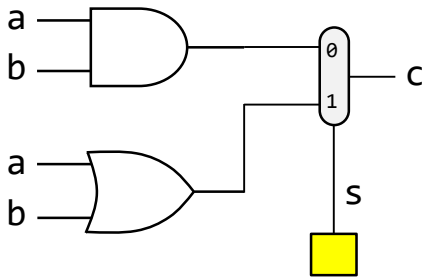
Intel Skylake microprocessor August 2015



Xilinx Artix-7 FPGA on Arty A7-35T

Reconfigurable Systems

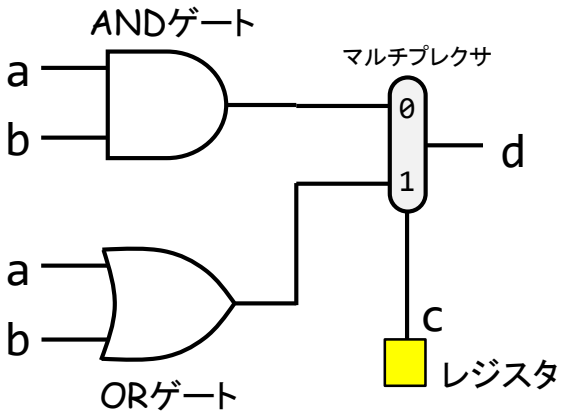
- 再構成可能(リコンフィギャラブル)システム
  - 問題の解法アルゴリズムをハードウェア化してユーザが書き換え可能なデバイス上で直接実行することにより，高い性能と柔軟性を実現するシステム
  - 書き換え可能なデバイスを活用するコンピューティングシステムをアダプティブコンピューティングと呼ぶ.
- FPGA (Field Programmable Gate Array)
  - AMD (Xilinx)
  - Intel (Altera)
  - Lattice
  - NanoBridge

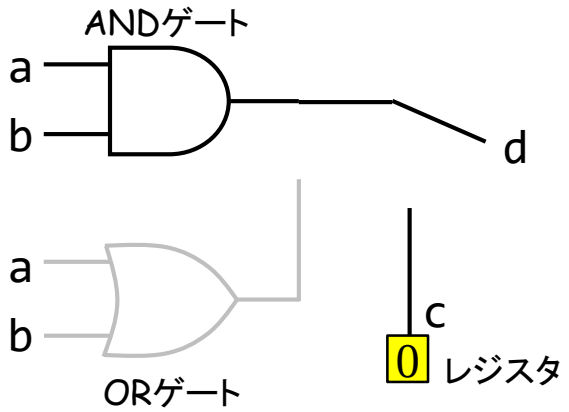


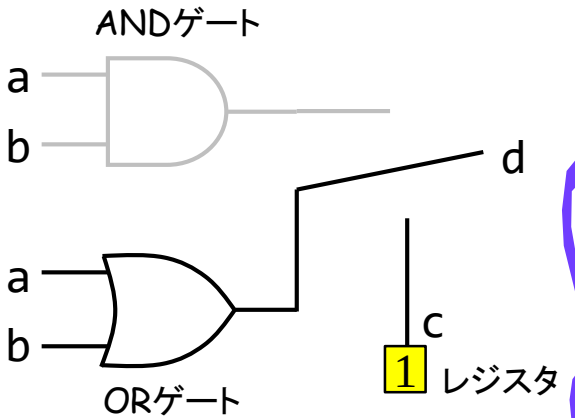
This circuit can change the role of AND or OR gate by the value stored in **s**.
Is this reconfigurable? No!

制御信号cによってANDゲートあるいはORゲートとなる回路

- ANDゲートとORゲートを切り替えることで性能が向上する例を考える.
- レジスタの値で出力を選択する回路
- この例は，再構成可能システムではない！

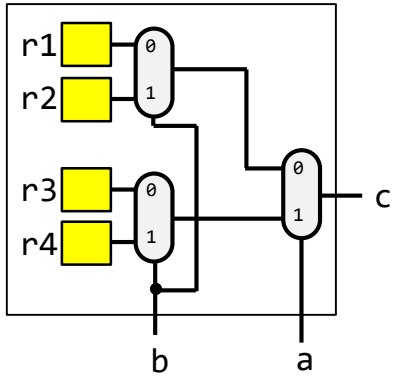




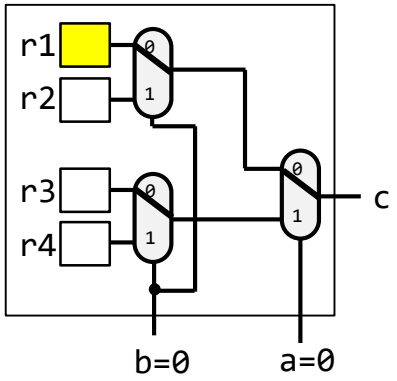


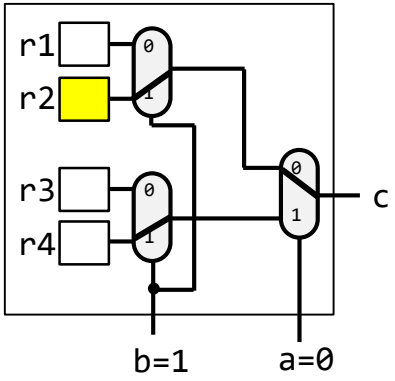
ルックアップテーブル (Lookup Table, LUT)

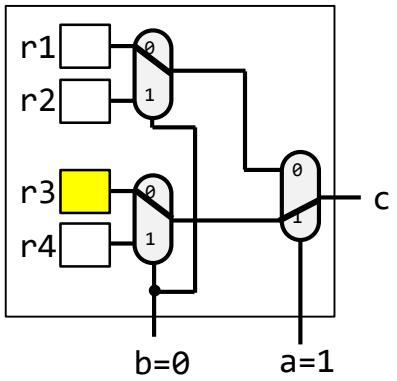
- a, b を入力として，c を出力とする LUT
- 値を保持するレジスタ（黄色）の値を選択する回路

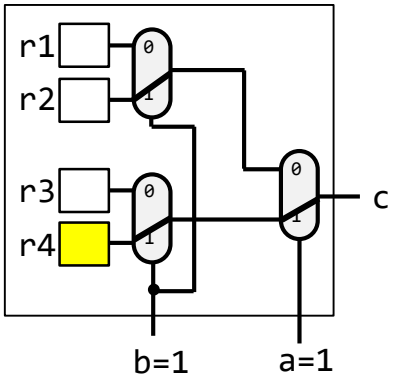


2入力のLUTの構成





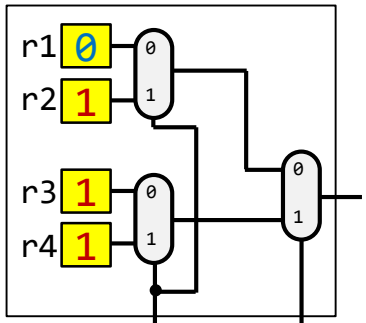




入力		出力
a	b	c
0	0	r1
0	1	r2
1	0	r3
1	1	r4

ルックアップテーブル (Lookup Table, LUT)

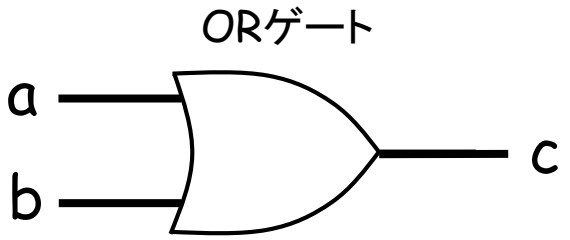
- レジスタの値を上から 0, 1, 1, 1 に設定すると、このLUTはORゲートと同じ動作をする.



2入力のLUT　b　a

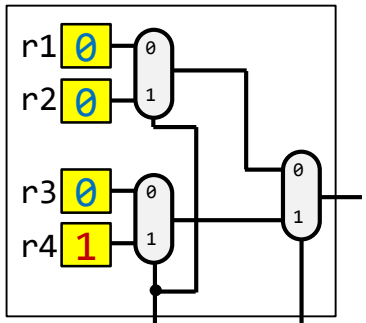
a	b	c
0	0	0
0	1	1
1	0	1
1	1	1

真理値表



左の構成のLUTと等価な回路

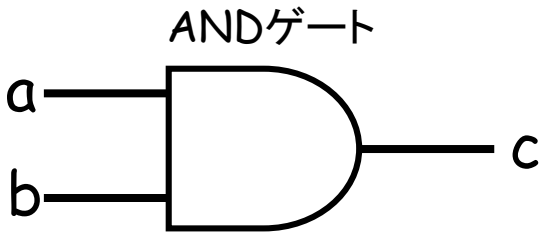
- レジスタの値を上から 0, 0, 0, 1 に設定すると、このLUTはANDゲートと同じ動作をする。



2入力のLUT　b　a

a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

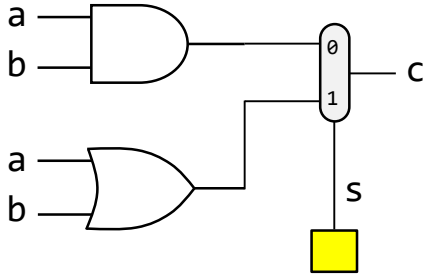
真理値表



左の構成のLUTと等価な回路

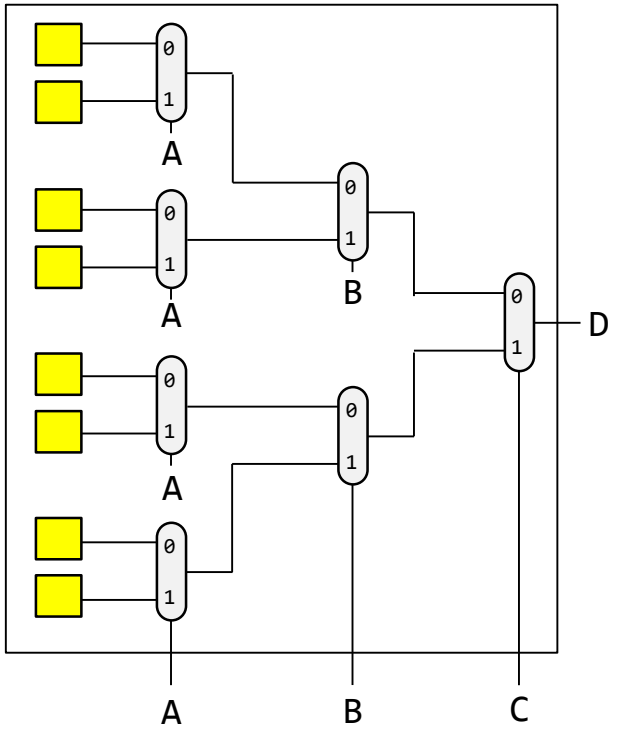
ルックアップテーブル (Lookup Table, LUT)

- Reconfigurable Logic



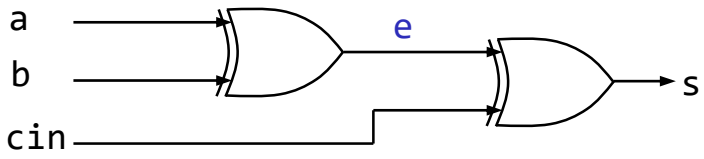
This circuit can change the role
of AND or OR gate by the value stored in **s**.
Is this reconfigurable?
No!

3-input LUT structure



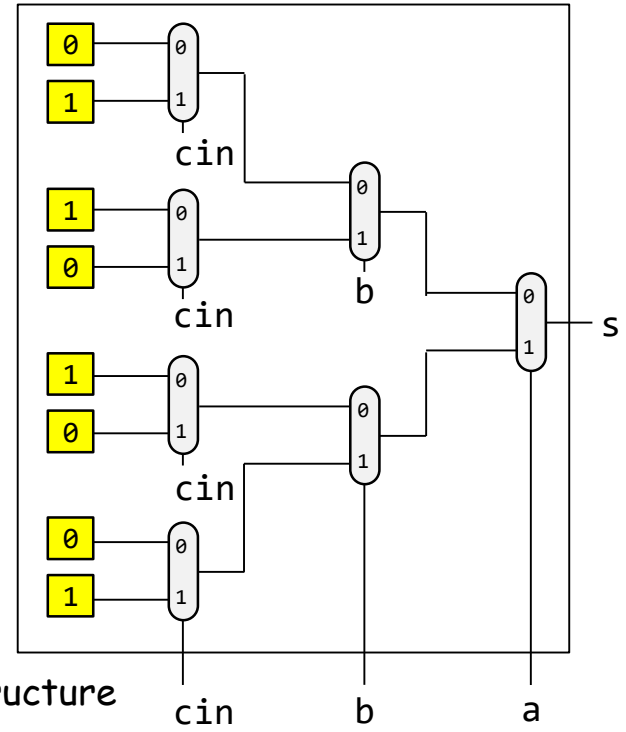
ルックアップテーブル (Lookup Table, LUT)

- Reconfigurable Logic
  - 3-input LUT has 8 configuration registers
  - 6-input LUT for Artix-7 FPGA

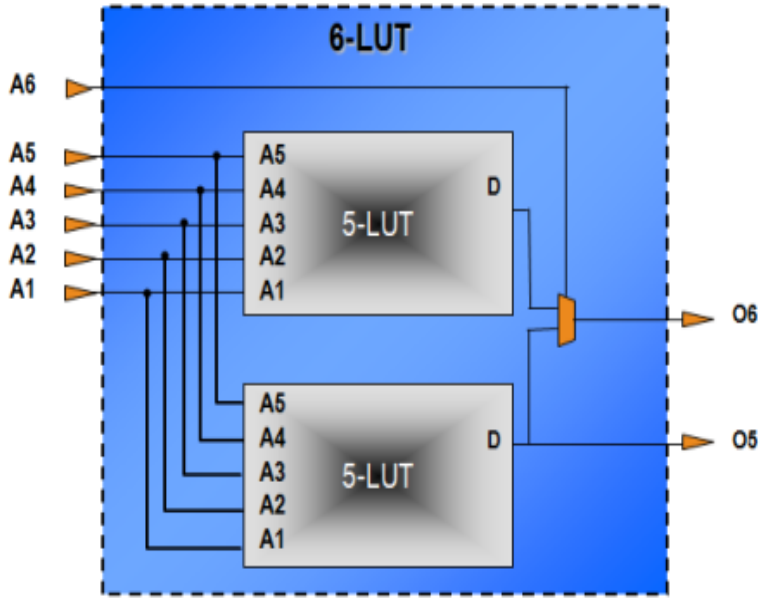


Truth table

a	b	cin	s
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



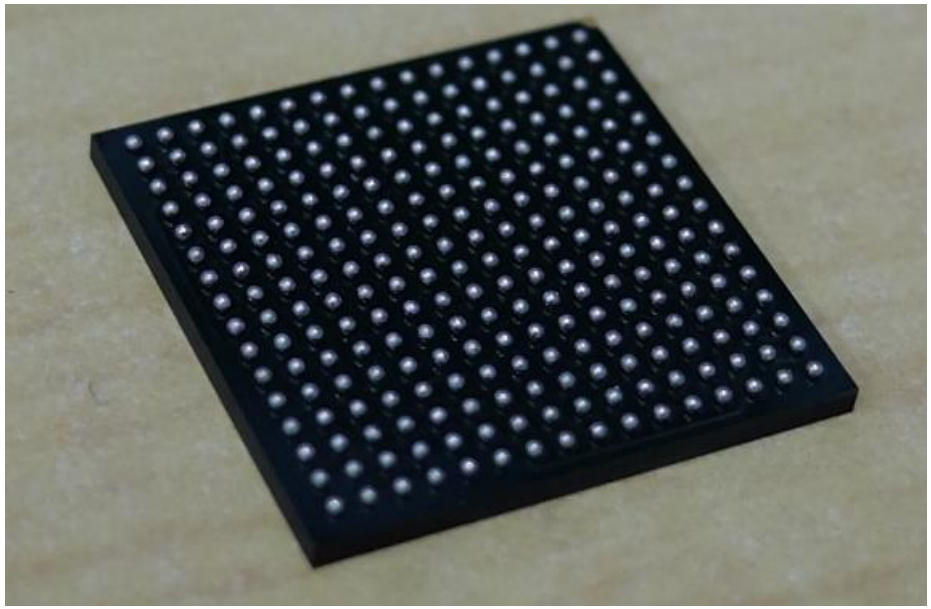
3-input LUT structure



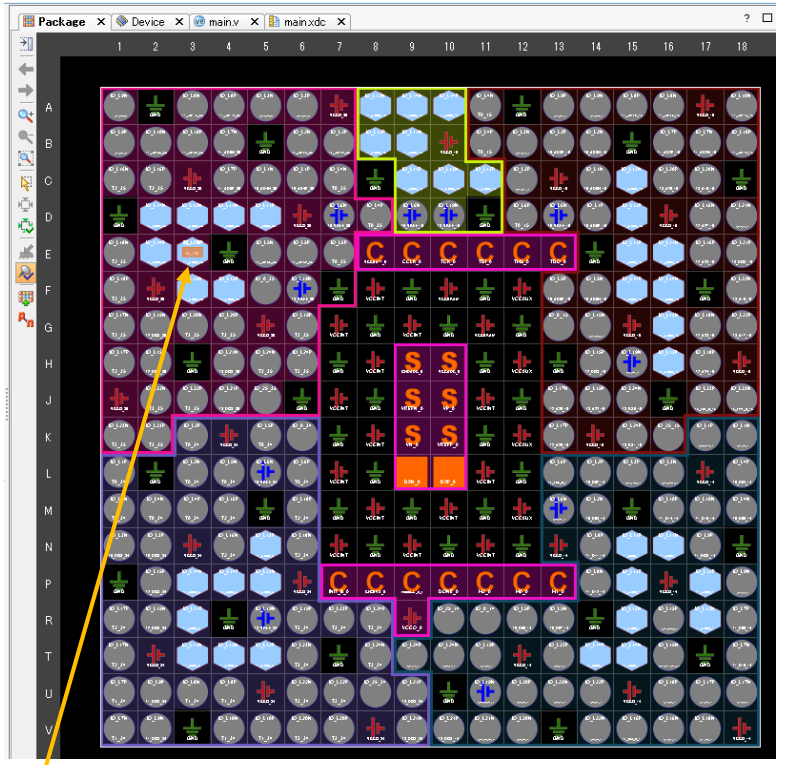
6-input LUT for Artix-7 FPGA

FPGA (Field Programmable Gate Array)

- Artix-7 FPGA (xc7a35tcsg324-1L)
- Vivado, open implemented design, and select Layout Menu, then I/O planning
  - 16 x 16 = 324 pins, max user pins of 250



FPGA chip photo

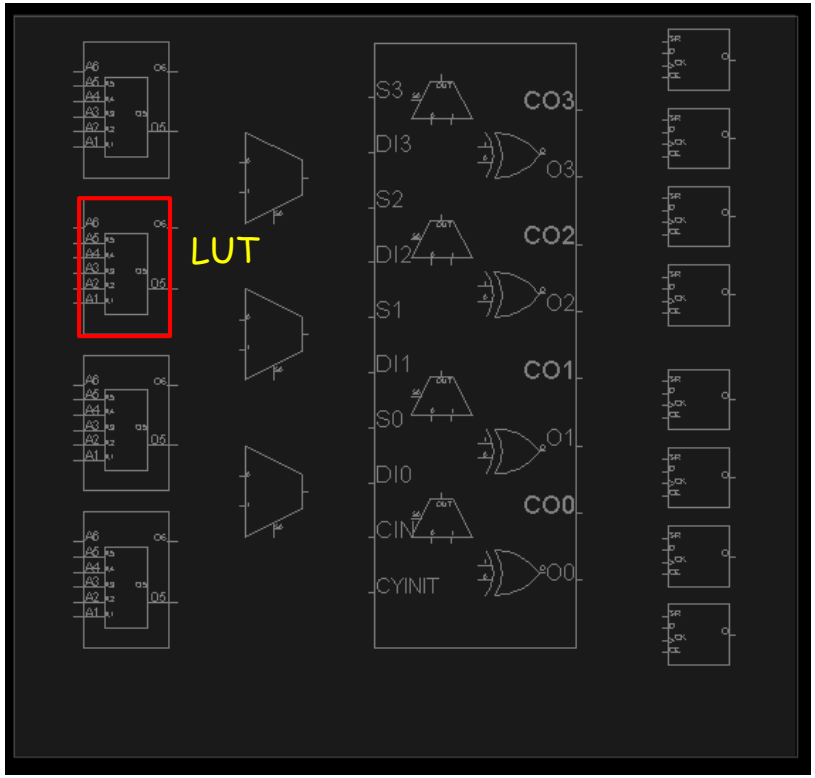


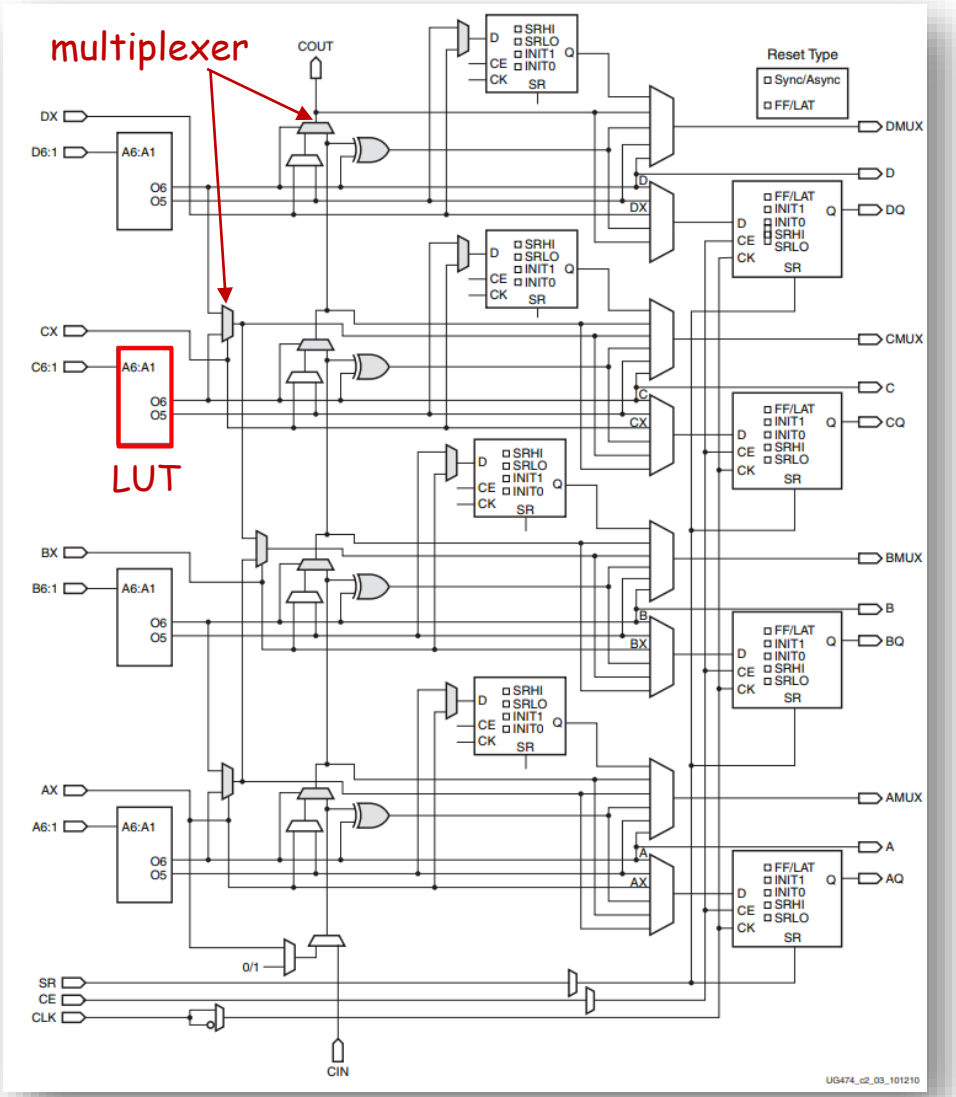
main11.xdcの最初の2行

```
set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33} [get_ports { w_clk }];
create_clock -add -name sys_clk -period 10.00 [get_ports {w_clk}];
```

Slice and SLICEL

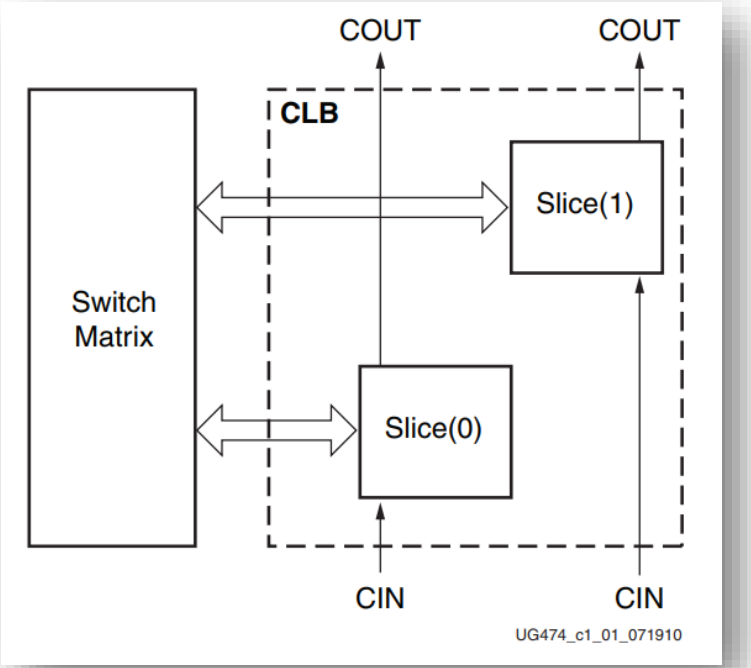
- There are two types of slices. They are SLICEL and SLICEM.
- Each slice has 4 LUTs and 8 FFs.





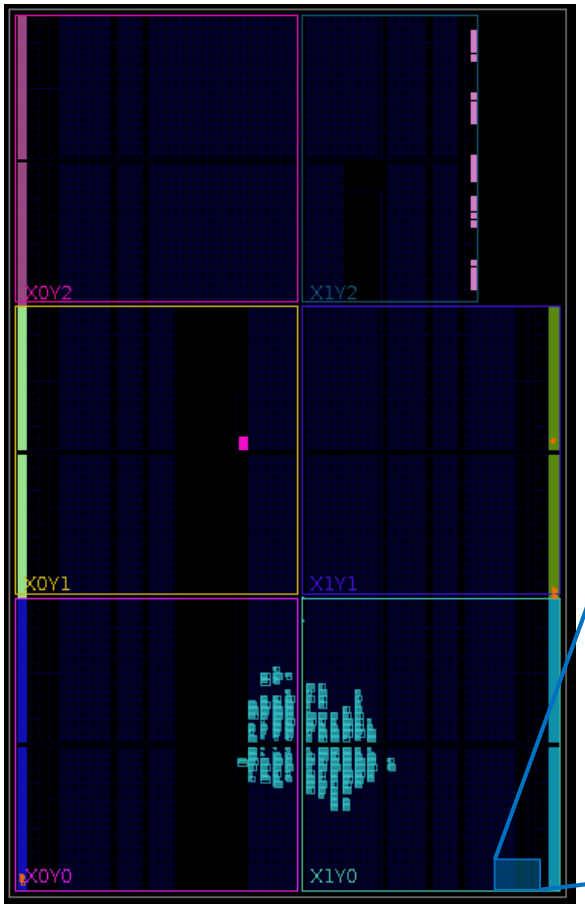
CLB (Configurable Logic Block) and Slice

- Each CLB has two slices
  - There are two types of slices. They are SLICEL and SLICEM.
  - Approximately two-thirds of the slices are SLICEL logic slices and the rest are SLICEM, which can also use their LUTs as distributed 64-bit RAM or as 32-bit shift registers (SRL32) or as two SRL16s.



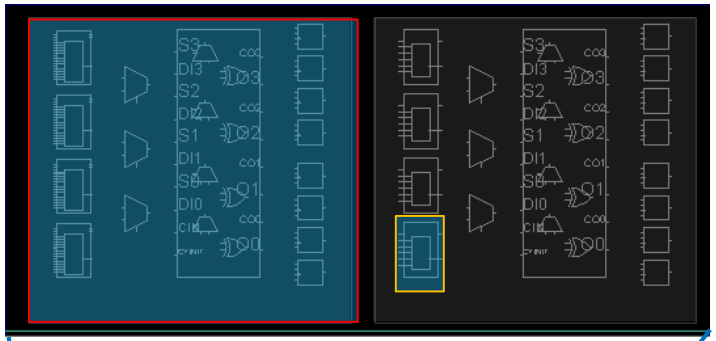
FPGA (Field Programmable Gate Array)

- Artix-7 FPGA (xc7a35tcsg324-1L)
  - 2,600 CLB = 5,200 slices



xc7a35tcsg324-1L FPGA die

Slice



LUT (Lookup Table)

CLB (Configurable Logic Block)

Xilinx 7 Series Configuration Logic Block (CLB)

7 Series FPGAs Configurable Logic Block

User Guide

UG474 (v1.8) September 27, 2016

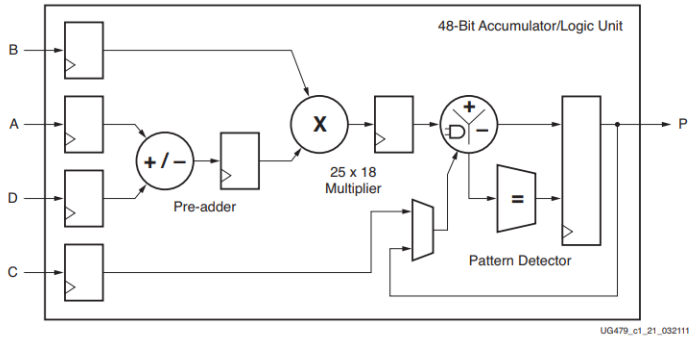
Slices = SLICEL + SLICEM
Distributed RAM (bit) = SLICEM x 256
Flip-Flops = LUTs x 2

Table 1-2: **Artix-7 FPGA CLB Resources**

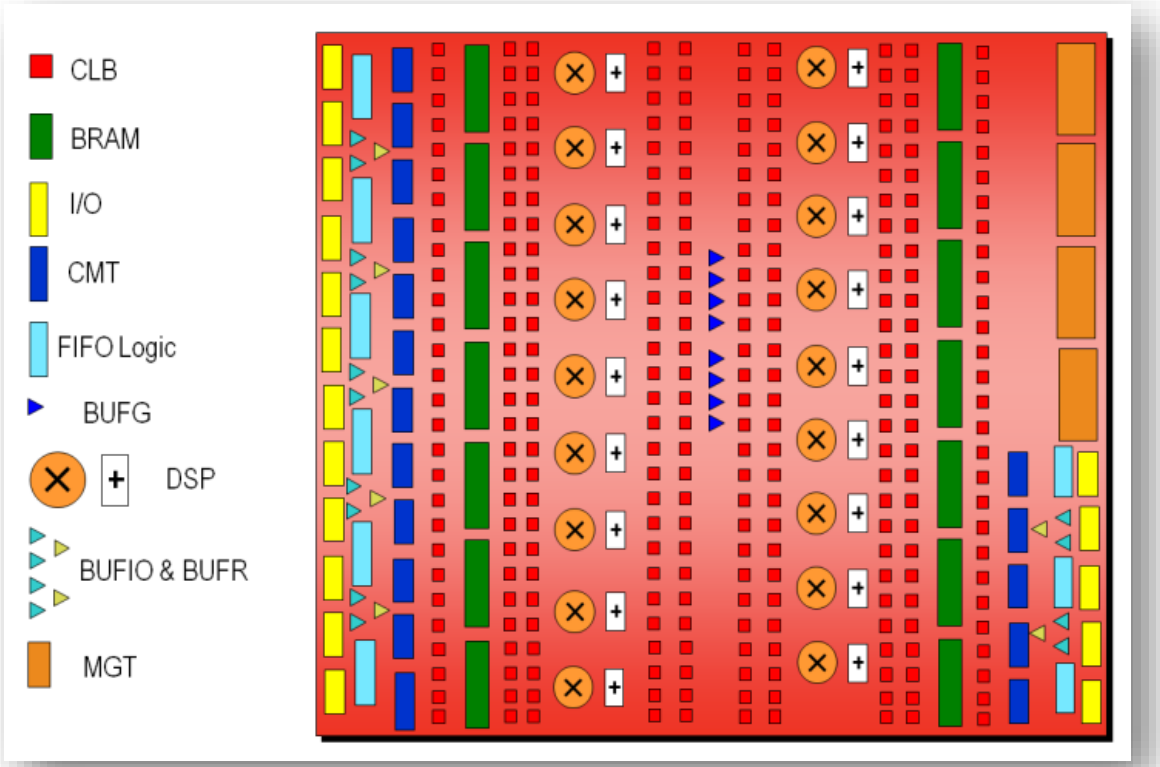
Device	Slices(1)	SLICEL	SLICEM	6-input LUTs	Distributed RAM (Kb)	Shift Register (Kb)	Flip-Flops
7A12T	2,000(2)	1,316	684	8,000	171	86	16,000
7A15T	2,600(2)	1,800	800	10,400	200	100	20,800
7A25T	3,650	2,400	1,250	14,600	313	156	29,200
7A35T	5,200(2)	3,600	1,600	20,800	400	200	41,600
7A50T	8,150	5,750	2,400	32,600	600	300	65,200
7A75T	11,800(2)	8,232	3,568	47,200	892	446	94,400
7A100T	15,850	11,100	4,750	63,400	1,188	594	126,800
7A200T	33,650	22,100	11,550	134,600	2,888	1,444	269,200

Artix-7 Architecture Overview

- CLB (Configurable Logic Block)
- BRAM (Block RAM, embedded memory)
- DSP (Digital Signal Processing)
- CMT (Clock Management Tile)
- Routing fabric

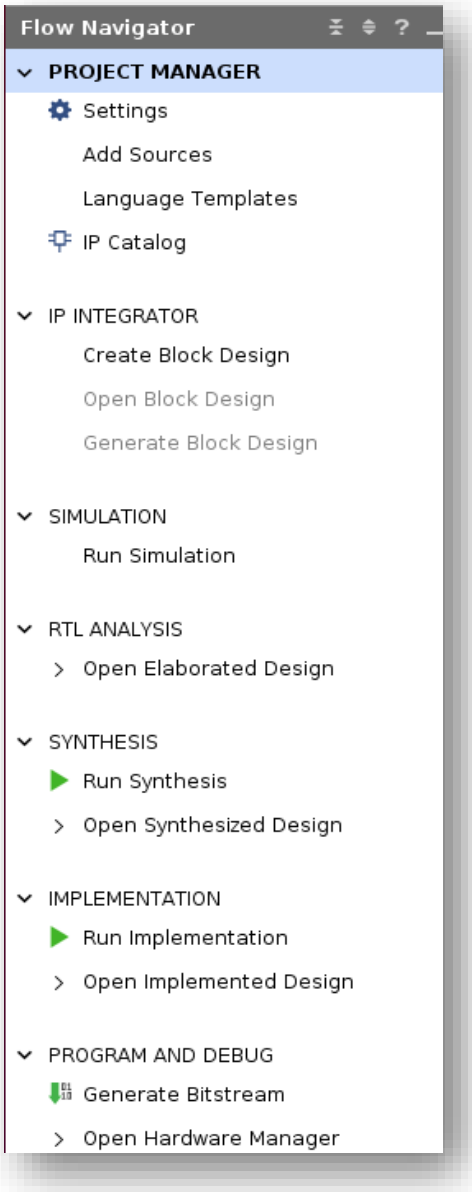


DSP Slice

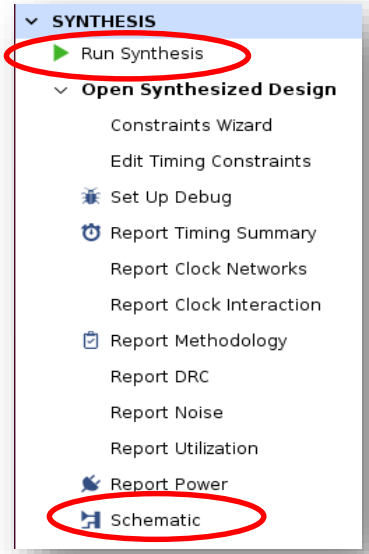


FPGA Design Flow

- IP Integrator
- Simulation
- RTL Analysis
- Synthesis
  - Logic Synthesis（論理合成），ネットリストの生成
- Implementation
  - Place and Route（配置・配線）
- Program and Debug
  - Bitstream generation
  - Program



Example Synthesis Result



code090.v

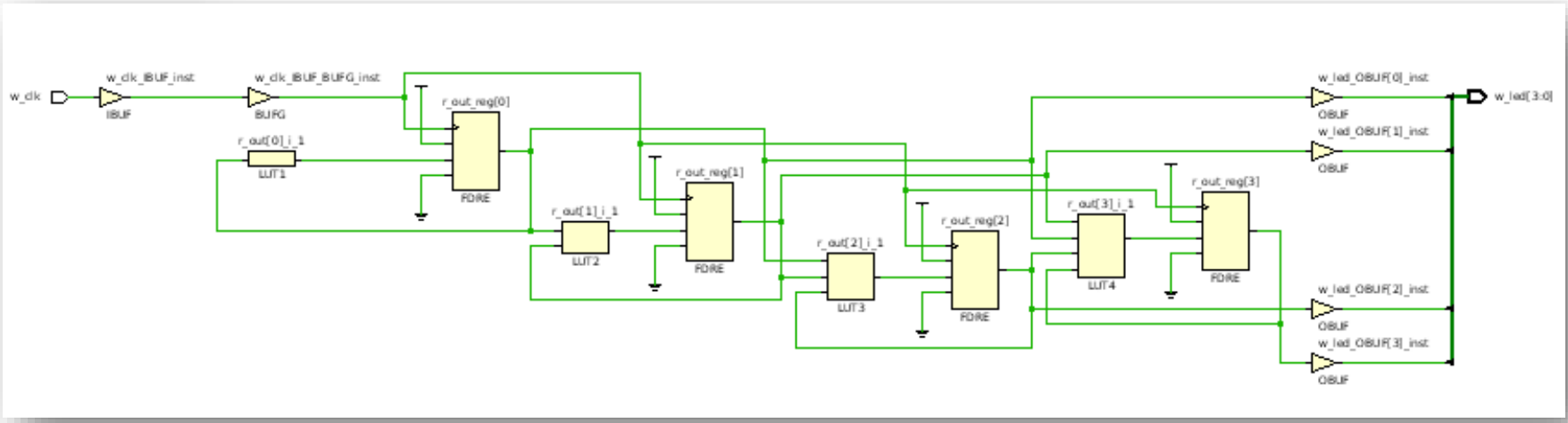
```
module m_main (w_clk, w_led);
  input  wire w_clk;
  output wire [3:0] w_led;

  reg [3:0] r_out = 0;
  always@(posedge w_clk) r_out <= r_out + 1;
  assign w_led = r_out;
endmodule
```

r_out[0]_i_1

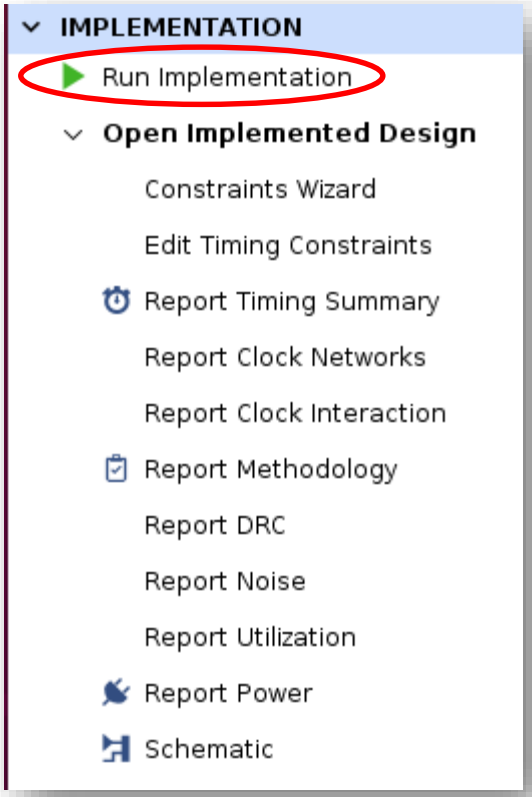
I0	O=!I0
0	1
1	0

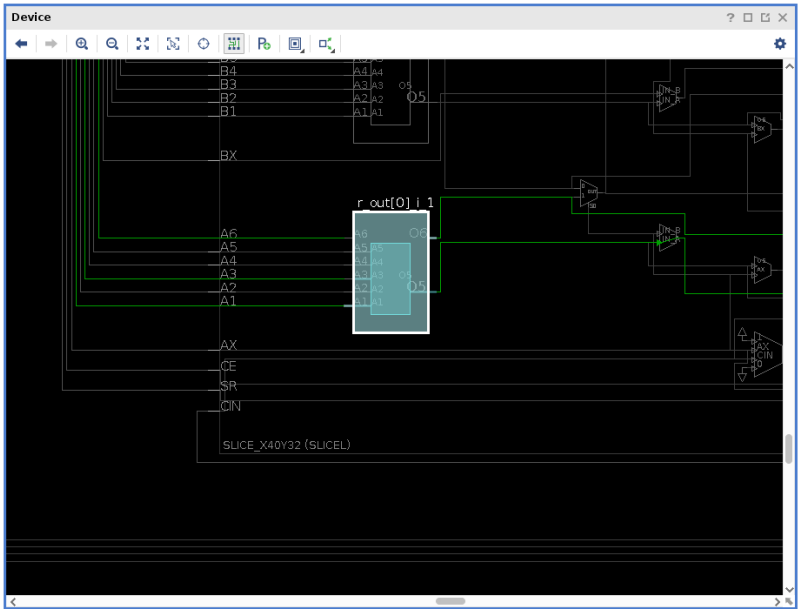
r_out[1]_i_1

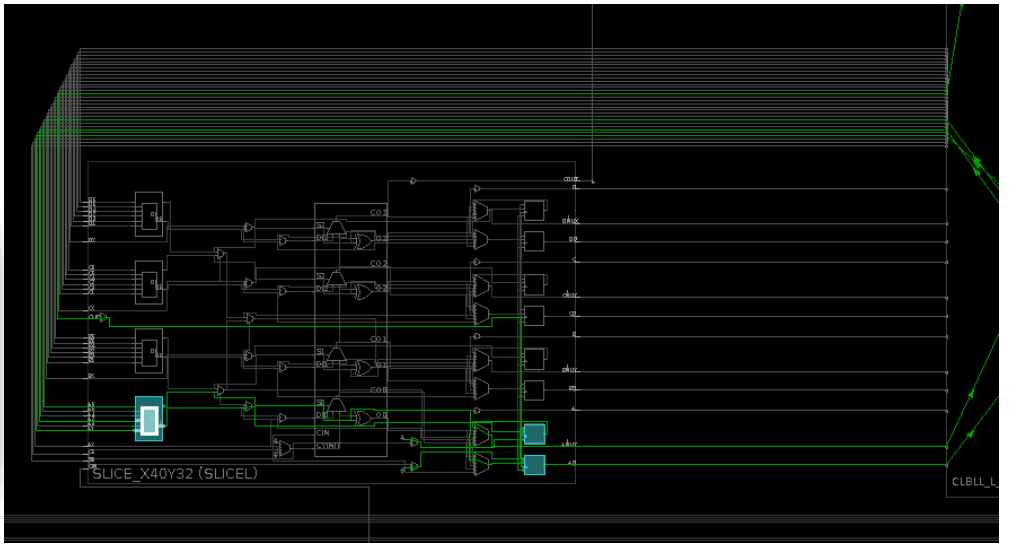
I1	I0	O=I0 & !I1 + !I0 & I1
0	0	0
0	1	1
1	0	1
1	1	0

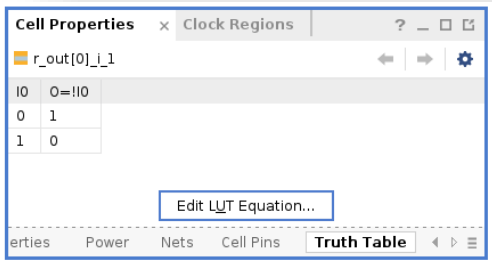


Example Implementation Result





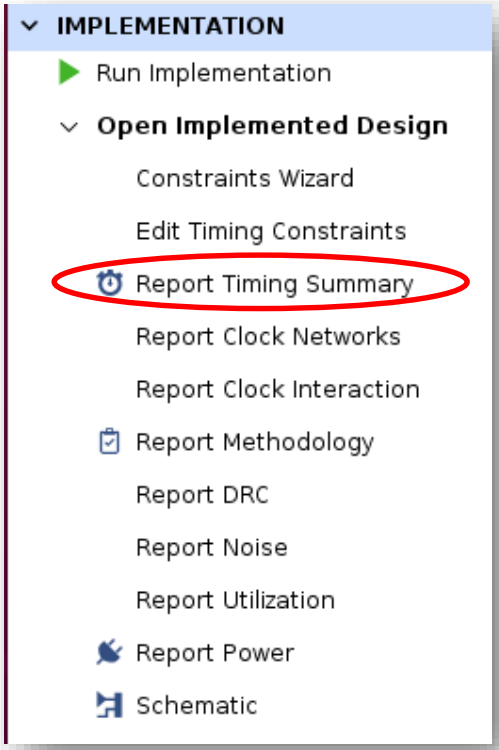


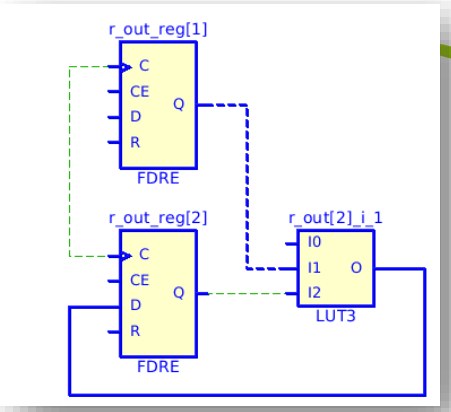


r_out[1]_i_1

I1	I0	O=I0 & !I1 + !I0 & I1
0	0	0
0	1	1
1	0	1
1	1	0

Example Implementation Result





Design Timing Summary

click here

Setup		Hold	
Worst Negative Slack (WNS):	6.674 ns	Worst Hold Slack (WHS):	0.465 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4	Total Number of Endpoints:	4

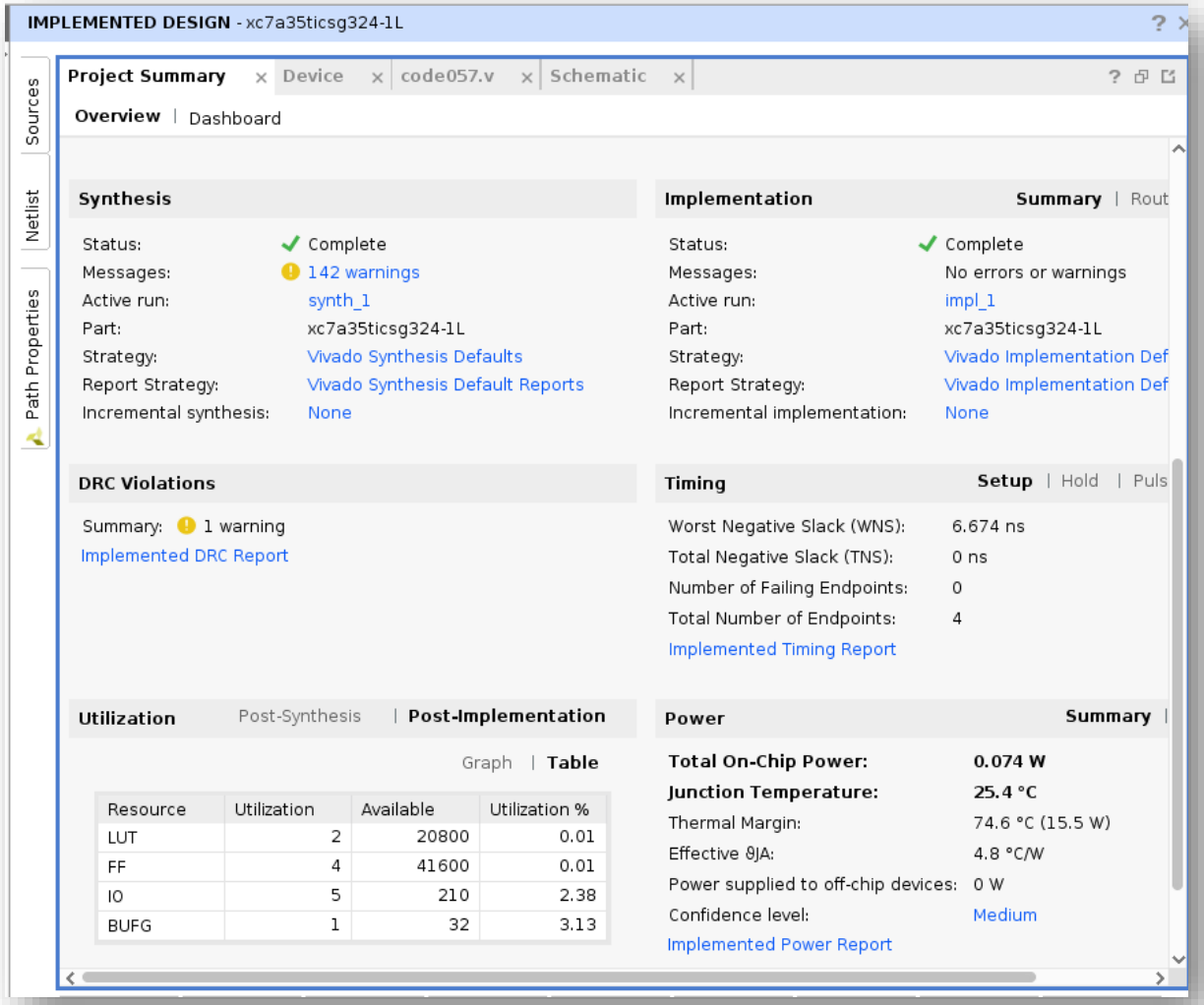
All user specified timing constraints are met.

Intra-Clock Paths - sys_clk - Setup

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement
Path 1	6.674	1	1	4	r_out_reg[1]/C	r_out_reg[2]/D	3.290	0.718	2.572	10.0
Path 2	6.694	1	1	4	r_out_reg[1]/C	r_out_reg[3]/D	3.316	0.744	2.572	10.0
Path 3	8.156	1	1	4	r_out_reg[1]/C	r_out_reg[1]/D	1.884	0.746	1.138	10.0
Path 4	8.172	1	1	5	r_out_reg[0]/C	r_out_reg[0]/D	1.822	0.580	1.242	10.0

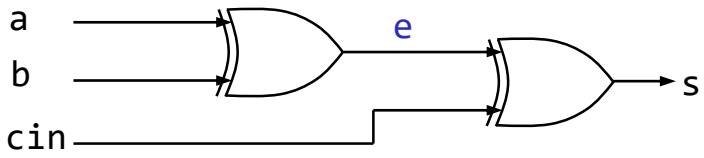
Example Implementation Result

- Project Summary Window
  - This window is useful to check the synthesis and implementation result at a glance.



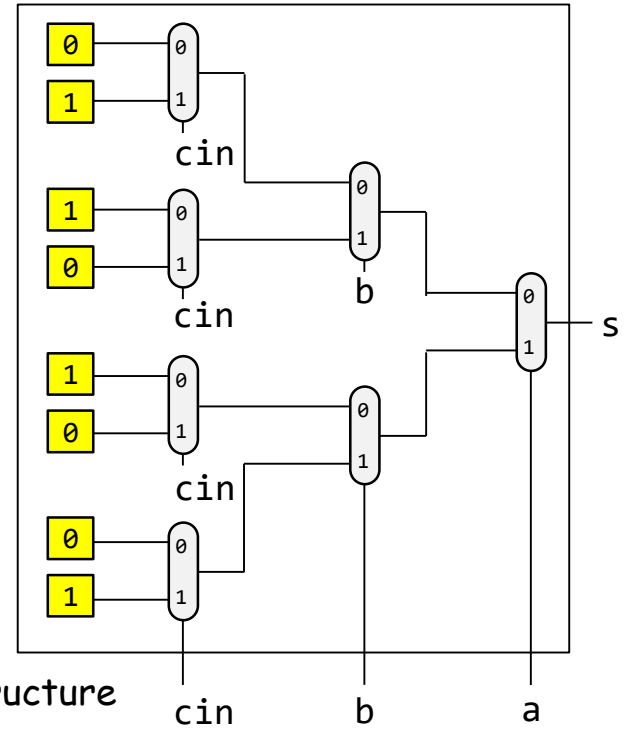
Distributed RAM としても利用できる LUT

- Reconfigurable Logic
  - 3-input LUT has 8 configuration registers
  - 6-input LUT for Artix-7 FPGA

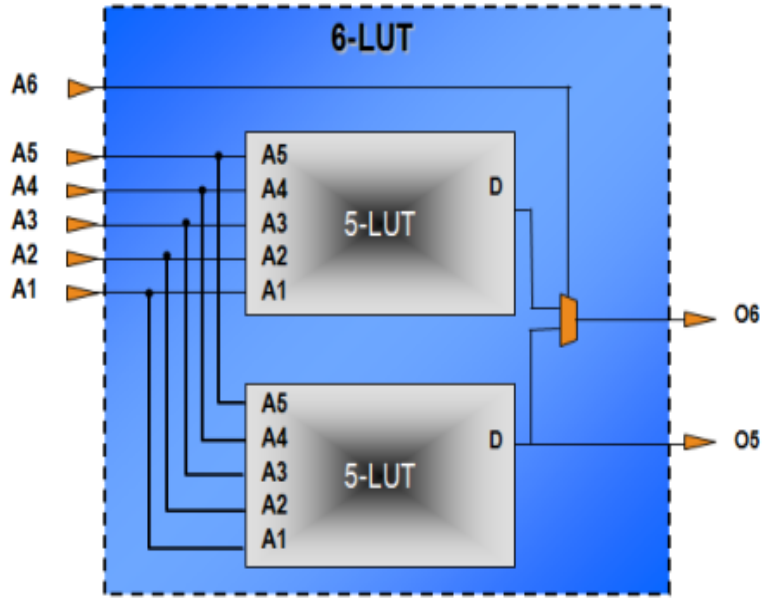


Truth table

a	b	cin	s
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



3-input LUT structure



6-input LUT for Artix-7 FPGA

Slice, SLICEL and SLICEM

Approximately two-thirds of the slices are SLICEL logic slices and the rest are SLICEM, which can also use their LUTs as distributed 64-bit RAM (LUTRAM) or as 32-bit shift registers (SRL32) or as two SRL16s.

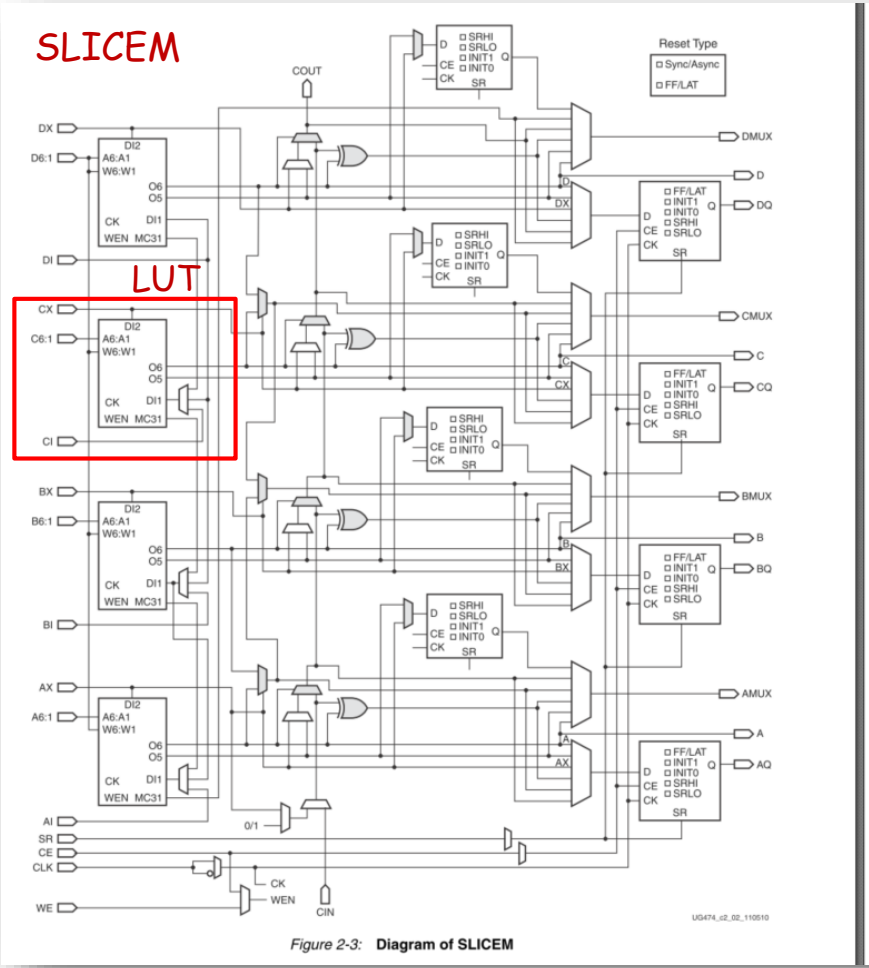


Figure 2-3: **Diagram of SLICEM**

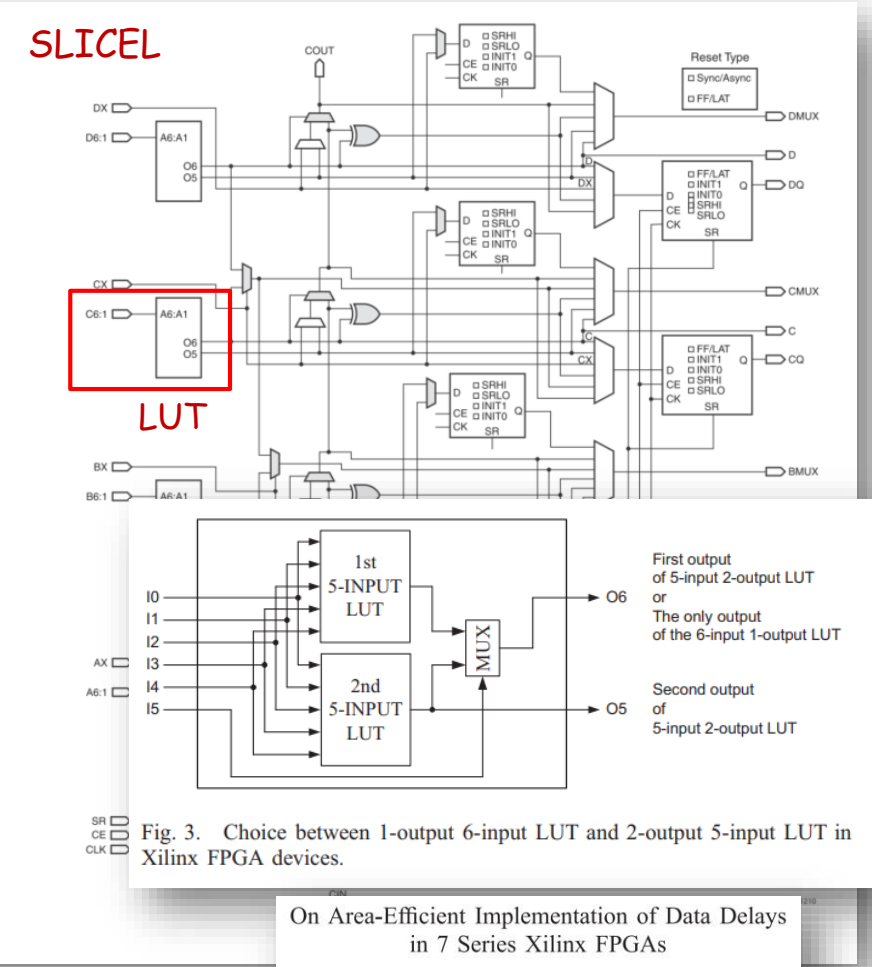


Fig. 3. Choice between 1-output 6-input LUT and 2-output 5-input LUT in Xilinx FPGA devices.

On Area-Efficient Implementation of Data Delays in 7 Series Xilinx FPGAs

Marek Parfieniuk
Department of Digital Media and Computer Graphics
Bialystok University of Technology
Wiejska 45A, 15-351 Bialystok, Poland
Email: m.parfieniuk@pb.edu.pl

Sang Yoon Park
Department of Electronic Engineering and MPEES-ARC
Myongji University
Yongin 449-728, Korea
Email: sypark@mju.ac.kr

Distributed RAM（分散メモリ）

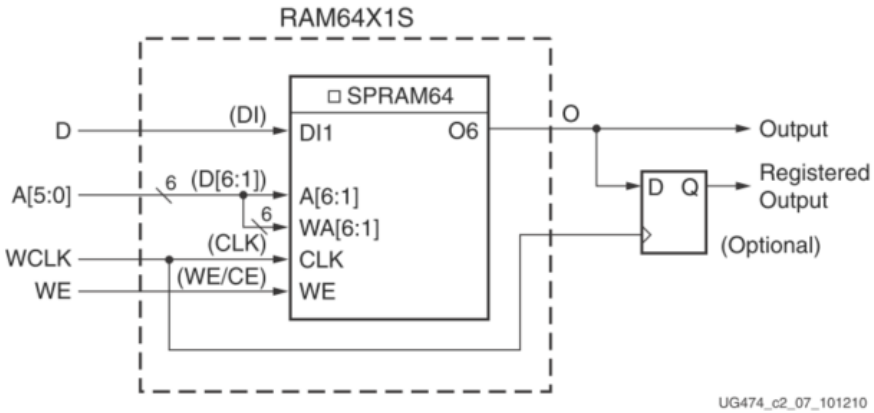


Figure 2-8: **64 X 1 Single Port Distributed RAM (RAM64X1S)**

```
module m_RAM64X1S (clk, a, d, we, dout);
  input wire clk;     // clock signal
  input wire [5:0] a; // address
  input wire d, we;   // data_in, write_enable
  output wire dout;   // data_out

  reg [0:0] mem [0:63];
  assign dout = mem[a];
  always @(posedge clk) if(we) mem[a] <= d;
endmodule
```

Table 2-3: **Distributed RAM Configuration**

RAM	Description	Primitive	Number of LUTs
32 x 1S	Single port	RAM32X1S	1
32 x 1D	Dual port	RAM32X1D	2
32 x 2Q	Quad port	RAM32M	4
32 x 6SDP	Simple dual port	RAM32M	4
64 x 1S	Single port	RAM64X1S	1
64 x 1D	Dual port	RAM64X1D	2
64 x 1Q	Quad port	RAM64M	4
64 x 3SDP	Simple dual port	RAM64M	4
128 x 1S	Single port	RAM128X1S	2
128 x 1D	Dual port	RAM128X1D	4
256 x 1S	Single port	RAM256X1S	4

- Single port
  - Common address port for synchronous writes and asynchronous reads
    - Read and write addresses share the same address bus

LUTRAM = 1

Distributed RAM（分散メモリ）

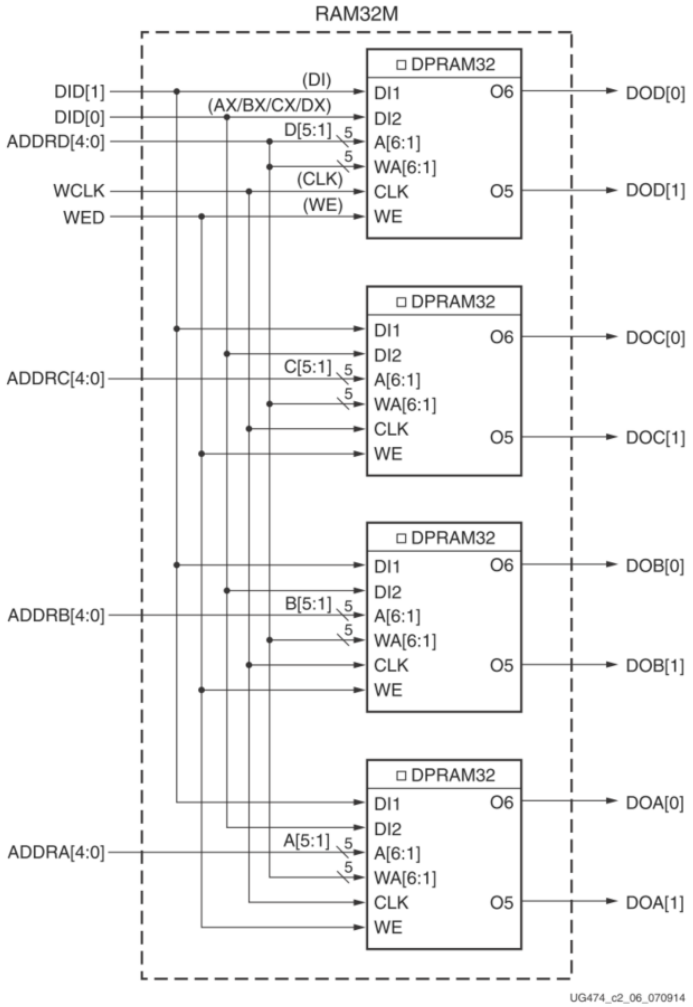


Figure 2-6: **32 X 2 Quad Port Distributed RAM (RAM32M)**

Table 2-3: **Distributed RAM Configuration**

RAM	Description	Primitive	Number of LUTs
32 x 1S	Single port	RAM32X1S	1
32 x 1D	Dual port	RAM32X1D	2
32 x 2Q	Quad port	RAM32M	4
32 x 6SDP	Simple dual port	RAM32M	4
64 x 1S	Single port	RAM64X1S	1
64 x 1D	Dual port	RAM64X1D	2
64 x 1Q	Quad port	RAM64M	4
64 x 3SDP	Simple dual port	RAM64M	4
128 x 1S	Single port	RAM128X1S	2
128 x 1D	Dual port	RAM128X1D	4
256 x 1S	Single port	RAM256X1S	4

- Quad port
  - One port for synchronous writes and asynchronous reads
  - Three ports for asynchronous reads

Distributed RAM（分散メモリ）

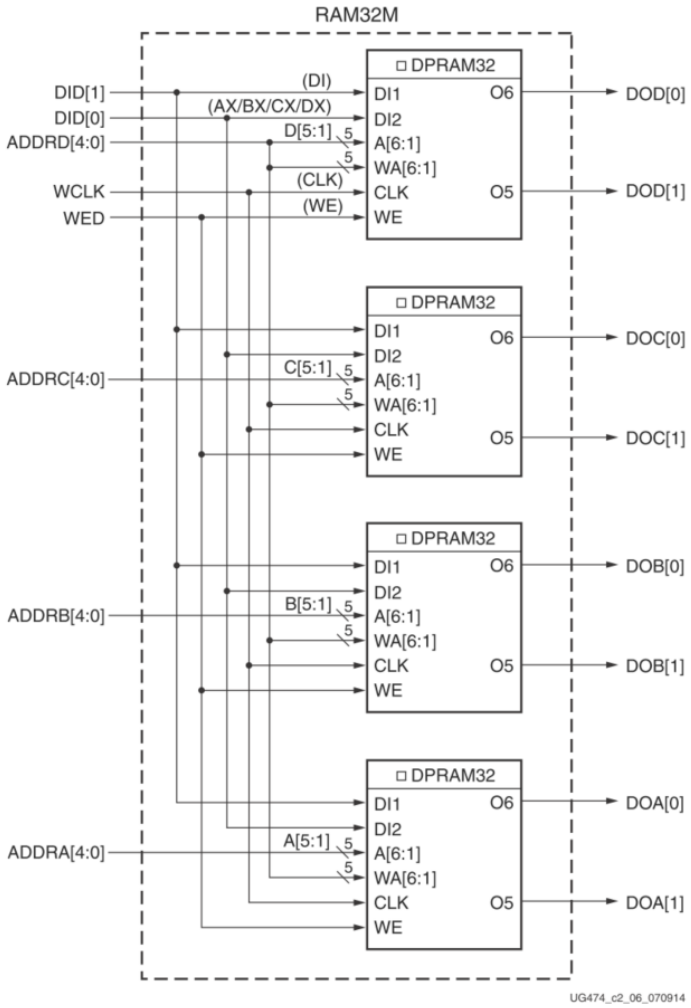
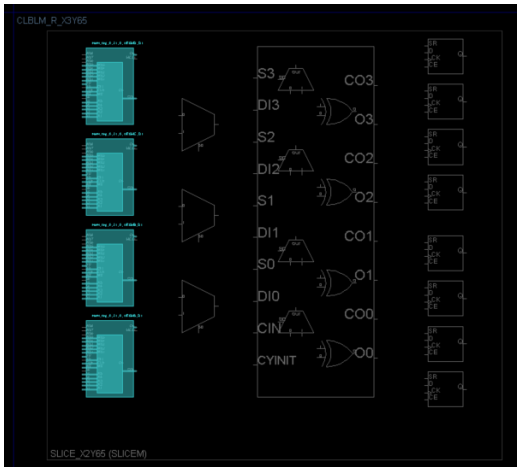


Figure 2-6: **32 X 2 Quad Port Distributed RAM (RAM32M)**

```
module m_RAM32M_Q (clk, a1, a2, a3, a4, d, we, dout1, dout2, dout3, dout4);
  input wire clk;
  input wire [4:0] a1, a2, a3, a4;
  input wire [1:0] d;
  input wire we;
  output wire [1:0] dout1, dout2, dout3, dout4;

  reg [1:0] mem [0:31];
  assign dout1 = mem[a1];
  assign dout2 = mem[a2];
  assign dout3 = mem[a3];
  assign dout4 = mem[a4];
  always @(posedge clk) if(we) mem[a1] <= d;
endmodule
```

Failed Routes	LUT	FF	BRAM	URAM	DSP
	4	0	0.0	0	0
0	4	0	0.0	0	0

LUTRAM = 4



Register file design of RISC-V soft processors

32x32 (32regs-32bit) single port register file

```
module m_RegFile_S (clk, a, d, we, dout);
  input wire clk;
  input wire [4:0] a;
  input wire [31:0] d;
  input wire we;
  output wire [31:0] dout;

  reg [31:0] mem [0:31];
  assign dout = mem[a];
  always @(posedge clk) if(we) mem[a] <= d;
endmodule
```

LUTRAM = 32

32x32 (32regs-32bit) dual port register file

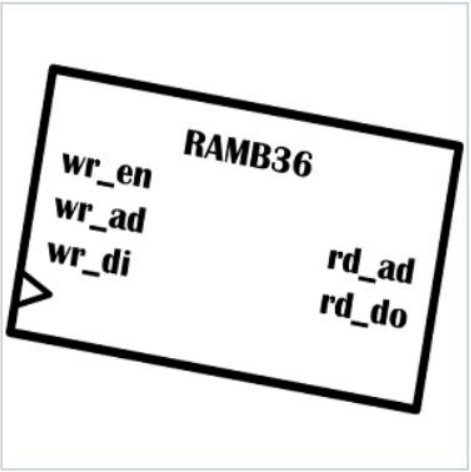
```
module m_RegFile_D (clk, a1, a2, d, we, dout1, dout2);
  input wire clk;
  input wire [4:0] a1, a2;
  input wire [31:0] d;
  input wire we;
  output wire [31:0] dout1, dout2;

  reg [31:0] mem [0:31];
  assign dout1 = mem[a1];
  assign dout2 = mem[a2];
  always @(posedge clk) if(we) mem[a1] <= d;
endmodule
```

Resource	Utilization	Available	Utilization %
LUT	32	63400	0.05
LUTRAM	32	19000	0.17
IO	108	210	51.43
BUFG	1	32	3.13

BRAM (Block RAM)

https://www.acri.c.titech.ac.jp/wordpress/archives/10236

BRAM 達人への道 (1) 構造と基本的な使い方



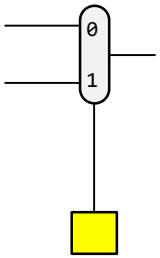
2021.01.21

FPGA を用いて AI のアクセラレーションのような高度な処理を行うことが当たり前の時代になりました。しかし、FPGA を利用する全てのエンジニアがそうしたアプリケーションに携わっているわけではなく、初心者、初級者レベルのエンジニアが多数おられるということも事実ではないかと思います。

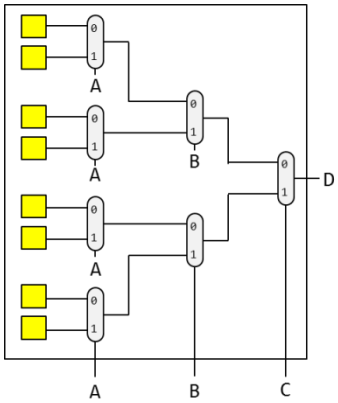
この記事では、現場レベルで使えるちょっとした工夫や考え方のヒントを紹介し、初級者のレベルアップや、現場の方々の手助けにつながればと考えています。

Configuration bitstream

- Bit vector of all configuration registers.
  - Program FPGA using a configuration chain of shift register like hardware.



Routing fabric



LUT

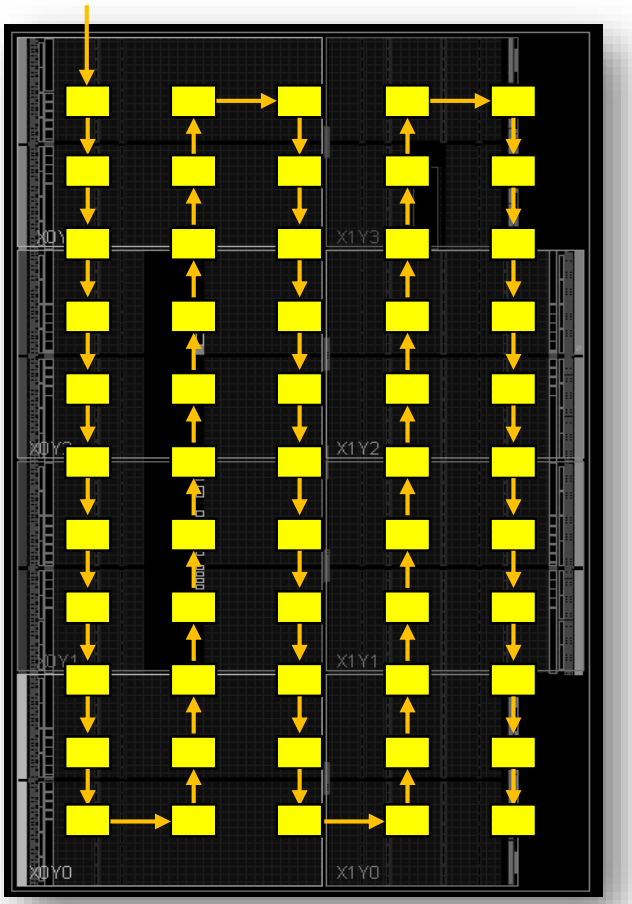
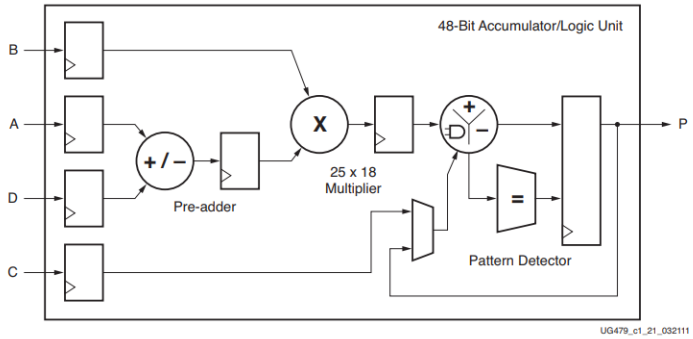


表 1-1：ビットストリームの長さ

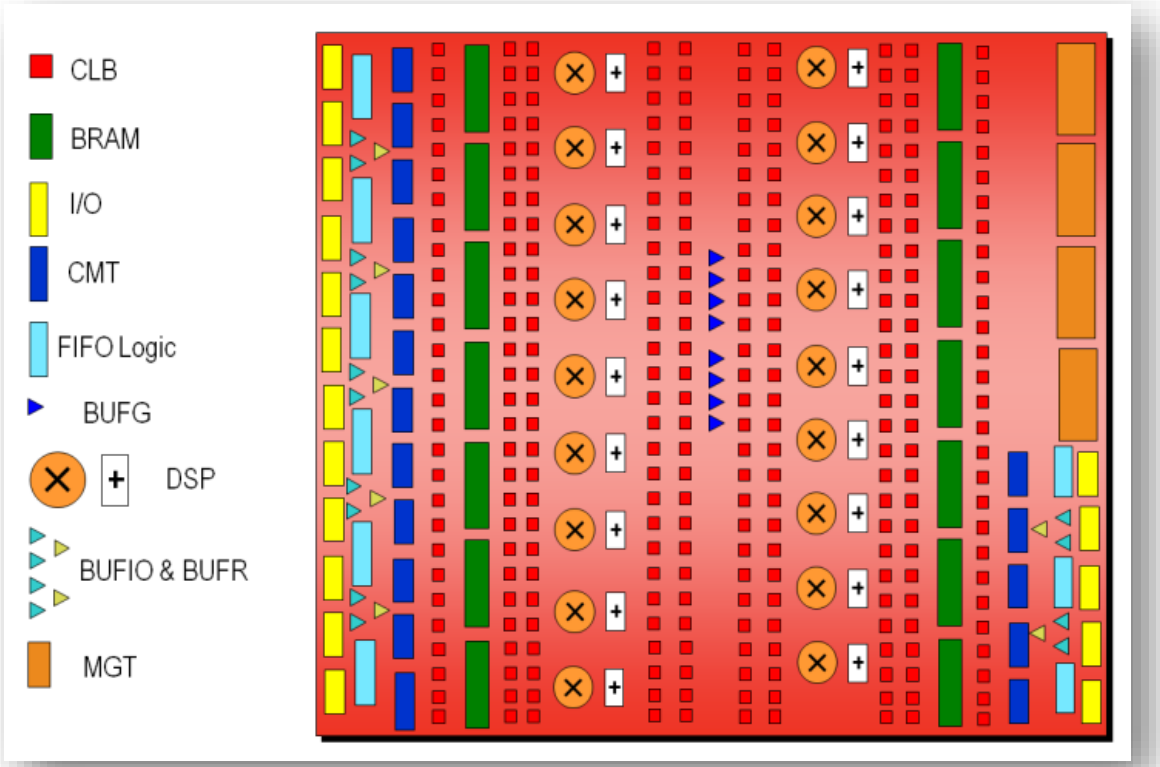
デバイス	コンフィギュレーション ビットストリームの長さ (ビット)
Artix-7 ファミリ	
7A15T	17,536,096
7A35T	17,536,096
7A50T	17,536,096
7A75T	30,606,304
7A100T	30,606,304
7A200T	77,845,216

Artix-7 Architecture Overview

- CLB (Configurable Logic Block)
- BRAM (Block RAM, embedded memory)
- DSP (Digital Signal Processing)
- CMT (Clock Management Tile)
- Routing fabric

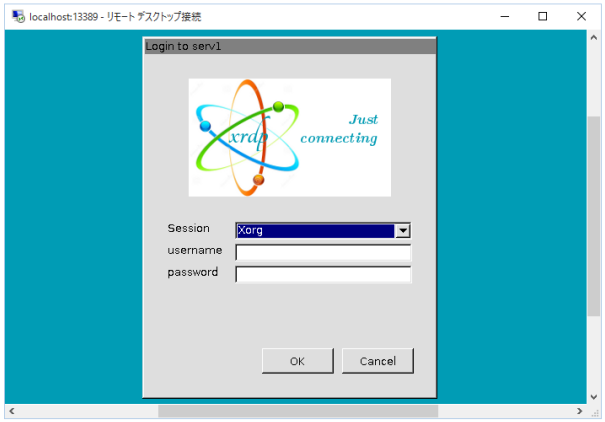


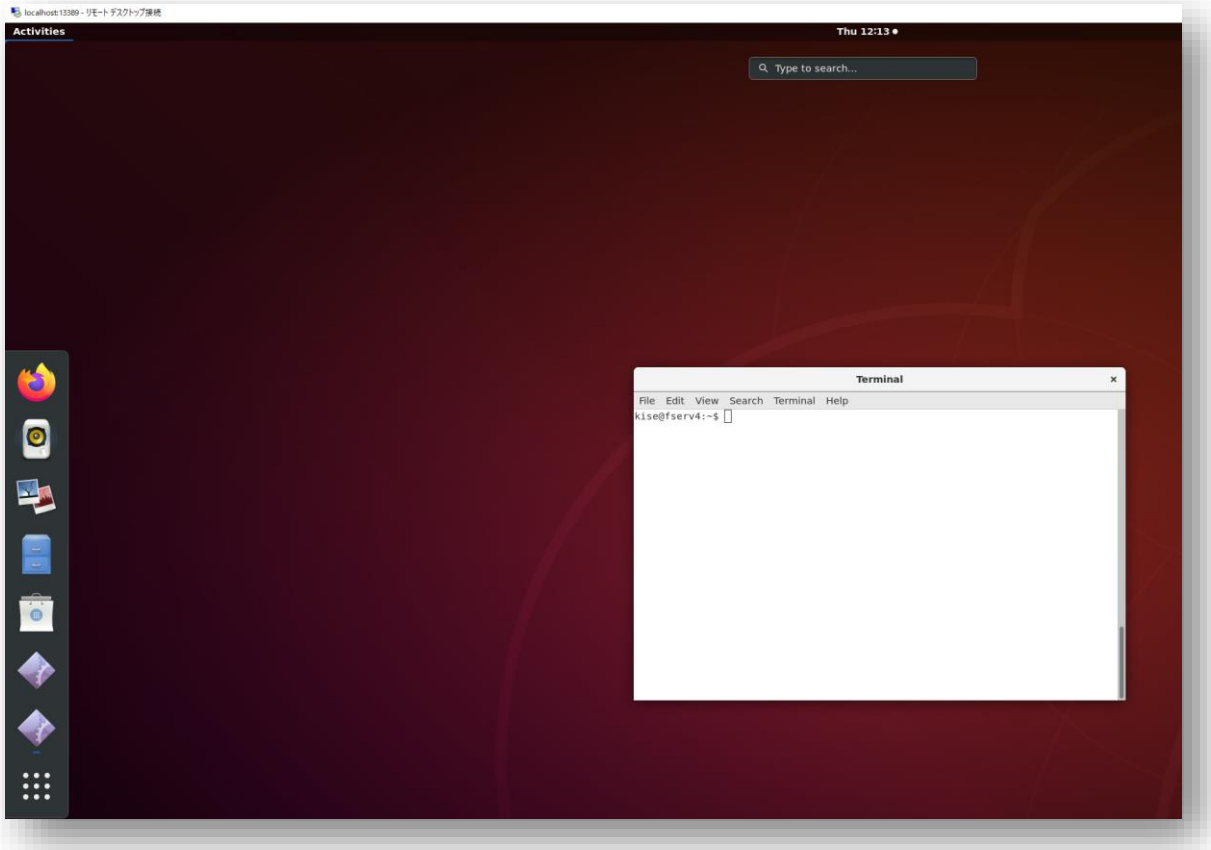
DSP Slice



ACRiルームのデモンストレーション

- LUT で実現される真理値表を確認する方法.
- Clocking Wizard を使って 20MHz のクロック信号を生成する方法.





Carry Cain in slice

electronics

MDPI

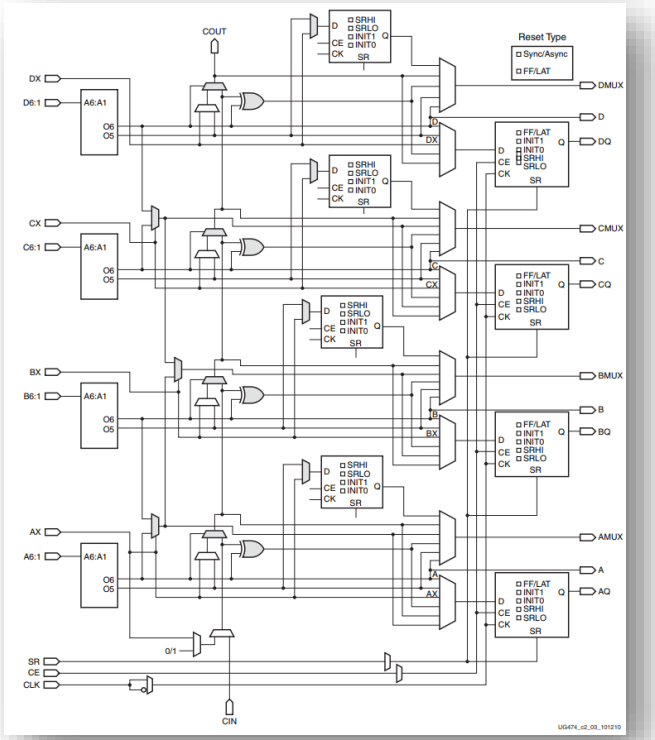
Article

Mapping Arbitrary Logic Functions onto Carry Chains in FPGAs

Raouf Senhadji-Navarro *,† and **Ignacio Garcia-Vargas** †

Department of Computer Architecture and Technology, University of Seville, 41012 Seville, Spain; iggv@us.es
* Correspondence: raouf@us.es
† These authors contributed equally to this work.

Abstract: Current Field Programmable Gate Arrays (FPGAs) provide fast routing links and special logic to perform carry operations; however, these resources can also be used to implement non-arithmetic circuits. In this paper, a new approach for mapping logic functions onto carry chains is presented. Unlike other approaches, the proposed technique can be applied to any logic function



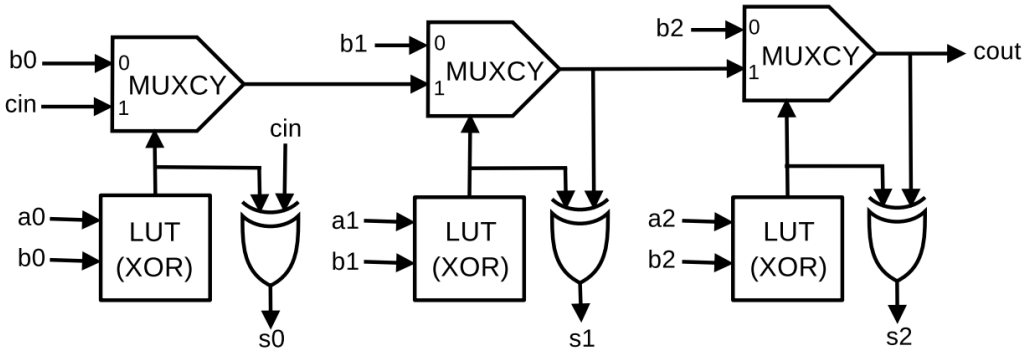


Figure 2. A 3-bit full adder implemented in a Xilinx FPGA device using carry chains.

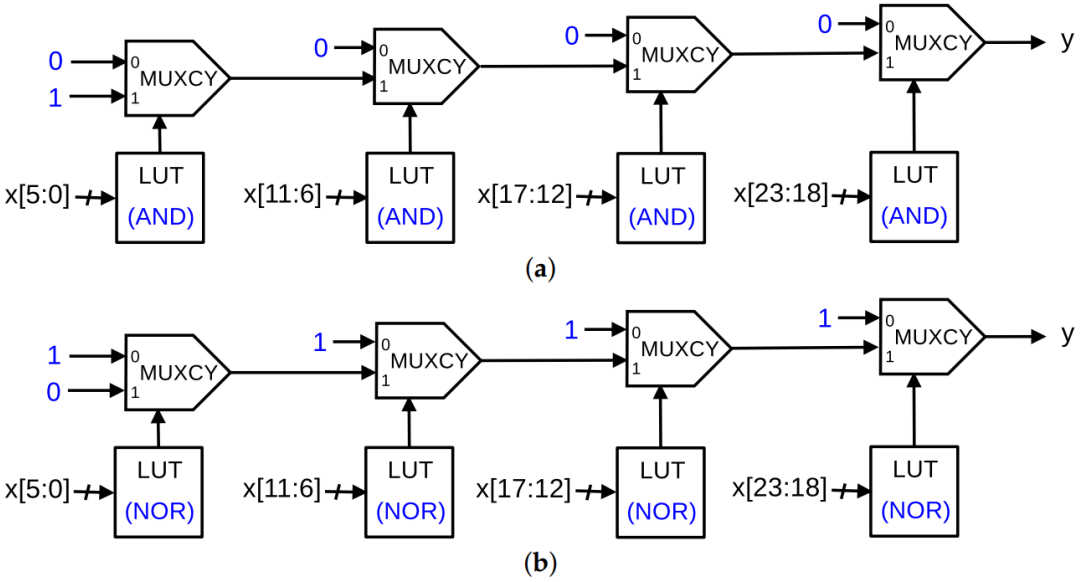


Figure 3. Examples of the implementation of wide input functions using carry chains: (**a**) 24-input AND and (**b**) 24-input OR.

Department of Computer Science
Course number: CSC.T341

コンピュータ論理設計
Computer Logic Design

6. 命令セットアーキテクチャ：命令形式とデータ表現
Instruction Set Architecture:
Instruction Format and Data Representation

吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
講義：月曜日 10:45-12:25, 木曜日 10:45-12:25

Sample Verilog HDL code

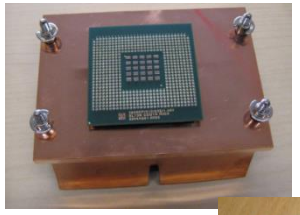
- ACRi Room のサーバーにリモートデスクトップでログインする.
  - /home/tu_kise/cld/lec6/ にサンプルのコードがあるので，Ubuntu のターミナルで次のコマンドを入力して，自分のディレクトリにコピーする.
  - /home/tu_kise は automount のディレクトリなので，アクセスしないとファイルが見えない．tabキーによる補完がうまく動作しないことがあるので注意する.

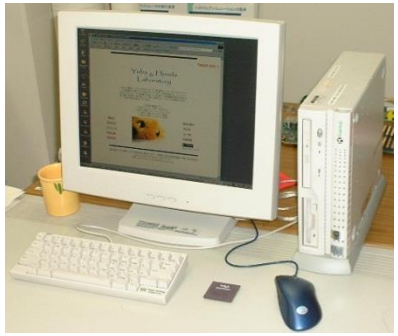
```
$ cd
$ mkdir cld
$ cd cld
$ cp /home/tu_kise/cld/lec6/* .
```

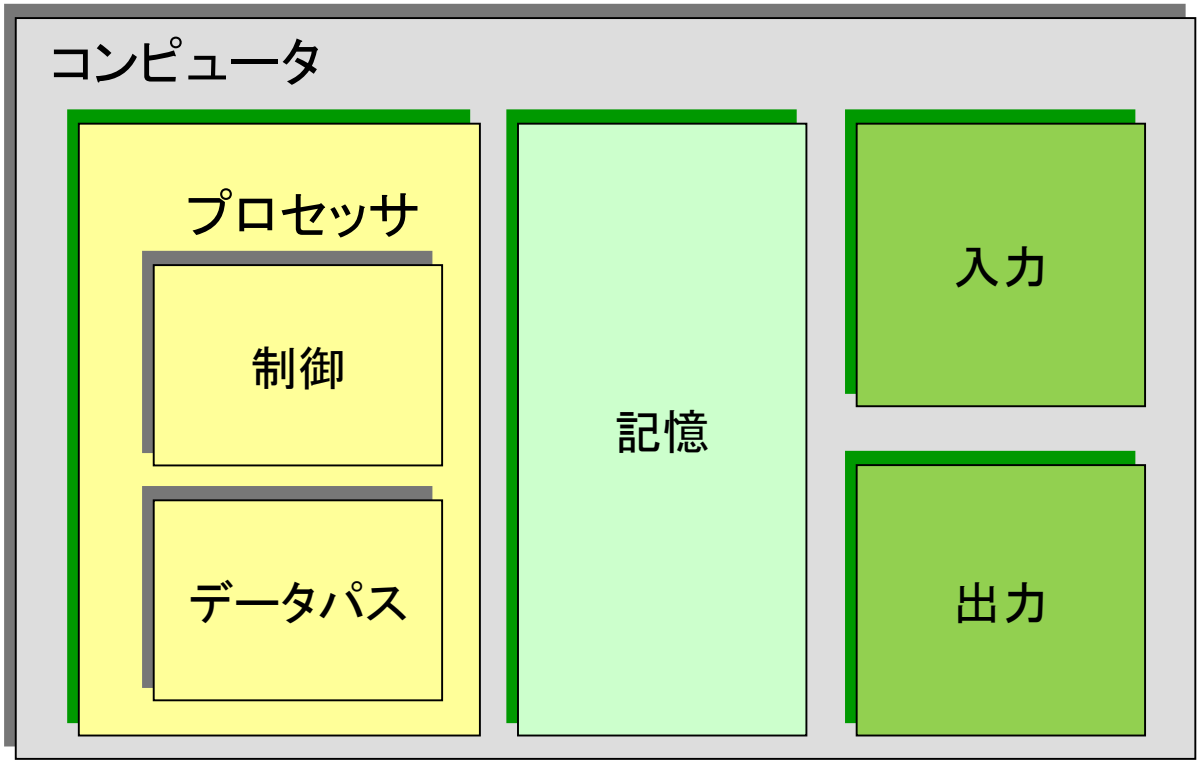
- code094.v をシミュレーションするためには.
  - リモートデスクトップで接続した Ubuntu のターミナルで次のコマンドを入力する.
  - コマンド iverilog でコンパイルして，生成される a.out を実行する.

```
$ iverilog code094.v
$ ./a.out
```

コンピュータ(ハードウェア)の古典的な要素







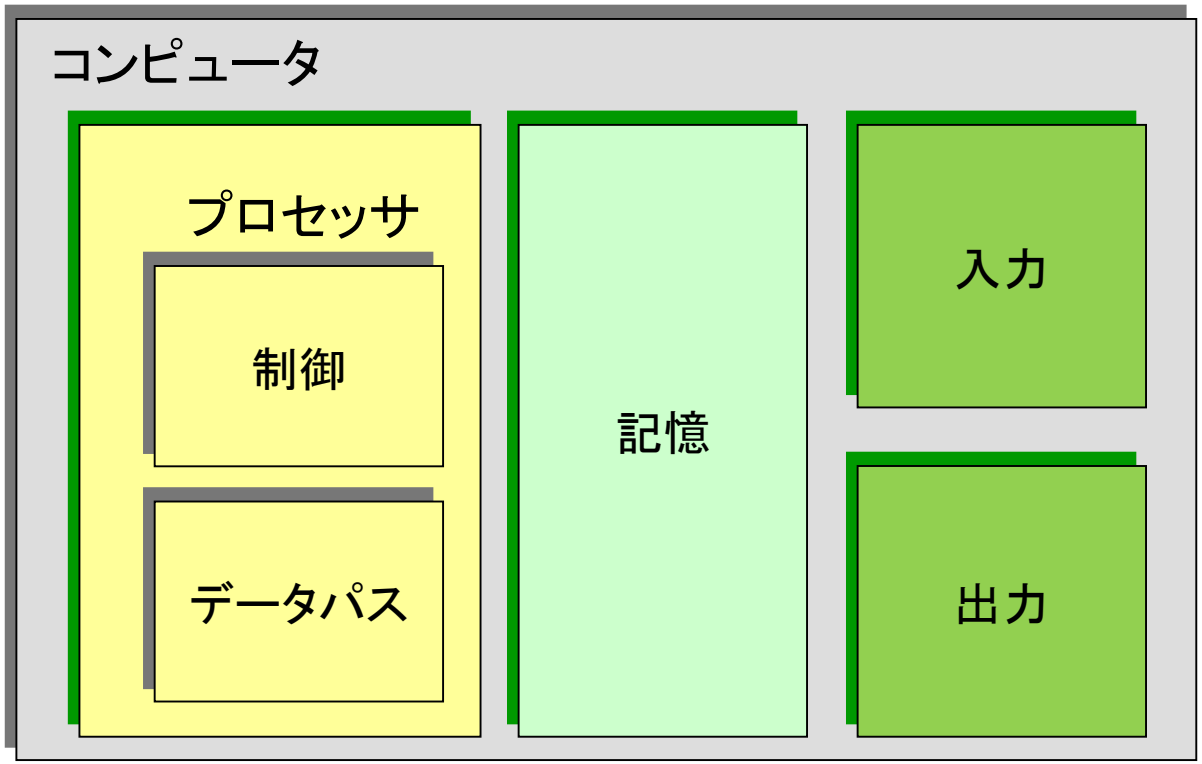
プロセッサは記憶装置から命令とデータを取り出す。入力装置はデータを記憶装置に書き込む。出力装置は記憶装置からデータを読みだす。制御装置は、データパス、記憶装置、入力装置、そして出力装置の動作を指定する信号を送る。

コンピュータの古典的な要素

コンパイラ

インタフェース　Instruction Set Architecture (ISA), 命令セットアーキテクチャ

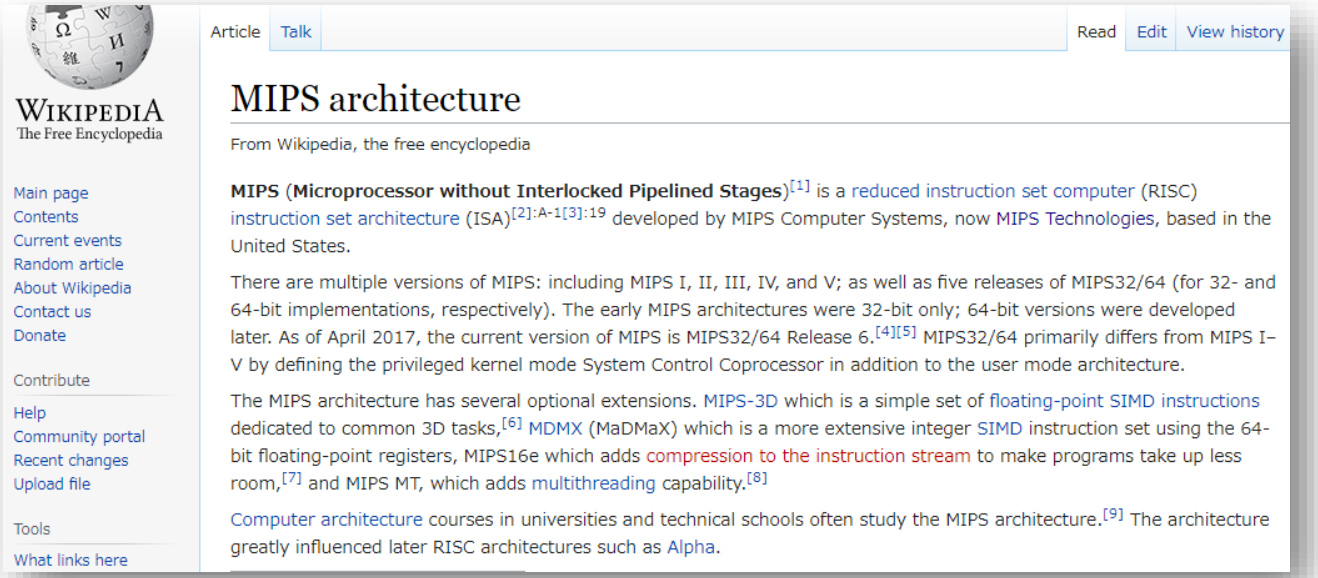
性能の評価



Two major ISA types: RISC vs CISC

- RISC (Reduced Instruction Set Computer) philosophy
  - fixed instruction lengths
  - load-store instruction sets
  - limited addressing modes
  - limited operations
  - RISC: MIPS, Alpha, ARM, RISC-V, …
- CISC (Complex Instruction Set Computer) philosophy
  - ! fixed instruction lengths
  - ! load-store instruction sets
  - ! limited addressing modes
  - ! limited operations
  - CISC : DEC VAX11, Intel 80x86, …

MIPS, ARM, and RISC-V



WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute
Help
Community portal
Recent changes
Upload file

Tools
What links here

Article Talk Read Edit View history

MIPS architecture

From Wikipedia, the free encyclopedia

MIPS (**Microprocessor without Interlocked Pipelined Stages**)[1] is a reduced instruction set computer (RISC) instruction set architecture (ISA)[2]:A-1[3]:19 developed by MIPS Computer Systems, now MIPS Technologies, based in the United States.

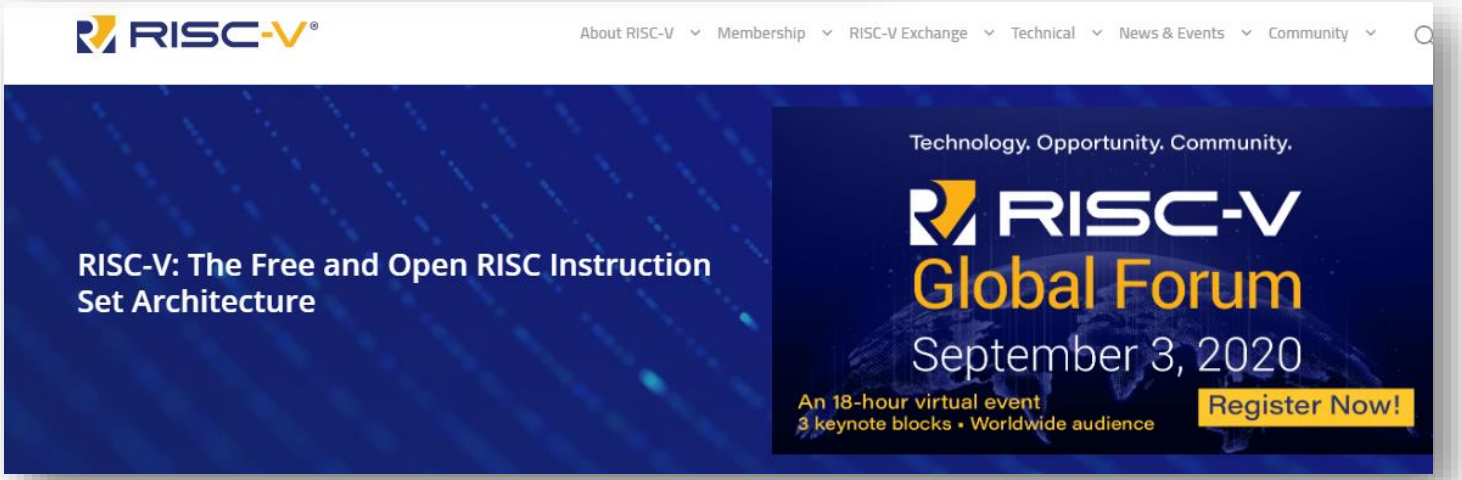
There are multiple versions of MIPS: including MIPS I, II, III, IV, and V; as well as five releases of MIPS32/64 (for 32- and 64-bit implementations, respectively). The early MIPS architectures were 32-bit only; 64-bit versions were developed later. As of April 2017, the current version of MIPS is MIPS32/64 Release 6.[4][5] MIPS32/64 primarily differs from MIPS I–V by defining the privileged kernel mode System Control Coprocessor in addition to the user mode architecture.

The MIPS architecture has several optional extensions. MIPS-3D which is a simple set of floating-point SIMD instructions dedicated to common 3D tasks,[6] MDMX (MaDMaX) which is a more extensive integer SIMD instruction set using the 64-bit floating-point registers, MIPS16e which adds compression to the instruction stream to make programs take up less room,[7] and MIPS MT, which adds multithreading capability.[8]

Computer architecture courses in universities and technical schools often study the MIPS architecture.[9] The architecture greatly influenced later RISC architectures such as Alpha.

ARM (Advanced RISC Machine)

https://en.wikipedia.org/wiki/MIPS_architecture



https://riscv.org/

C言語と RISC-V のアセンブリ言語で書いたプログラム例

main1.c

```
1  int main(){
2      int i=0, sum=1;
3      for(i=0; i<10; i++) sum = sum * i;
4      return sum;
5  }
```

main1.s

```
 1          .file   "main1.c"
 2          .option nopic
 3          .text
 4          .section        .text.startup,"ax",@progbits
 5          .align  2
 6          .globl  main
 7          .type   main, @function
 8  main:
 9          li      a0,1
10          li      a5,0
11          li      a4,10
12  .L2:
13          mul     a0,a0,a5
14          addi    a5,a5,1
15          bne     a5,a4,.L2
16          ret
17          .size   main, .-main
18          .ident  "GCC: (GNU) 11.1.0"
19          .section        .note.GNU-stack,"",@progbits
```

RISC-V の基本整数 ISA と機能拡張

ISA base and extensions (20191213)

Name	Description	Version	Status[a]
Base			
RVWMO	Weak Memory Ordering	2.0	Ratified
RV32I	Base Integer Instruction Set, 32-bit	2.1	Ratified
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 registers	1.9	Open
RV64I	Base Integer Instruction Set, 64-bit	2.1	Ratified
RV128I	Base Integer Instruction Set, 128-bit	1.7	Open
Extension			
M	Standard Extension for Integer Multiplication and Division	2.0	Ratified
A	Standard Extension for Atomic Instructions	2.1	Ratified
F	Standard Extension for Single-Precision Floating-Point	2.2	Ratified
D	Standard Extension for Double-Precision Floating-Point	2.2	Ratified
G	Shorthand for the base integer set (I) and above extensions (MAFD)	N/A	N/A
Q	Standard Extension for Quad-Precision Floating-Point	2.2	Ratified
L	Standard Extension for Decimal Floating-Point	0.0	Open
C	Standard Extension for Compressed Instructions	2.0	Ratified
B	Standard Extension for Bit Manipulation	0.92	Open
J	Standard Extension for Dynamically Translated Languages	0.0	Open
T	Standard Extension for Transactional Memory	0.0	Open
P	Standard Extension for Packed-SIMD Instructions	0.2	Open
V	Standard Extension for Vector Operations	0.9	Open
N	Standard Extension for User-Level Interrupts	1.1	Open
H	Standard Extension for Hypervisor	0.4	Open
ZiCSR	Control and Status Register (CSR)	2.0	Ratified
Zifencei	Instruction-Fetch Fence	2.0	Ratified
Zam	Misaligned Atomics	0.1	Open
Ztso	Total Store Ordering	0.1	Frozen

RISC-V の汎用レジスタ

XLEN = 32
for 32bit ISA

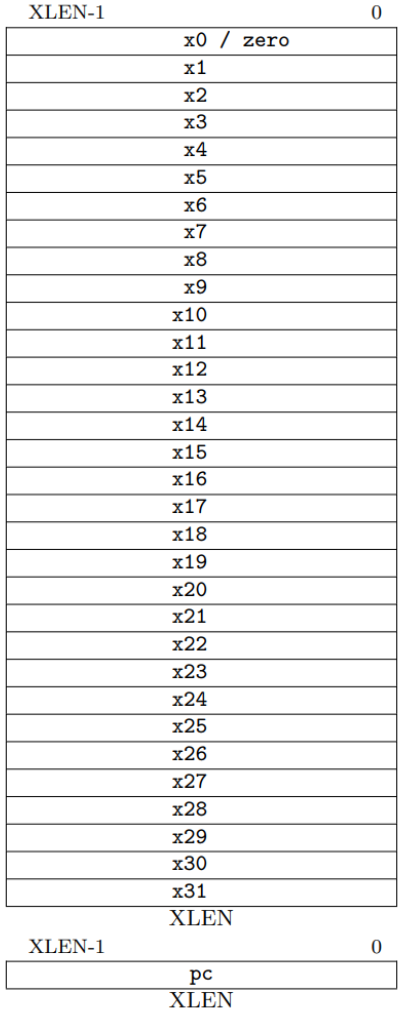


Figure 2.1: RISC-V base unprivileged integer register state.

ABI(Application Binary Interface) name

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Table 18.2: RISC-V calling convention register usage.

RISC-V の命令長

The RISC-V Instruction Set Manual
Volume I: Unprivileged ISA
Document Version 20191214-*draft*

Editors: Andrew Waterman[1], Krste Asanović[1,2]
[1]SiFive Inc.,
[2]CS Division, EECS Department, University of California, Berkeley
andrew@sifive.com, krste@berkeley.edu
November 12, 2021

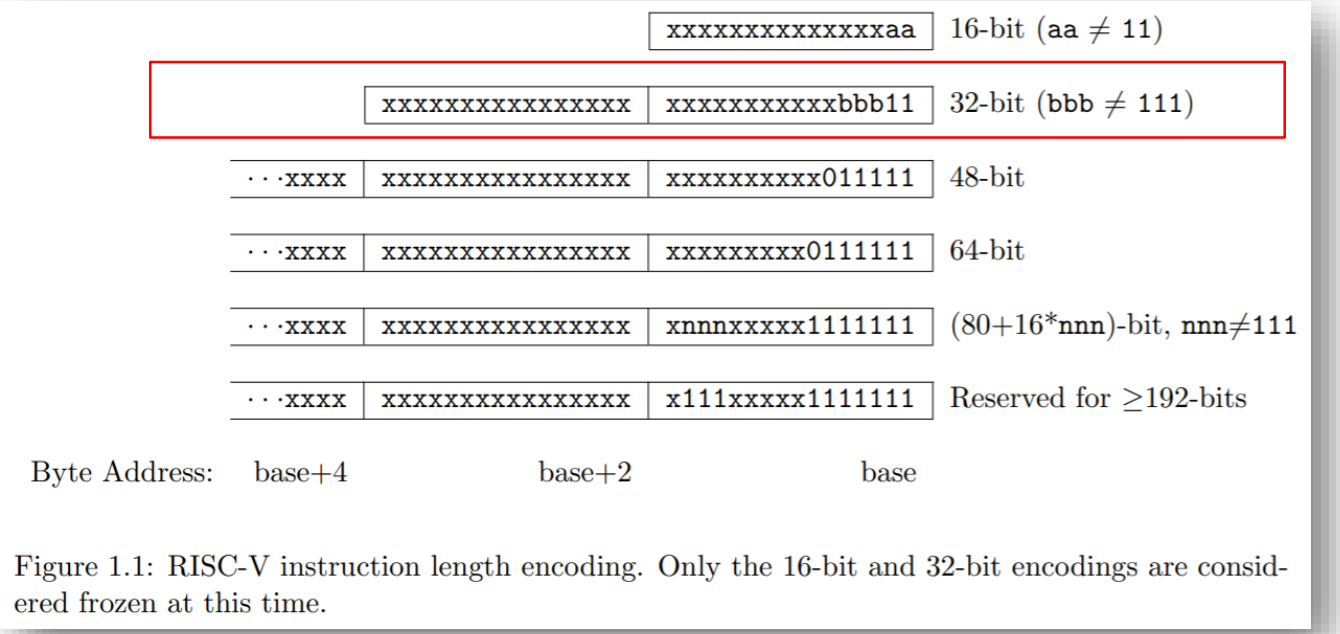


Figure 1.1: RISC-V instruction length encoding. Only the 16-bit and 32-bit encodings are considered frozen at this time.

RISC-V の命令フォーマット

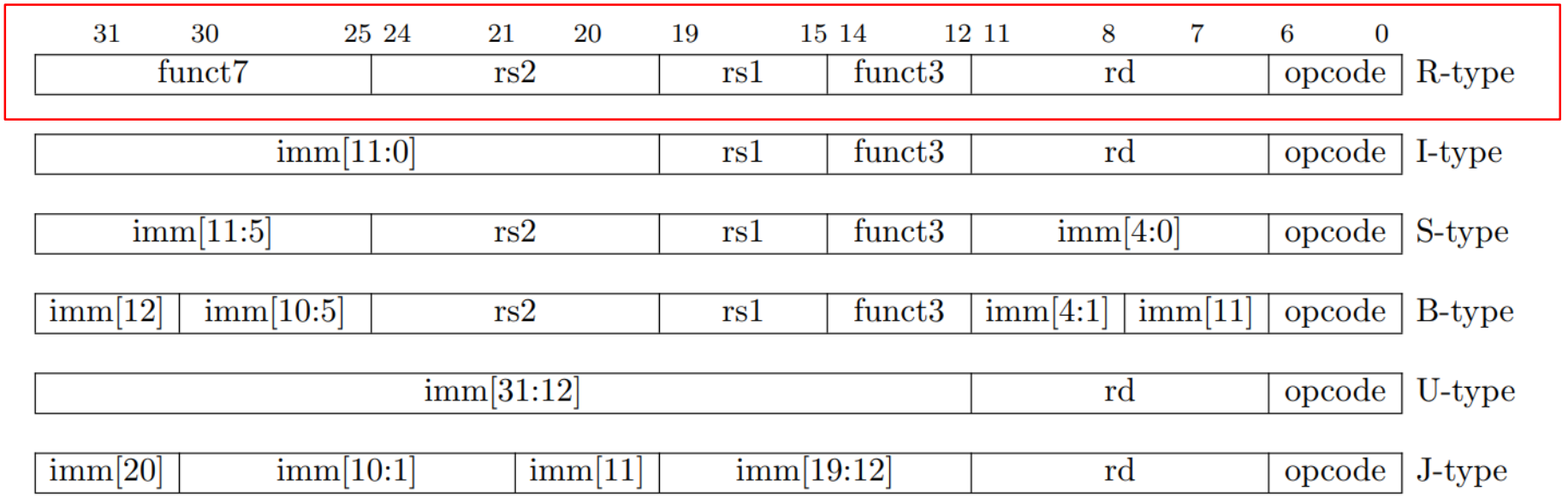


Figure 2.3: RISC-V base instruction formats showing immediate variants.

RISC-V Arithmetic Instructions

- RISC-V assembly language arithmetic statement

```
add   x7, x8, x9
sub   x7, x8, x9
```

- Each arithmetic instruction performs only one operation
- Each arithmetic instruction fits in 32 bits and specifies exactly three operands

```
destination <- source1 op source2
```

- Operand order is fixed (destination first)
- Those operands are all contained in the datapath's register file (x0, ..., x31)

Example（例題）

- `f = ( g + h ) - ( i + j )`

  f, g, h, i, j をそれぞれレジスタ x3, x4, x5, x6, x7 に割り付けるとする．上のステートメントをコンパイルした結果のRISC-Vのコードはどうなるか．

Answer

- $f = \underbrace{(g + h)}_{x10} - \underbrace{(i + j)}_{x11}$ （x3, x4, x5, x6, x7）

f, g, h, i, j をそれぞれレジスタ x3, x4, x5, x6, x7 に割り付けるとする．上のステートメントをコンパイルした結果のRISC-Vのコードはどうなるか．

```
add x10, x4, x5   # x10 = (g + h)
add x11, x6, x7   # x11 = (i + j)
sub x3, x10, x11  # f   = x10 - x11
```

計算結果を保存する一時的なレジスタとして x10, x11を用いたが，別のレジスタを用いても良い．

Machine Language - Add Instruction

- Instructions, like registers and words of data, are 32 bits long
- Arithmetic Instruction Format (R-type):

```
add  x7, x8, x9
```

R-type

opcode 7-bits *opcode* that specifies the operation

rs1 5-bits register file address of the first source operand

rs2 5-bits register file address of the second source operand

rd 5-bits register file address of the result's destination

funct3 and **funct7** 10-bits select the type of operation (function)

RISC-V の32ビット基本命令セット RV32I

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

Table 24.1: RISC-V base opcode map, inst[1:0]=11

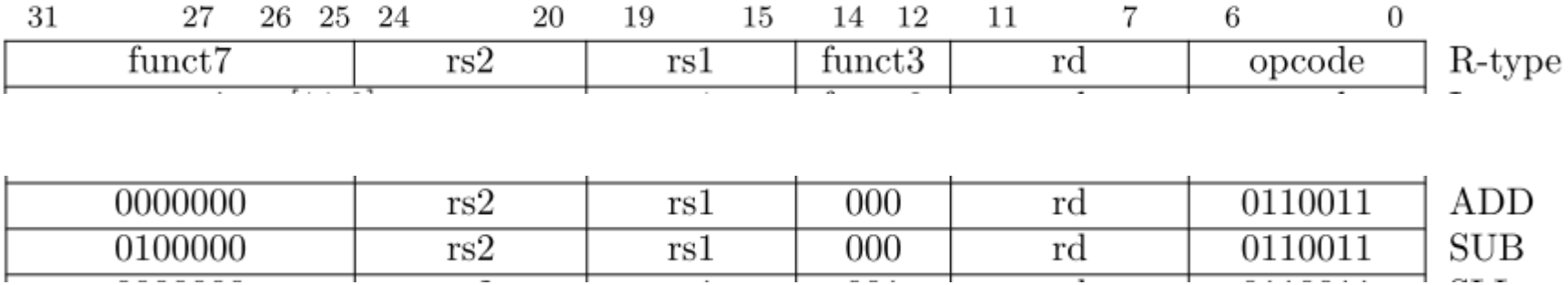
RV32I Base Instruction Set

imm[31:12]			rd	0110111	LUI
imm[31:12]			rd	0010111	AUIPC
imm[20|10:1|11|19:12]			rd	1101111	JAL
imm[11:0]	rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
				n[4:1|11]	1100011	BLTU
				n[4:1|11]	1100011	BGEU
				rd	0000011	LB
				rd	0000011	LH
				rd	0000011	LW
				rd	0000011	LBU
				rd	0000011	LHU
				m[4:0]	0100011	SB
				m[4:0]	0100011	SH
				m[4:0]	0100011	SW
				rd	0010011	ADDI
				rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Example（例題）

- 次のRISC-Vの命令列の機械語コードはどうなるか．それぞれの命令を2進数と16進数で表示する Verilog HDLのコードを記述して，その結果を示せ．

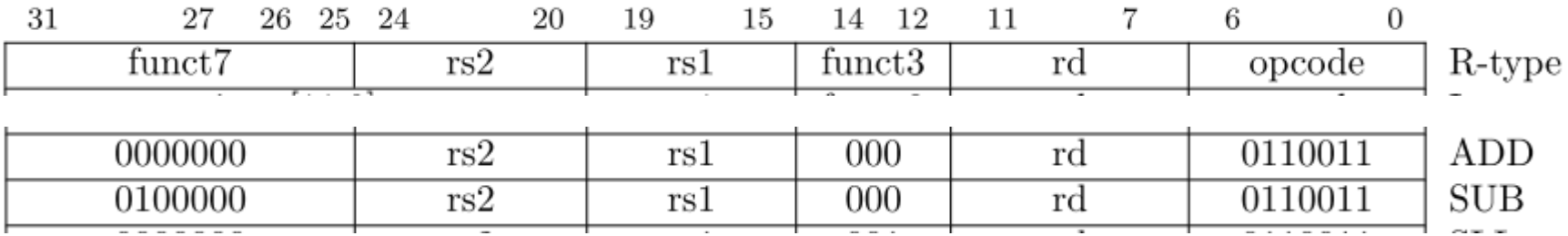
```
add x10, x4, x5     # x10 = (g + h)
add x11, x6, x7     # x11 = (i + j)
sub x3, x10, x11    # f   = x10 - x11
```



31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB

Answer

```
add x10, x4, x5      # x10 = (g + h)
add x11, x6, x7      # x11 = (i + j)
sub x3, x10, x11     # f   = x10 - x11
```



31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB

```
module m_top();
  reg [31:0] r_i1, r_i2, r_i3;
  initial begin
    r_i1 <= {7'b0000000, 5'd5,  5'd4,  3'b000, 5'd10, 7'b0110011};
    r_i2 <= {7'b0000000, 5'd7,  5'd6,  3'b000, 5'd11, 7'b0110011};
    r_i3 <= {7'b0100000, 5'd11, 5'd10, 3'b000, 5'd3,  7'b0110011};
  end
  initial #1 begin
    $display("i1: %b %x", r_i1, r_i1);
    $display("i2: %b %x", r_i2, r_i2);
    $display("i3: %b %x", r_i3, r_i3);
  end
endmodule
```

```
i1: 00000000010100100000010100110011 00520533
i2: 00000000011100110000010110110011 007305b3
i3: 01000000101101010000000110110011 40b501b3
```

code094.v

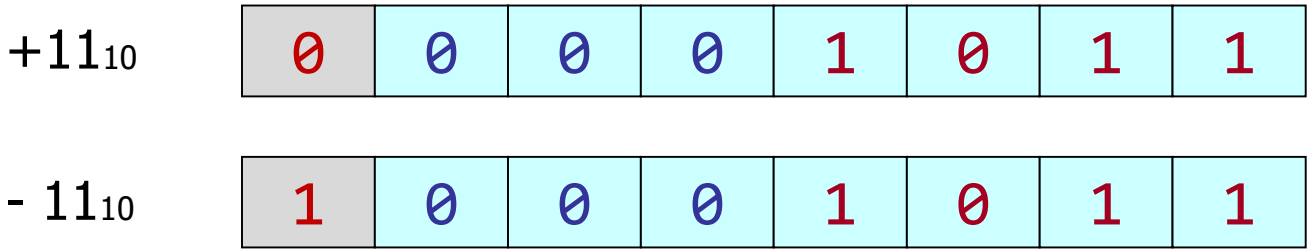
Integer（整数）Representation

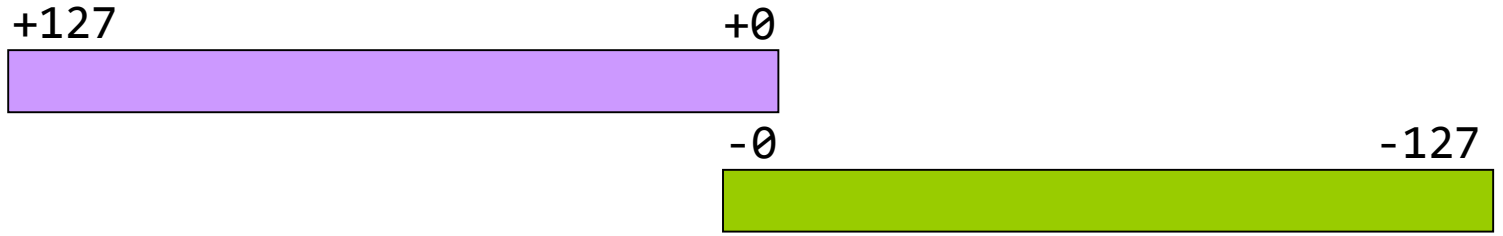
- まずは，「符号なし数」の表現を考える．
  - 欠点は負の数を表現できないこと．
- 整数を2進数の符号なし数で表現するには
  - 例えば，3_{10} であれば，11_2 として下位ビットを決める（右下の小さい数が10の場合には10進数、2の場合には2進数を示す）．
  - 上位の残ったビットを0で埋める．
  - 8ビットであれば，0～255 の256種類の整数を表現できる．

0	0	0	0	0	0	1	1

Integer（整数）Representation

- 符号つき絶対値 (sign and magnitude) による符号付き数の表現
  - 11_{10} であれば，1011_2 として下位ビットを決める．
  - ただし，最上位ビットを用いて符号を表す（符号ビット）．符号ビットが0であれば正数，1であれば負数とする．
  - 残ったビットを0で埋める．
  - 8ビットであれば，- 127 ～127 までの 255種類の整数を表現できる





Integer（整数）Representation

- 2の補数 (two's complement) による符号付き数の表現
  - 正数のビットを反転させ，1を加えたものを負数とする．
  - 8ビットであれば，- 128 ～127 までの 256種類の整数を表現できる

$0000\ 0000_2 = 0_{10}$		
$0000\ 0001_2 = +1_{10}$	$1111\ 1110_2 = -1_{10}$	$1111\ 1111_2 = -1_{10}$
$0000\ 0010_2 = +2_{10}$	$1111\ 1101_2 = -2_{10}$	$1111\ 1110_2 = -2_{10}$
…	…	…
$0111\ 1101_2 = +125_{10}$	$1000\ 0010_2 = -125_{10}$	$1000\ 0011_2 = -125_{10}$
$0111\ 1110_2 = +126_{10}$	$1000\ 0001_2 = -126_{10}$	$1000\ 0010_2 = -126_{10}$
$0111\ 1111_2 = +127_{10}$	$1000\ 0000_2 = -127_{10}$	$1000\ 0001_2 = -127_{10}$
		$1000\ 0000_2 = -128_{10}$
0と1～127の正数	1～127の正数のビット反転（1の補数表現）	ビット反転に1を加えて得る負数

Integer（整数）Representation

- 2の補数 (two's complement) による符号付き数の表現
  - 正数のビットを反転させ，1を加えたものを負数とする．
  - 8ビットであれば，- 128 ～127 までの 256種類の整数を表現できる

$0000\ 0000_2 = 0_{10}$

$0000\ 0001_2 = +1_{10}$

$0000\ 0010_2 = +2_{10}$

…

$0111\ 1101_2 = +125_{10}$

$0111\ 1110_2 = +126_{10}$

$0111\ 1111_2 = +127_{10}$

0と1～127の正数

$1111\ 1111_2 = -1_{10}$

$1111\ 1110_2 = -2_{10}$

…

$1000\ 0011_2 = -125_{10}$

$1000\ 0010_2 = -126_{10}$

$1000\ 0001_2 = -127_{10}$

$1000\ 0000_2 = -128_{10}$

ビット反転に1を加えて得る負数

Integer（整数）Representation

- 2の補数 (two's complement) 表現の特徴
  - 全てのビットを反転させて1を加えると，正負が反転する.
  - 最上位ビットは符号ビットと呼ばれ，0であれば正数，1であれば負数.
  - 符号拡張(sign extension)と呼ばれるビット長を増やす処理は，符号ビットを複製して補填すればよい.

$$x + \overline{x} = -1$$

$$\overline{x} + 1 = -x$$

Example（例題）

- 16ビットの2進数の 2_{10} と -2_{10} を32ビットの2進数に変換せよ．これらは2の補数で表現されている．

Answer

- 16ビットの2進数の 2_{10} と -2_{10} を32ビットの2進数に変換せよ. これらは2の補数で表現されている.

16ビットの2進数の 2_{10} 0000 0000 0000 0010

16ビットの2進数の -2_{10} 1111 1111 1111 1110

32ビットの2進数の 2_{10} 0000 0000 0000 0000 0000 0000 0000 0010

32ビットの2進数の -2_{10} 1111 1111 1111 1111 1111 1111 1111 1110

Verilog HDLで「2の補数」として表示

- ワイヤ型の信号の定義を wire signed とすることで，2の補数表現の符号付き整数として扱われる．

code095.v

```
module m_top();
  reg [15:0] r_data = 16'b1111111111111110;
  wire signed [15:0] w_data = r_data;

  initial #1 begin
    $display("%6d", r_data);
    $display("%6d", w_data);
  end
endmodule
```

```
 65534
    -2
```

Verilog HDLで2の補数表現の符号拡張

- 符号拡張(sign extension)と呼ばれるビット長を増やす処理は，符号ビットを複製して補填すればよい．
- 2の補数表現の16ビットの整数を32ビットの整数に符号拡張する例
  `w_data2 = {{16{w_data1[15]}}, w_data1};`

code096.v

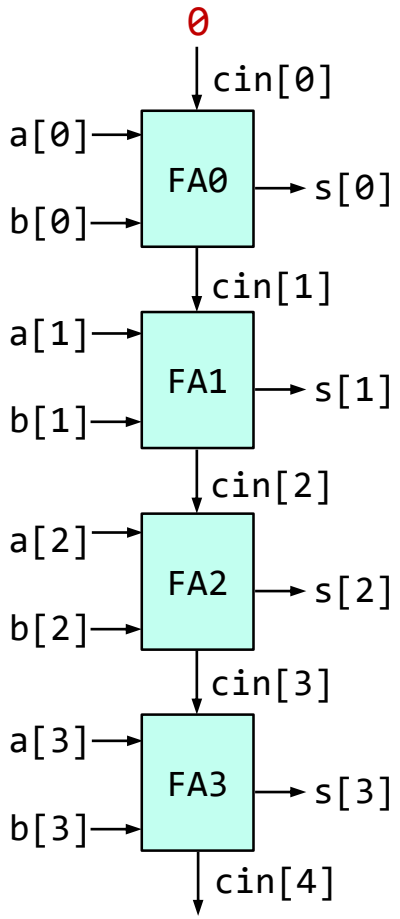
```
module m_top();
  wire signed [15:0] w_data1 = 16'b1111111111111110;
  wire signed [31:0] w_data2 = {{16{w_data1[15]}}, w_data1};

  initial #1 begin
    $display(“%5d %32b", w_data1, w_data1);
    $display("%5d %32b", w_data2, w_data2);
  end
endmodule
```

```
   -2                  1111111111111110
   -2 11111111111111111111111111111110
```

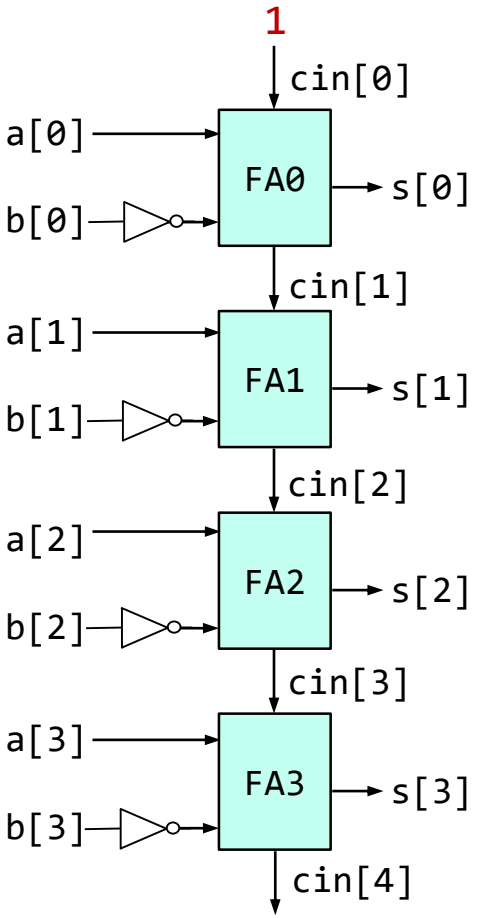
Adder and Subtractor

- 加算器を用いて，減算することでハードウエア量を節約できる
- 4-bit Ripple Carry Adder

```
s = a + b
```

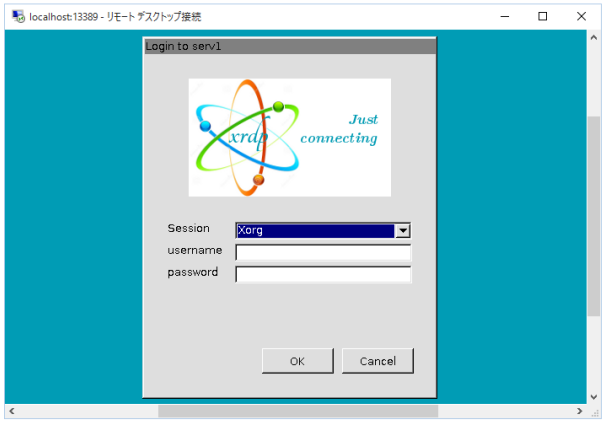


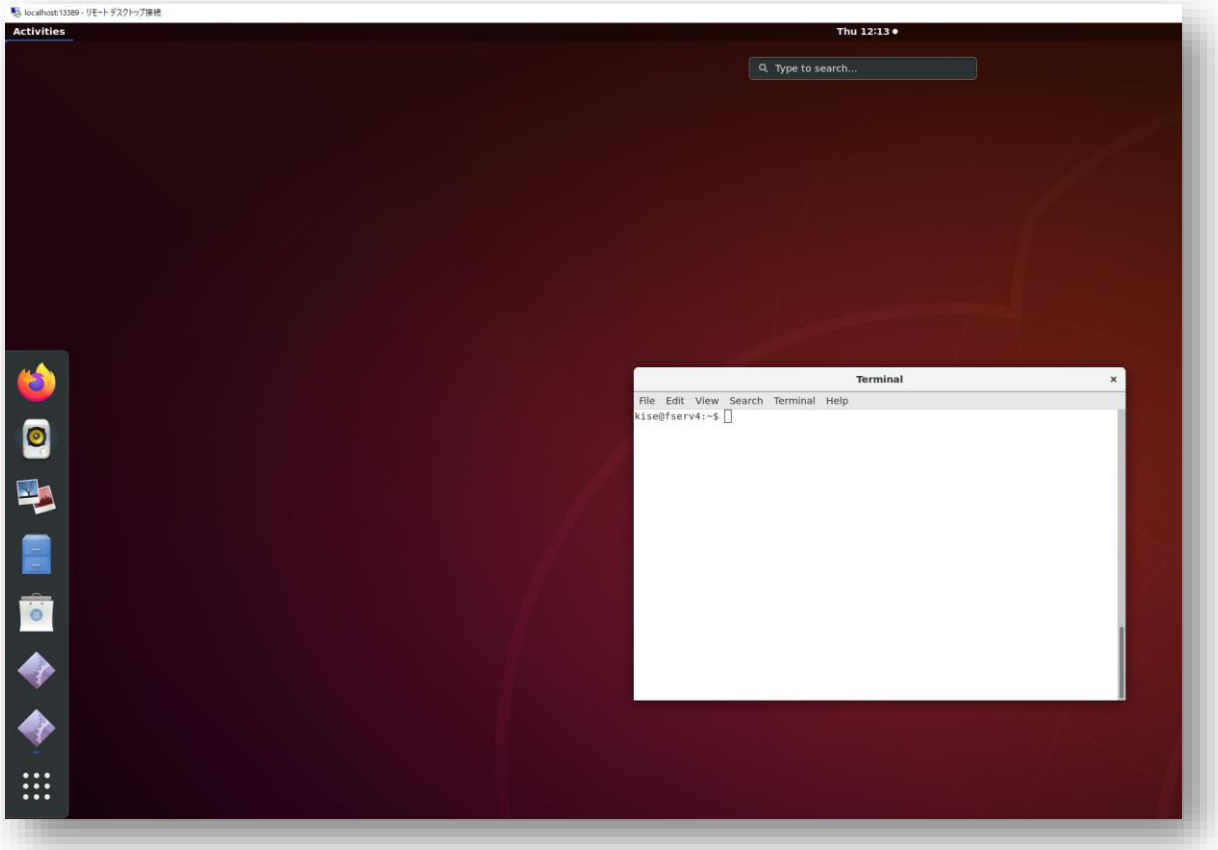

```
s = a - b
  = a + (-b)
  = a + (~b + 1)
```



ACRiルームのデモンストレーション

- LUT で実現される真理値表を確認する方法.
- Clocking Wizard を使って 20MHz のクロック信号を生成する方法.





Department of Computer Science
Course number: CSC.T341

コンピュータ論理設計
Computer Logic Design

7. 命令セットアーキテクチャ：算術論理演算命令
Instruction Set Architecture: Arithmetic and Logic Instructions

吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
講義：月曜日 10:45-12:25, 木曜日 10:45-12:25

Sample Verilog HDL code

- ACRi Room のサーバーにリモートデスクトップでログインする.
  - /home/tu_kise/cld/lec7/ にサンプルのコードがあるので，Ubuntu のターミナルで次のコマンドを入力して，自分のディレクトリにコピーする.
  - /home/tu_kise は automount のディレクトリなので，アクセスしないとファイルが見えない．tabキーによる補完がうまく動作しないことがあるので注意する.

```
$ cd
$ mkdir cld
$ cd cld
$ cp /home/tu_kise/cld/lec7/* .
```

- code102.v をシミュレーションするためには.
  - リモートデスクトップで接続した Ubuntu のターミナルで次のコマンドを入力する.
  - コマンド iverilog でコンパイルして，生成される a.out を実行する.

```
$ iverilog code102.v
$ ./a.out
```

RISC-V の32ビット基本命令セット RV32I

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

Table 24.1: RISC-V base opcode map, inst[1:0]=11

RV32I Base Instruction Set

imm[31:12]		rd	0110111		LUI
imm[31:12]		rd	0010111		AUIPC
		rd	1101111		JAL
		rd	1100111		JALR
		n[4:1|11]	1100011		BEQ
		n[4:1|11]	1100011		BNE
		n[4:1|11]	1100011		BLT
		n[4:1|11]	1100011		BGE
		n[4:1|11]	1100011		BLTU
		n[4:1|11]	1100011		BGEU
		rd	0000011		LB
		rd	0000011		LH
		rd	0000011		LW
		rd	0000011		LBU
		rd	0000011		LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

RISC-V Arithmetic Instructions

- RISC-V assembly language arithmetic statement

```
add   x7, x8, x9
sub   x7, x8, x9
```

- Each arithmetic instruction performs only one operation
- Each arithmetic instruction fits in 32 bits and specifies exactly three operands

```
destination <- source1 op source2
```

- Operand order is fixed (destination first)
- Those operands are all contained in the datapath's register file (x0, ..., x31)

RISC-V Arithmetic Instructions in Verilog HDL

```
reg signed [31:0] x [0:31];  // signed registers
wire [31:0] w_rs1 = x[rs1];  // unsigned wire
wire [31:0] w_rs2 = x[rs2];  // unsigned wire

add : x[rd] <= x[rs1] + x[rs2];         // addition
sub : x[rd] <= x[rs1] - x[rs2];         // subtraction
sll : x[rd] <= x[rs1] << x[rs2[4:0]];   // shift left logical
slt : x[rd] <= (x[rs1] < x[rs2]);       // set less than
sltu: x[rd] <= (w_rs1 < w_rs2);         // set less than unsigned
xor : x[rd] <= x[rs1] ^ x[rs2];         // exclusive-or
srl : x[rd] <= x[rs1] >> x[rs2[4:0]];   // shift right logical
sra : x[rd] <= x[rs1] >>> x[rs2[4:0]]; // shift right arithmetic
or  : x[rd] <= x[rs1] | x[rs2];         // or
and : x[rd] <= x[rs1] & x[rs2];         // and
```

RISC-V の命令フォーマット

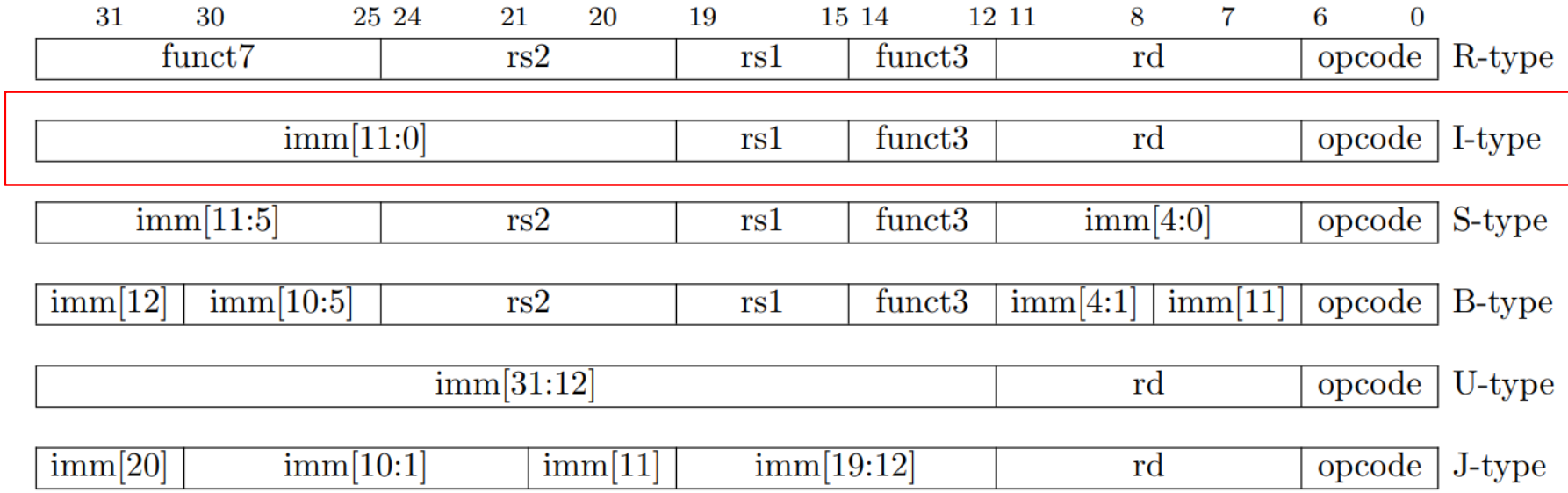


Figure 2.3: RISC-V base instruction formats showing immediate variants.

RISC-V の32ビット基本命令セット RV32I

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

Table 24.1: RISC-V base opcode map, inst[1:0]=11

RV32I Base Instruction Set

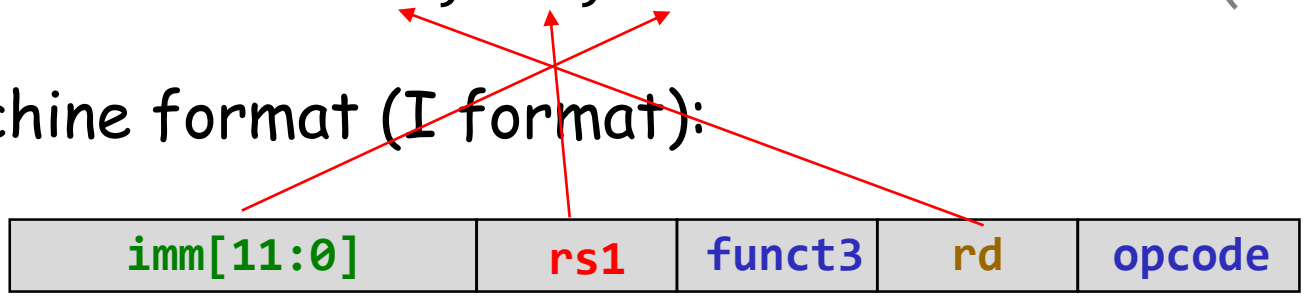
imm[31:12]		rd	0110111		LUI
imm[31:12]		rd	0010111		AUIPC
		rd	1101111		JAL
		rd	1100111		JALR
		n[4:1|11]	1100011		BEQ
		n[4:1|11]	1100011		BNE
		n[4:1|11]	1100011		BLT
		n[4:1|11]	1100011		BGE
		n[4:1|11]	1100011		BLTU
		n[4:1|11]	1100011		BGEU
		rd	0000011		LB
		rd	0000011		LH
		rd	0000011		LW
		rd	0000011		LBU
		rd	0000011		LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011		SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011		SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011		SW
imm[11:0]		rs1	000	rd	0010011		ADDI
imm[11:0]		rs1	010	rd	0010011		SLTI
imm[11:0]		rs1	011	rd	0010011		SLTIU
imm[11:0]		rs1	100	rd	0010011		XORI
imm[11:0]		rs1	110	rd	0010011		ORI
imm[11:0]		rs1	111	rd	0010011		ANDI
0000000	shamt	rs1	001	rd	0010011		SLLI
0000000	shamt	rs1	101	rd	0010011		SRLI
0100000	shamt	rs1	101	rd	0010011		SRAI
0000000	rs2	rs1	000	rd	0110011		ADD
0100000	rs2	rs1	000	rd	0110011		SUB
0000000	rs2	rs1	001	rd	0110011		SLL
0000000	rs2	rs1	010	rd	0110011		SLT
0000000	rs2	rs1	011	rd	0110011		SLTU
0000000	rs2	rs1	100	rd	0110011		XOR
0000000	rs2	rs1	101	rd	0110011		SRL
0100000	rs2	rs1	101	rd	0110011		SRA
0000000	rs2	rs1	110	rd	0110011		OR
0000000	rs2	rs1	111	rd	0110011		AND
fm | pred | succ		rs1	000	rd	0001111		FENCE
000000000000		00000	000	00000	1110011		ECALL
000000000001		00000	000	00000	1110011		EBREAK

RISC-V Immediate Instructions

- Small constants are used often in typical code
- Possible approaches?
  - put "typical constants" in memory and load them
  - create hard-wired registers (like x0) for constants like 1
  - have special instructions that contain constants !

```
addi  x7, x8, -2   # x7 = x8 + (-2)
```

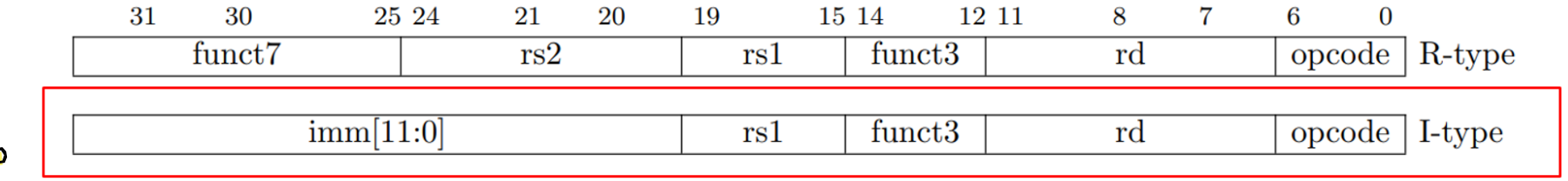
- Machine format (I format):



- The constant is kept inside the instruction itself
  - Immediate format limits values to the range $+2^{11}-1$ to -2^{11}

RISC-V Instructions with Immediate in Verilog HDL

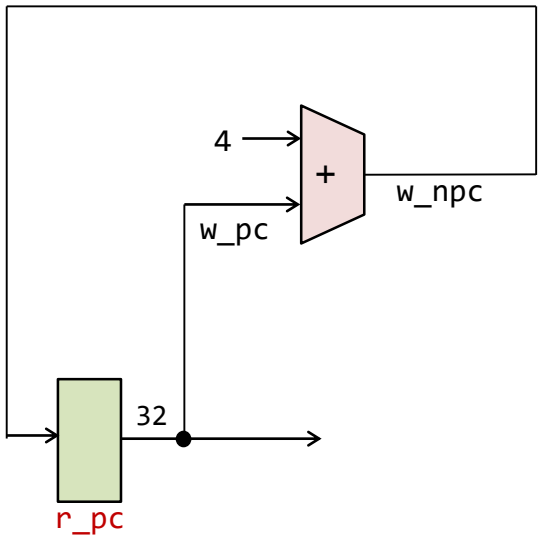
```
reg signed [31:0] x [0:31];  // signed registers
wire       [31:0] w_ir;      // instruction
wire signed [31:0] w_sext_imm   = {{20{w_ir[31]}}, w_ir[31:20]};
wire       [31:0] w_sext_imm_u = {{20{w_ir[31]}}, w_ir[31:20]};

addi : x[rd] <= x[rs1] + w_sext_imm;     // add immediate
slti : x[rd] <= (x[rs1] < w_sext_imm);   // set less than immediate
sltiu: x[rd] <= (x[rs1] < w_sext_imm_u); // slt immediate, unsigned
xori : x[rd] <= x[rs1] ^ w_sext_imm;     // exclusive-or immediate
ori  : x[rd] <= x[rs1] | w_sext_imm;     // or immediate
andi : x[rd] <= x[rs1] & w_sext_imm;     // and immediate
```



プロセッサが命令を処理するための基本的な5つのステップ

- IF (Instruction Fetch)
  メモリから命令をフェッチする.
- ID (Instruction Decode)
  命令をデコード(解読)しながら, レジスタの値を読み出す(Operand Fetch)
- EX (Execution)
  命令操作の実行またはアドレスの生成を行う.
- MEM (Memory Access)
  必要であれば, メモリ(データ・メモリ)のオペランドにアクセスする.
- WB (Write Back)
  必要であれば, 結果をレジスタに書き込む.

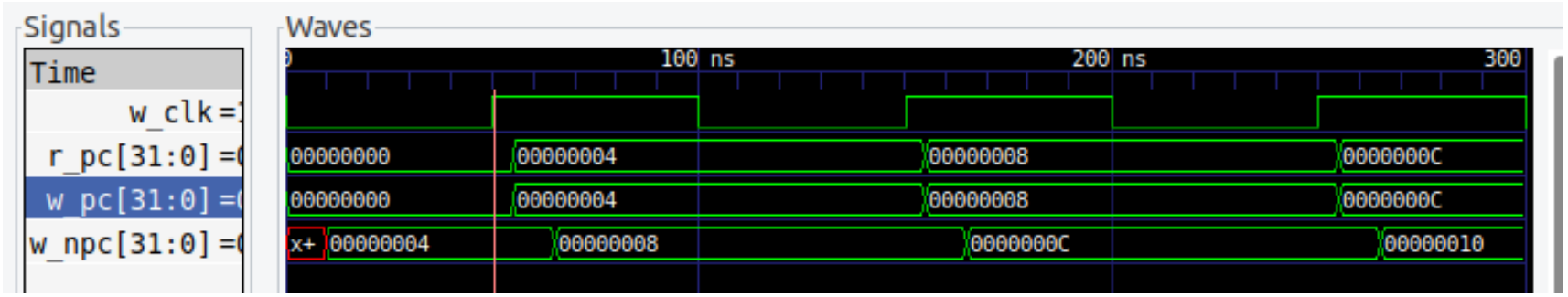
m_proc01 プロセッサの設計と実装に向けた一歩



code102.v

```
module m_top ();
  reg r_clk=0; initial forever #50 r_clk = ~r_clk;
  wire [31:0] w_pc;
  m_main m_main0 (r_clk, w_pc);
  always@(*) #1 $write("%3d %x¥n", $time, w_pc);
  initial $dumpfile("main.vcd"); /* file name for GTKWave */
  initial $dumpvars(0, m_top);   /* module    for GTKWave */
  initial #300 $finish();
endmodule

module m_main (w_clk, w_pc);
  input  wire w_clk;
  output wire [31:0] w_pc;

  reg [31:0] r_pc = 0;
  assign w_pc = r_pc;
  wire [31:0] #10 w_npc = w_pc + 4;
  always@(posedge w_clk) #5 r_pc <= w_npc;
endmodule
```

```
  1 00000000
 56 00000004
156 00000008
256 0000000c
```



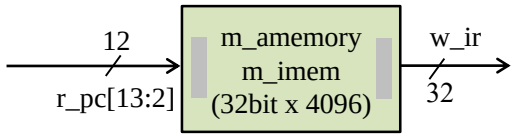
m_amemory 非同期式メモリの記述とシミュレーション

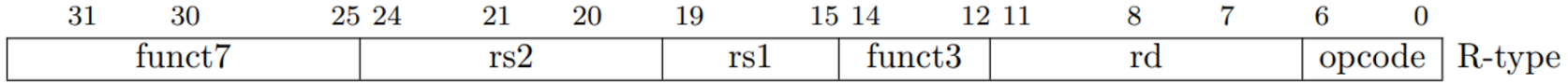
- Verilog HDLでは，ビット幅Bでワード数Wのメモリ m を reg [B-1:0] m [0:W-1] として宣言できる．
- 読み出す動作でクロック信号を利用しないメモリを非同期メモリ (asynchronous memory) と呼ぶ．
- 非同期式メモリの記述例を示す．シミュレーションでの読み出しの遅延を 20nsec とした．w_addr で指定されたアドレスの内容を読み出す．posedge w_clk のタイミングで，w_we (write enable) が1の時に，w_addr で指定されたアドレスに w_din (data in) の値を書き込む．
- このコードをシミュレーションして，波形を確認すること．

```
module m_amemory (w_clk, w_addr, w_we, w_din, w_dout);
  input  wire w_clk, w_we;
  input  wire [11:0] w_addr;
  input  wire [31:0] w_din;
  output wire [31:0] w_dout;

  reg [31:0] cm_ram [0:4095]; // 4K word (4096 x 32bit) memory
  always @(posedge w_clk) if (w_we) cm_ram[w_addr] <= w_din;
  assign #20 w_dout = cm_ram[w_addr];

  initial begin
    cm_ram[0]={7'd0, 5'd0, 5'd0, 3'd0, 5'd0, 7'b0110011}; // add x0, x0, x0
    cm_ram[1]={7'd0, 5'd1, 5'd0, 3'd0, 5'd4, 7'b0110011}; // add x4, x0, x1
    cm_ram[2]={7'd0, 5'd2, 5'd1, 3'd0, 5'd5, 7'b0110011}; // add x5, x1, x2
    cm_ram[3]={7'd0, 5'd5, 5'd4, 3'd0, 5'd6, 7'b0110011}; // add x6, x4, x5
    cm_ram[4]={7'd0, 5'd0, 5'd0, 3'd0, 5'd0, 7'b0110011}; // add x0, x0, x0
  end
endmodule
```

```
module m_top ();
  reg r_clk=0; initial forever #50 r_clk = ~r_clk;
  reg [31:0] r_pc = 0;
  always @(posedge r_clk) r_pc <= #3 r_pc + 4;

  wire [31:0] w_data;
  m_amemory m (r_clk, r_pc[13:2], 1'd0, 32'd0, w_data);

  initial $dumpfile("main.vcd"); /* file name for GTKWave */
  initial $dumpvars(0, m_top);   /* module    for GTKWave */
  initial #1000 $finish;
  always@(*) #80 $write("%3d %d %x¥n", $time, r_pc, w_data);
endmodule
```

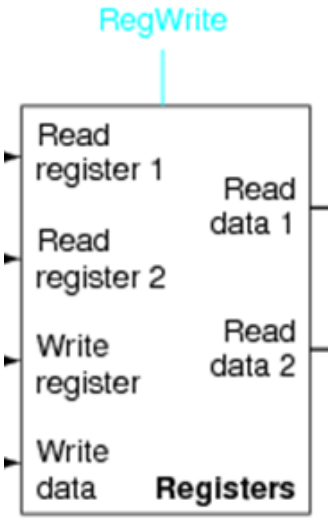
code111.v





Register file, レジスタファイル m_regfile の実装

- Verilog HDLでは，ビット幅Bでワード数Wのメモリ m を reg [B-1:0] m [0:W-1] として宣言できる．
- w_rr1 で指定したレジスタの値を読み出し w_rdata1 に出力する．非同期の読み出し．
- w_rr2 で指定したレジスタの値を読み出し w_rdata2 に出力する．非同期の読み出し．
  - ただし，x0 (zero) の読み出しは，値0を出力する．
- posedge w_clk のタイミングで，w_we (write enable) が1の時に，w_wr (write register) で指定されたレジスタに w_wdata (write data) の値を書き込む．
- このモジュールではadd命令の動作確認のために x1 を 1 で，x2 を 2 で初期化している．

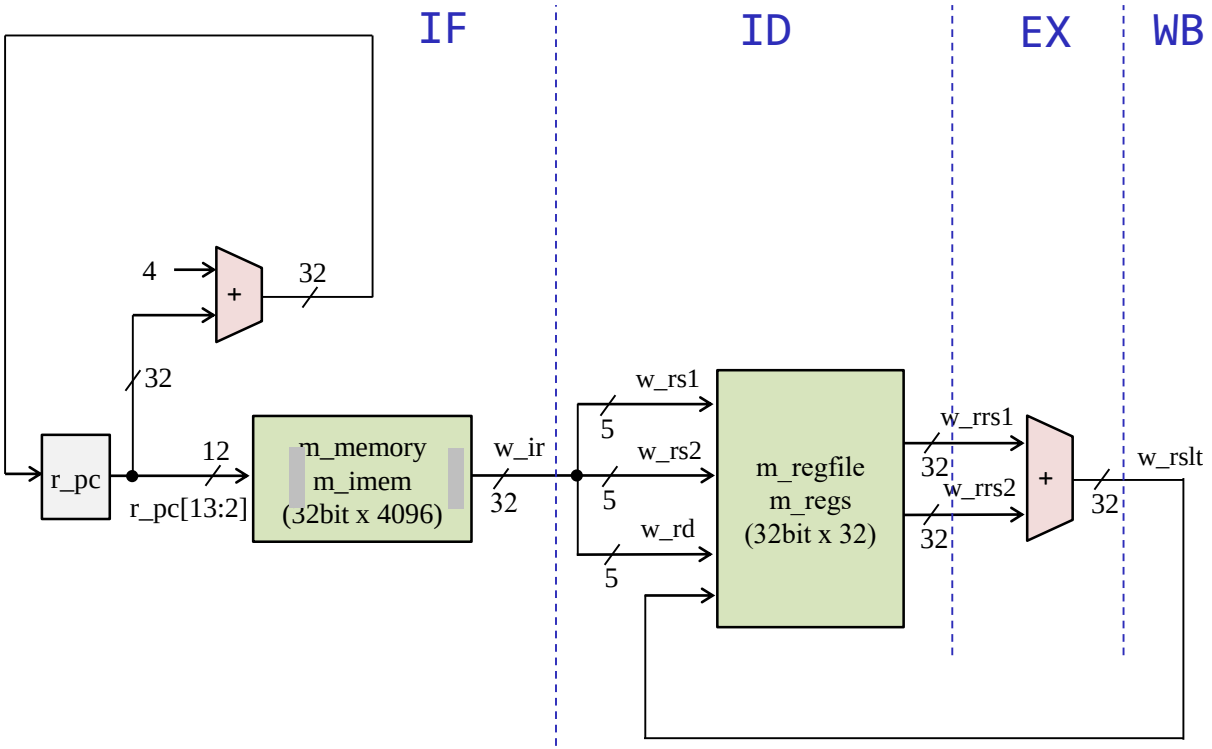


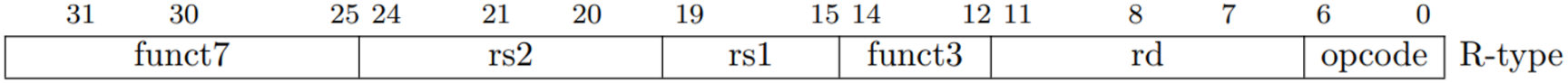
code112.v

```
module m_regfile (w_clk, w_rr1, w_rr2, w_wr, w_we, w_wdata, w_rdata1, w_rdata2);
  input  wire        w_clk;
  input  wire [4:0]  w_rr1, w_rr2, w_wr;
  input  wire [31:0] w_wdata;
  input  wire        w_we;
  output wire [31:0] w_rdata1, w_rdata2;

  reg [31:0] r[0:31];
  assign w_rdata1 = (w_rr1==0) ? 0 : r[w_rr1];
  assign w_rdata2 = (w_rr2==0) ? 0 : r[w_rr2];
  always @(posedge w_clk) if(w_we) r[w_wr] <= w_wdata;

  initial r[1] = 1;
  initial r[2] = 2;
endmodule
```

m_proc02 addを処理するシングルサイクルのプロセッサ

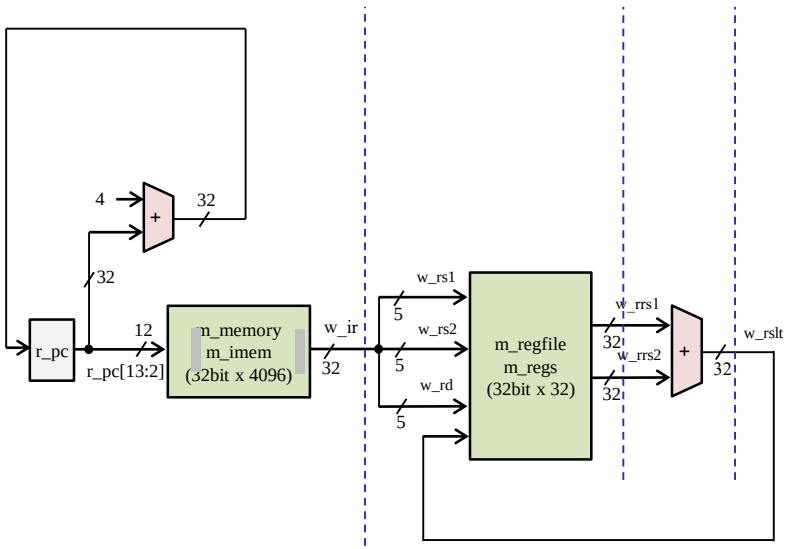
- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), ライトバック(WB) の処理をおこなう加算命令 (add) のみに対応したプロセッサのブロック図





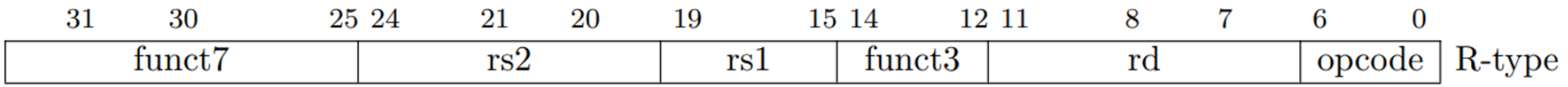
m_proc02 addを処理するシングルサイクルのプロセッサ

- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), ライトバック(WB) の処理をおこなう加算命令 (add) のみに対応したプロセッサのブロック図
- このプロセッサで，code111.v の命令列を実行するときの配線の値を考える.



code113.v の一部

```
module m_proc02 (w_clk, w_ce, w_led);
  input  wire w_clk, w_ce;
  output wire [31:0] w_led;

  reg [31:0] r_pc = 0;
  always @(posedge w_clk) #5 if(w_ce & r_pc[13:2]!=4) r_pc <= r_pc + 4;
  wire [31:0] w_ir;
  m_amemory m_imem (w_clk, r_pc[13:2], 1'd0, 32'd0, w_ir);

  wire [4:0]  w_rs1 = w_ir[19:15];
  wire [4:0]  w_rs2 = w_ir[24:20];
  wire [4:0]  w_rd  = w_ir[11:7];
  wire [31:0] w_rrs1, w_rrs2, w_rslt;
  m_regfile m_regs (w_clk, w_rs1, w_rs2, w_rd, w_ce, w_rslt, w_rrs1, w_rrs2);

  assign #10 w_rslt = w_rrs1 + w_rrs2;

  reg [31:0] r_led = 0;
  always @(posedge w_clk) if(w_ce & w_rd==6) r_led <= w_rslt;
  assign w_led = r_led;
endmodule
```

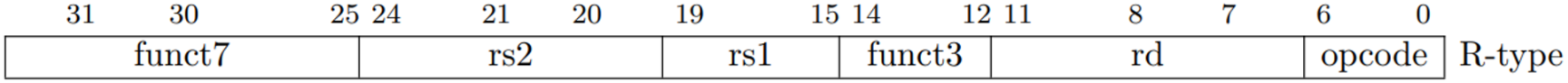


m_proc02 addを処理するシングルサイクルのプロセッサ

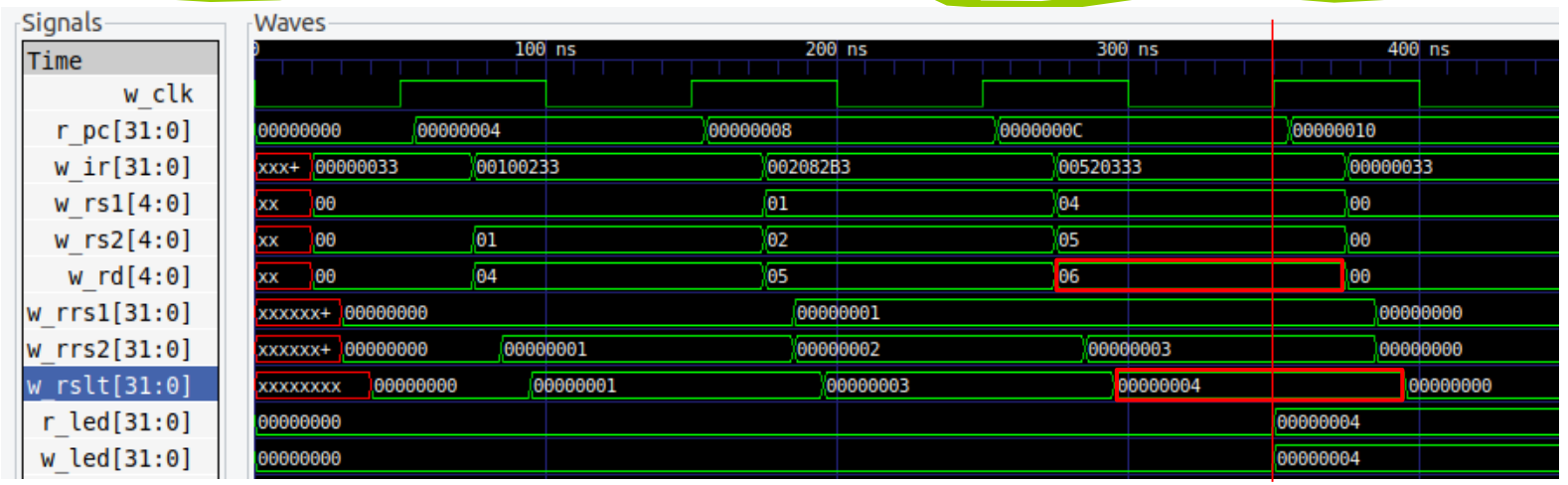
- code113.v をシミュレーションして，その波形を確認すること.
- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), ライトバック(WB) の処理をおこなう加算命令 (add) のみに対応したプロセッサ
- m_proc02 のインスタンス名を p とする. p の内部の r_pc は，ピリオドを用いて p.r_pc として参照できる.
- 同様に，p に含まれるインスタンス m_reg の内部の r[1] は， p.m_reg.r[1] として参照できる。

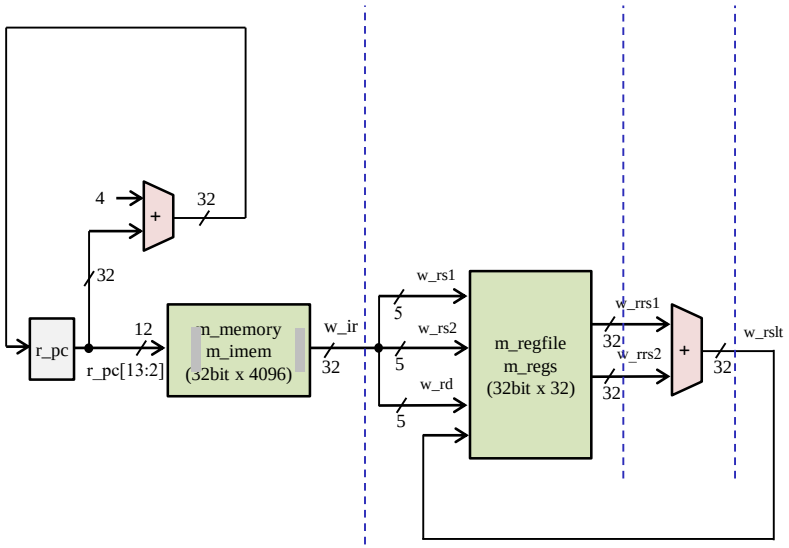
code113.v の一部

```
module m_top ();
  reg r_clk=0; initial forever #50 r_clk = ~r_clk;
  wire [31:0] w_led;
  m_proc02 p (r_clk, 1'b1, w_led);
  initial $dumpfile("main.vcd"); /* file name for GTKWave */
  initial $dumpvars(0, m_top);   /* module    for GTKWave */
  initial #550 $finish;
  always@(posedge r_clk) #1 $write("%4d %x: %x  %x -> %x¥n",
                                   $time, p.r_pc, p.w_rrs1, p.w_rrs2, p.w_rslt);
endmodule
```



m_proc02 addを処理するシングルサイクルのプロセッサ





code113.v の一部

```
module m_proc02 (w_clk, w_ce, w_led);
  input  wire w_clk, w_ce;
  output wire [31:0] w_led;

  reg [31:0] r_pc = 0;
  always @(posedge w_clk) #5 if(w_ce & r_pc[13:2]!=4) r_pc <= r_pc + 4;
  wire [31:0] w_ir;
  m_amemory m_imem (w_clk, r_pc[13:2], 1'd0, 32'd0, w_ir);

  wire [4:0]  w_rs1 = w_ir[19:15];
  wire [4:0]  w_rs2 = w_ir[24:20];
  wire [4:0]  w_rd  = w_ir[11:7];
  wire [31:0] w_rrs1, w_rrs2, w_rslt;
  m_regfile m_regs (w_clk, w_rs1, w_rs2, w_rd, w_ce, w_rslt, w_rrs1, w_rrs2);

  assign #10 w_rslt = w_rrs1 + w_rrs2;

  reg [31:0] r_led = 0;
  always @(posedge w_clk) if(w_ce & w_rd==6) r_led <= w_rslt;
  assign w_led = r_led;
endmodule
```

w_led の出力の値はどうなるか？

m_proc02 addを処理するシングルサイクルのプロセッサ

- FPGA で動作させるためのコード code115.v な内容を理解すること.
- 50MHz のクロック信号を生成するように clk_wiz_0 を生成する.
- 32ビットの入力を持つように vio_0 を生成する.
- FPGA で動作させたときの VIO の値はどうなるか？

```
module m_main (w_clk, w_led);
  input  wire w_clk;
  output wire [3:0] w_led;

  wire [31:0] w_dout;
  wire w_clk2, w_locked;
  clk_wiz_0 clk_w0 (w_clk2, 0, w_locked, w_clk);
  vio_0 vio_00(w_clk2, w_dout);

  m_proc02 p (w_clk2, w_locked, w_dout);

  reg [3:0] r_led = 0;
  always @(posedge w_clk2)
    r_led <= {^w_dout[31:24], ^w_dout[23:16], ^w_dout[15:8], ^w_dout[7:0]};
  assign w_led = r_led;
endmodule
```

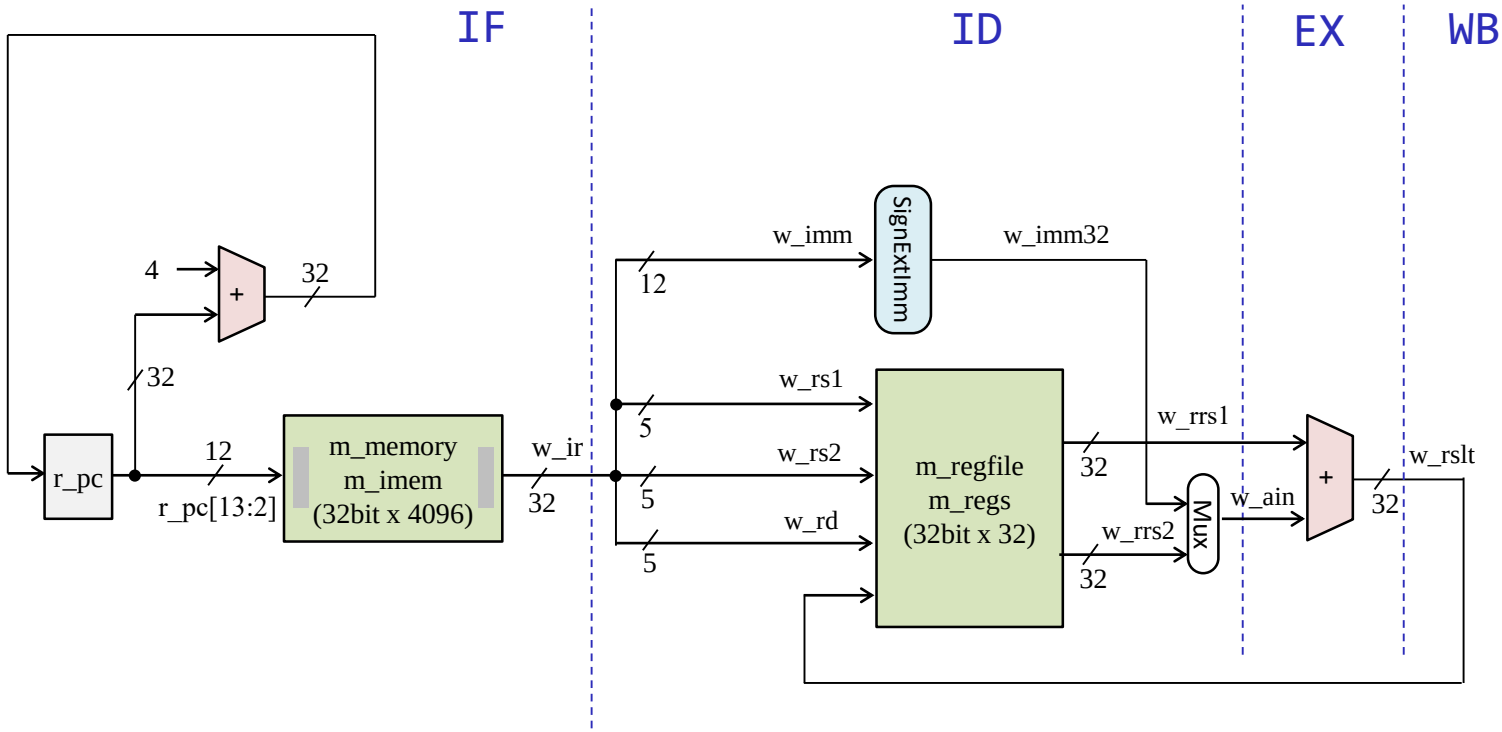
```
module m_proc02 (w_clk, w_ce, w_led);
  input  wire w_clk, w_ce;
  output wire [31:0] w_led;

  reg [31:0] r_pc = 0;
  always @(posedge w_clk) #5 if(w_ce & r_pc[13:2]!=4) r_pc <= r_pc + 4;
  wire [31:0] w_ir;
  m_amemory m_imem (w_clk, r_pc[13:2], 1'd0, 32'd0, w_ir);

  wire [4:0]  w_rs1 = w_ir[19:15];
  wire [4:0]  w_rs2 = w_ir[24:20];
  wire [4:0]  w_rd  = w_ir[11:7];
  wire [31:0] w_rrs1, w_rrs2, w_rslt;
  m_regfile m_regs (w_clk, w_rs1, w_rs2, w_rd, w_ce, w_rslt, w_rrs1, w_rrs2);

  assign #10 w_rslt = w_rrs1 + w_rrs2;

  reg [31:0] r_led = 0;
  always @(posedge w_clk) if(w_ce & w_rd==6) r_led <= w_rslt;
  assign w_led = r_led;
endmodule
```

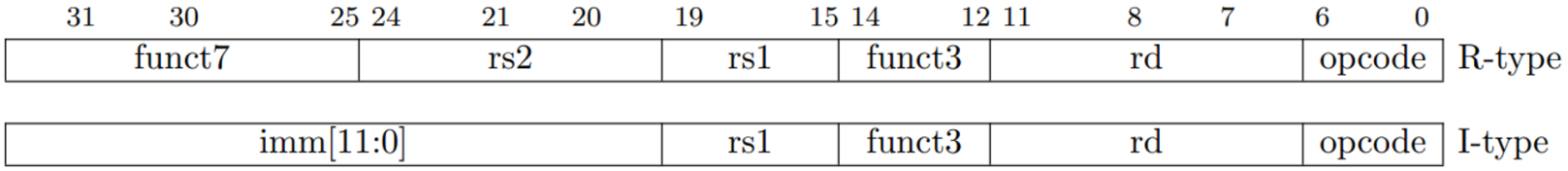
w_led の出力の値はどうなるか？

code115.v の一部

m_proc03 add と addi を処理するプロセッサ

- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), ライトバック(WB) の処理をおこなう加算命令(add, addi)に対応したプロセッサのブロック図





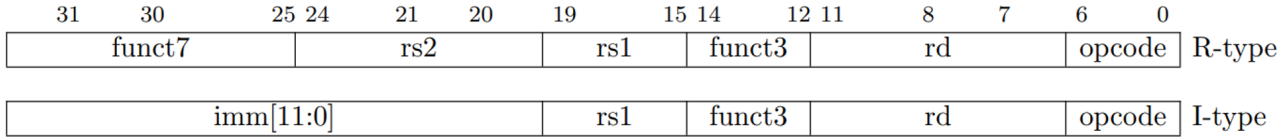
m_proc03 add と addi を処理するプロセッサ

- code150.v を修正すること.
- 波形を確認すること.

```
module m_proc03 (w_clk, w_ce, w_led);
  input  wire w_clk, w_ce;
  output wire [31:0] w_led;

  reg [31:0] r_pc = 0;
  always @(posedge w_clk) #5 if(w_ce & r_pc[13:2]!=4) r_pc <= r_pc + 4;
  wire [31:0] w_ir;
  m_amemory m_imem (w_clk, r_pc[13:2], 1'd0, 32'd0, w_ir);

  wire [4:0]  w_rs1 = w_ir[19:15];
  wire [4:0]  w_rs2 = w_ir[24:20];
  wire [4:0]  w_rd  = w_ir[11:7];
  wire [31:0] w_rrs1, w_rrs2, w_rslt;
  m_regfile m_regs (w_clk, w_rs1, w_rs2, w_rd, w_ce, w_rslt, w_rrs1, w_rrs2);

  /***** Please describe this part by yourself *****/


  reg [31:0] r_led = 0;
  always @(posedge w_clk) if(w_ce & w_rd==6) r_led <= w_rslt;
  assign w_led = r_led;
endmodule
```

```
module m_amemory (w_clk, w_addr, w_we, w_din, w_dout);
  input  wire w_clk, w_we;
  input  wire [11:0] w_addr;
  input  wire [31:0] w_din;
  output wire [31:0] w_dout;

  reg [31:0] cm_ram [0:4095]; // 4K word (4096 x 32bit) memory
  always @(posedge w_clk) if (w_we) cm_ram[w_addr] <= w_din;
  assign #20 w_dout = cm_ram[w_addr];

  initial begin
    cm_ram[0]={7'd0, 5'd0, 5'd0, 3'd0, 5'd0, 7'b0110011}; // add  x0, x0, x0
    cm_ram[1]={12'h008,    5'd0, 3'd0, 5'd4, 7'b0010011}; // addi x4, x0,  8
    cm_ram[2]={12'hffe,    5'd0, 3'd0, 5'd5, 7'b0010011}; // addi x5, x0, -2
    cm_ram[3]={7'd0, 5'd5, 5'd4, 3'd0, 5'd6, 7'b0110011}; // add  x6, x4, x5
    cm_ram[4]={7'd0, 5'd0, 5'd0, 3'd0, 5'd0, 7'b0110011}; // add  x0, x0, x0
  end
endmodule
```

code150.v の一部



Department of Computer Science
Course number: CSC.T341

コンピュータ論理設計
Computer Logic Design

8. 命令セットアーキテクチャ：ロードストア命令と分岐命令
Instruction Set Architecture: Load/Store and Branch Instructions

吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
講義：月曜日 10:45-12:25, 木曜日 10:45-12:25

Machine Language - Add Instruction

- Instructions, like registers and words of data, are 32 bits long
- Arithmetic Instruction Format (R-type):

```
add  x7, x8, x9
```

R-type

opcode 7-bits *opcode* that specifies the operation

rs1 5-bits register file address of the first source operand

rs2 5-bits register file address of the second source operand

rd 5-bits register file address of the result's destination

funct3 and **funct7** 10-bits select the type of operation (function)

RISC-V Immediate Instructions

- Small constants are used often in typical code
- Possible approaches?
  - put "typical constants" in memory and load them
  - create hard-wired registers (like x0) for constants like 1
  - have special instructions that contain constants !

```
addi  x7, x8, -2   # x7 = x8 + (-2)
```

- Machine format (I format):

I-type

- The constant is kept inside the instruction itself
  - Immediate format limits values to the range $+2^{11}-1$ to -2^{11}

Two major ISA types: RISC vs CISC

- RISC (Reduced Instruction Set Computer) philosophy
  - fixed instruction lengths
  - load-store instruction sets
  - limited addressing modes
  - limited operations
  - RISC: MIPS, Alpha, ARM, RISC-V, …
- CISC (Complex Instruction Set Computer) philosophy
  - ! fixed instruction lengths
  - ! load-store instruction sets
  - ! limited addressing modes
  - ! limited operations
  - CISC : DEC VAX11, Intel 80x86, …

RISC-V の32ビット基本命令セット RV32I

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

Table 24.1: RISC-V base opcode map, inst[1:0]=11

RISC-V Memory Access Instructions

- RISC-V has two basic data transfer instructions for accessing memory

```
lw  x5, 24(x7)  # load word from memory
sw  x3, 28(x9)  # store word to memory
```

- The data is loaded into (lw) or stored from (sw) a register in the register file
- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value

Machine Language - Load Instruction

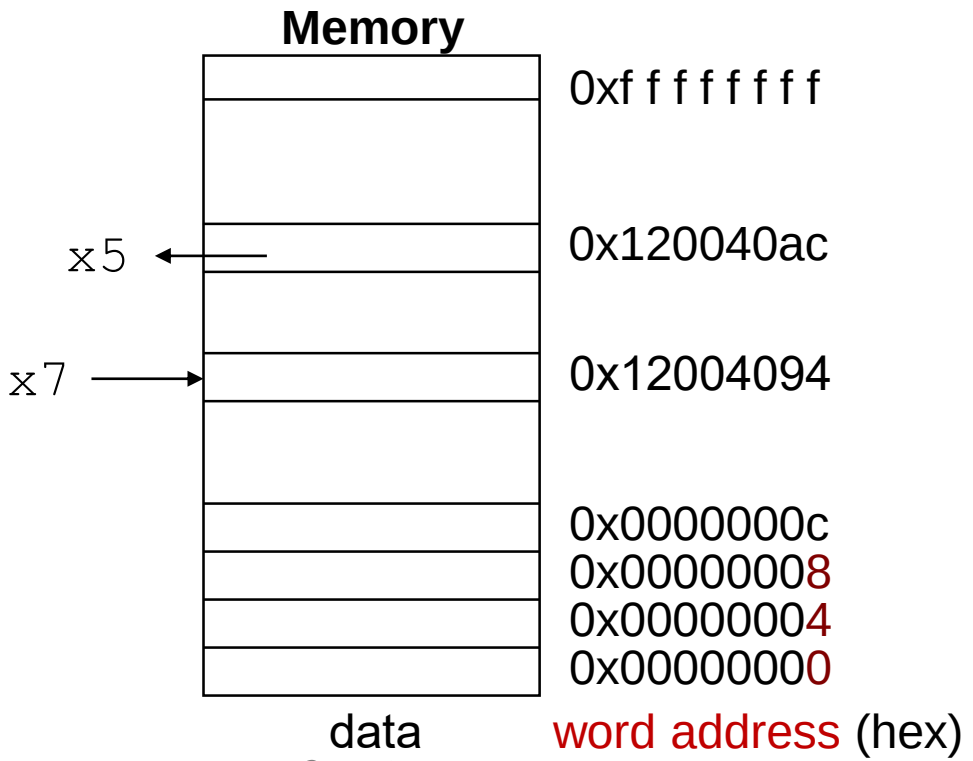
- Load Instruction Format (I-type):

```
lw x5, 24(x7)
```

imm[11:0]	rs1	funct3	rd	opcode	I-type

24_{10} + x7 =

. . . 0001 1000
+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac



Example（例題）

- g = h + A[8]

  100語から成る配列Aがあるとする．また，コンパイラは変数g, h にレジスタ x5, x6 を割り付ける．さらに配列の開始アドレスは x7 に納められているとする．
  上のステートメントをコンパイルせよ．

Answer

- g = h + A[8]

100語から成る配列Aがあるとする. また, コンパイラは変数g, h にレジスタ x5, x6 を割り付ける. さらに配列の開始アドレスは x7 に納められているとする.
上のステートメントをコンパイルせよ.

```
lw  x9, 32(x7)    # x9 = A[8]
add x5, x6, x9    # g = h + x9
```

Machine Language - Store Instruction

- Load Instruction Format (S-type):

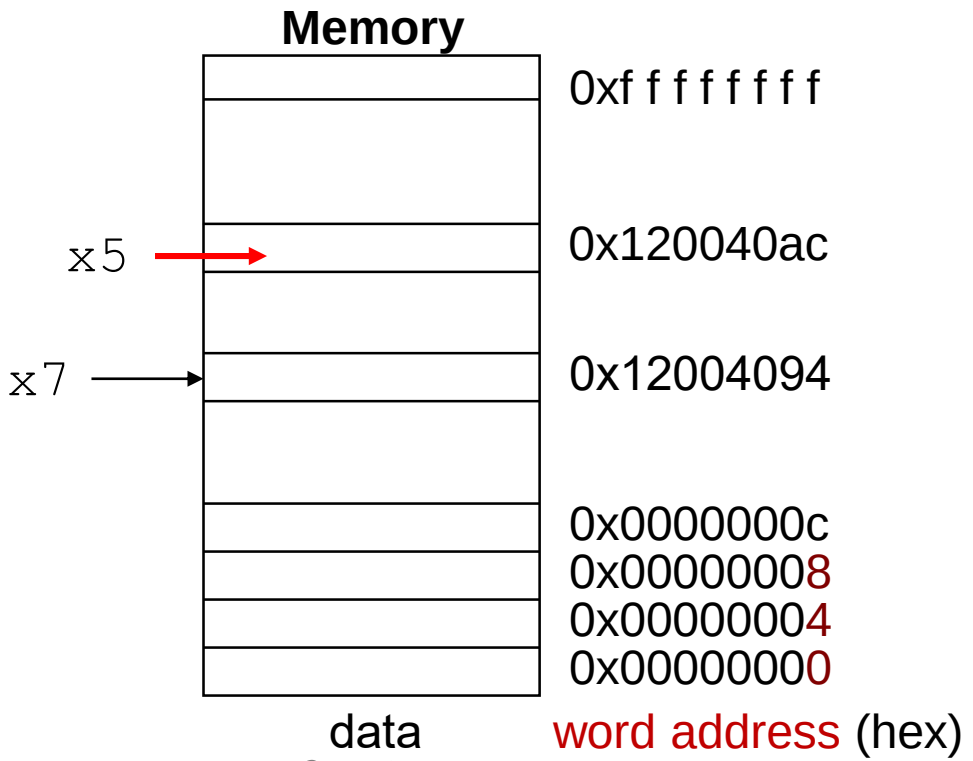
```
sw x5, 24(x7)
```

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode

S-type

24_{10} + x7 =

. . . 0001 1000
+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac



Example（例題）

- A[12] = h + A[8]

  100語から成る配列Aがあるとする．また，コンパイラは変数 h にレジスタ x6 を割り付ける．さらに配列の開始アドレスは x7 に納められているとする．
  上のステートメントをコンパイルせよ．

Answer

- A[12] = h + A[8]

  100語から成る配列Aがあるとする. また, コンパイラは変数 h にレジスタ x6 を割り付ける. さらに配列の開始アドレスは x7 に納められているとする.
  上のステートメントをコンパイルせよ.

```
lw  x9, 32(x7)    # x9 = A[8]
add x9, x6, x9    # x9 = h + x9
sw  x9, 48(x7)    # A[12] = x9
```

RISC-V の32ビット基本命令セット RV32I の分岐命令

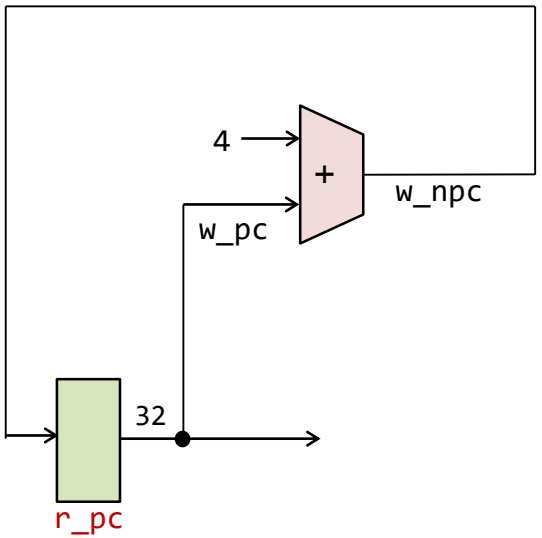
RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm	pred | succ	rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

Table 24.1: RISC-V base opcode map, inst[1:0]=11

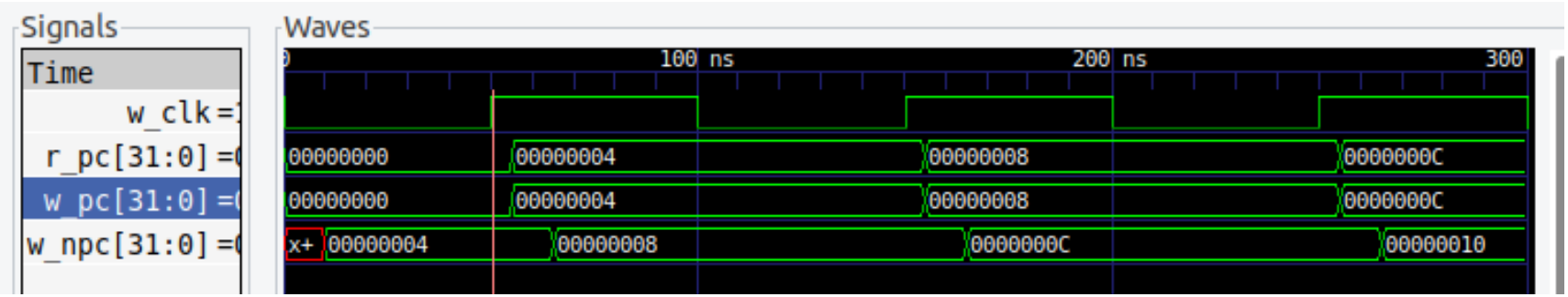
m_proc01 プロセッサの設計と実装に向けた一歩



code102.v

```
module m_top ();
  reg r_clk=0; initial forever #50 r_clk = ~r_clk;
  wire [31:0] w_pc;
  m_main m_main0 (r_clk, w_pc);
  always@(*) #1 $write("%3d %x¥n", $time, w_pc);
  initial $dumpfile("main.vcd"); /* file name for GTKWave */
  initial $dumpvars(0, m_top);   /* module    for GTKWave */
  initial #300 $finish();
endmodule

module m_main (w_clk, w_pc);
  input  wire w_clk;
  output wire [31:0] w_pc;

  reg [31:0] r_pc = 0;
  assign w_pc = r_pc;
  wire [31:0] #10 w_npc = w_pc + 4;
  always@(posedge w_clk) #5 r_pc <= w_npc;
endmodule
```

```
  1 00000000
 56 00000004
156 00000008
256 0000000c
```



RISC-V Control Flow Instructions

- RISC-V conditional branch instructions:

```
beq x4, x5, Lbl  # go to Lbl if x4==x5
bne x4, x5, Lbl  # go to Lbl if x4!=x5
```

```
Ex:   if (i==j) h = i + j;
```

```
      bne x4, x5, Lbl1   # if (i!=j) goto Lbl1
      add x6, x4, x5     # h = i + j;
Lbl1: ...
```

- Instruction Format (B-type):

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type

- How is the branch destination address specified?

RISC-V Control Flow Instructions

- B形式のimmを適切に並べ替えて連結することで得られる12ビットのimm[12:1]の下位に1ビットの0を連結して，13ビットの即値を得る．
- これを符号拡張することで32ビットの即値に変換する．
- このように得られた32ビットの即値にPCの値を加算することで，分岐先のアドレスを得る．

imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

```
beq  (branch if equal)
bne  (branch if not equal)
blt  (branch if less than)
bge  (branch if greater than or equal)
bltu (branch if less than, unsigned)
bgeu (branch if greater than or equal, unsigned)
```

RISC-V の命令長

The RISC-V Instruction Set Manual
Volume I: Unprivileged ISA
Document Version 20191214-*draft*

Editors: Andrew Waterman[1], Krste Asanović[1,2]
[1]SiFive Inc.,
[2]CS Division, EECS Department, University of California, Berkeley
andrew@sifive.com, krste@berkeley.edu
November 12, 2021

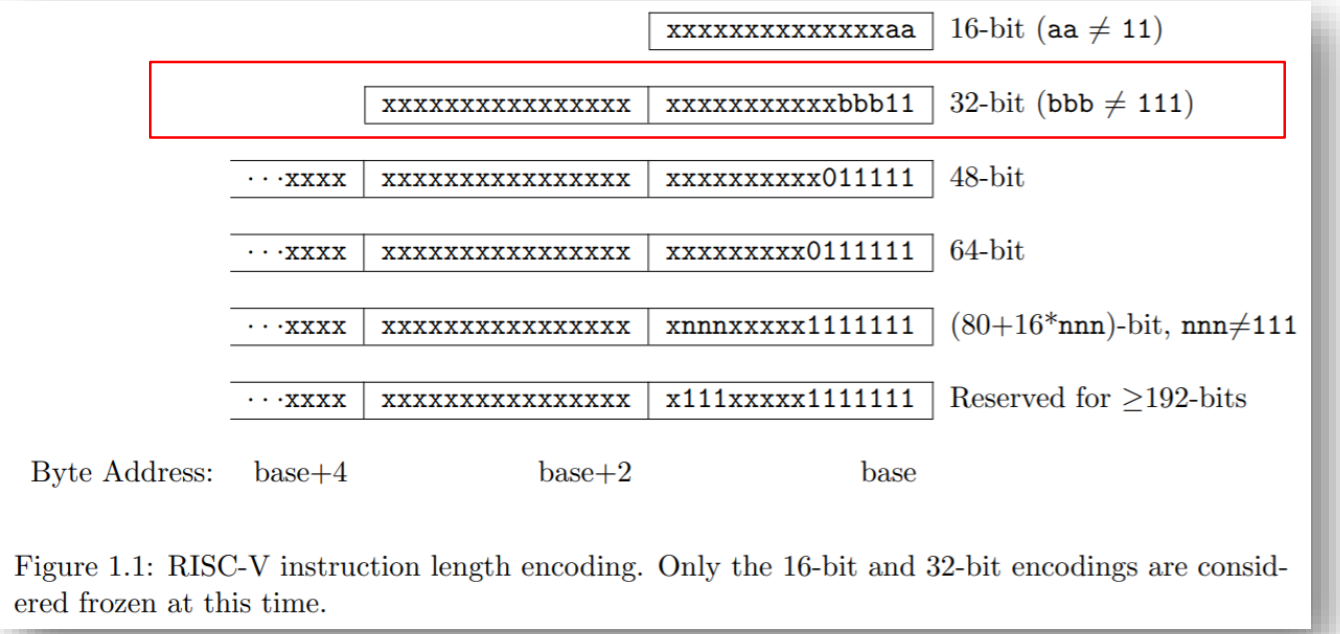


Figure 1.1: RISC-V instruction length encoding. Only the 16-bit and 32-bit encodings are considered frozen at this time.

RISC-V の命令形式と即値

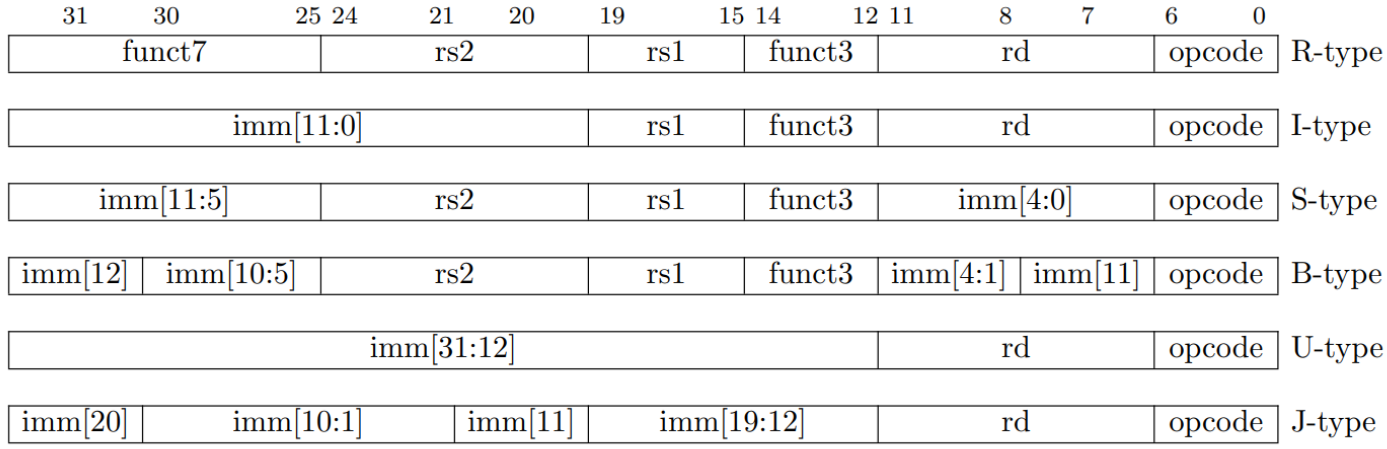


Figure 2.3: RISC-V base instruction formats showing immediate variants.

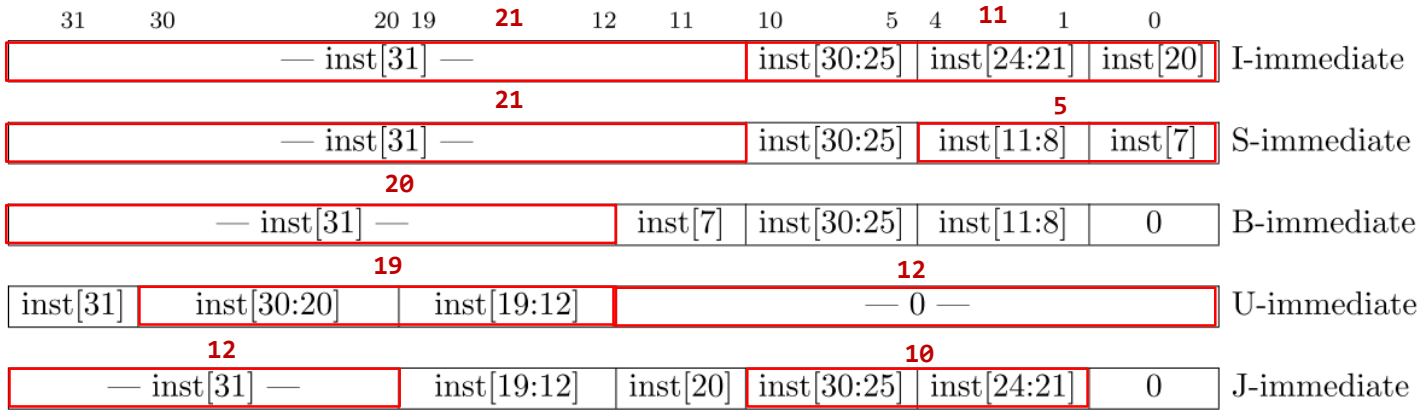
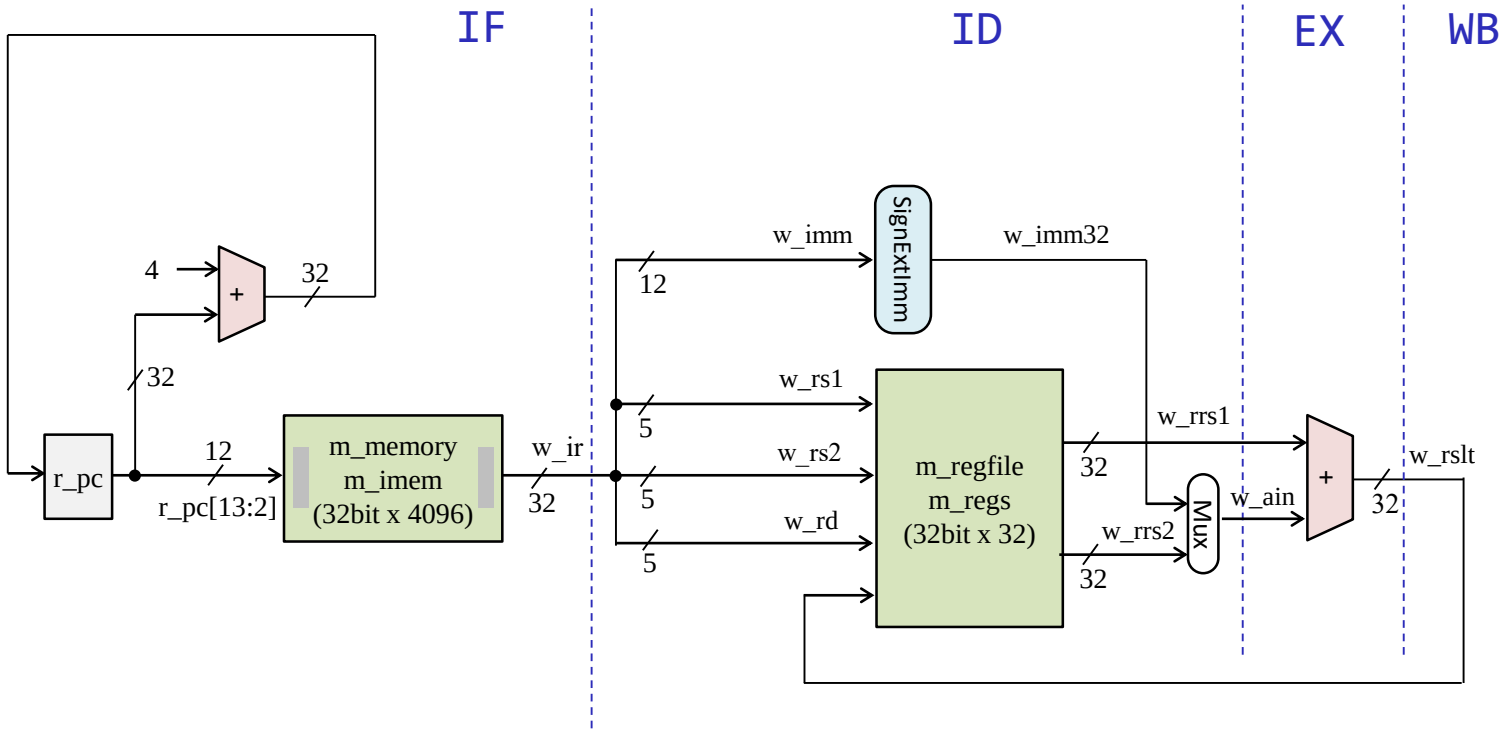
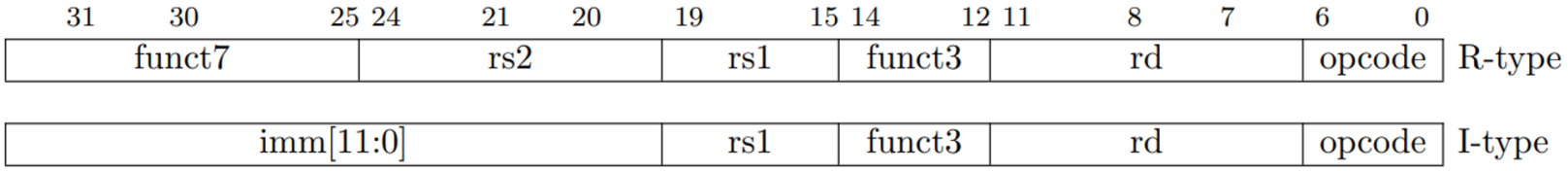


Figure 2.4: Types of immediate produced by RISC-V instructions. The fields are labeled with the instruction bits used to construct their value. Sign extension always uses inst[31].

m_proc03 add と addi を処理するプロセッサ

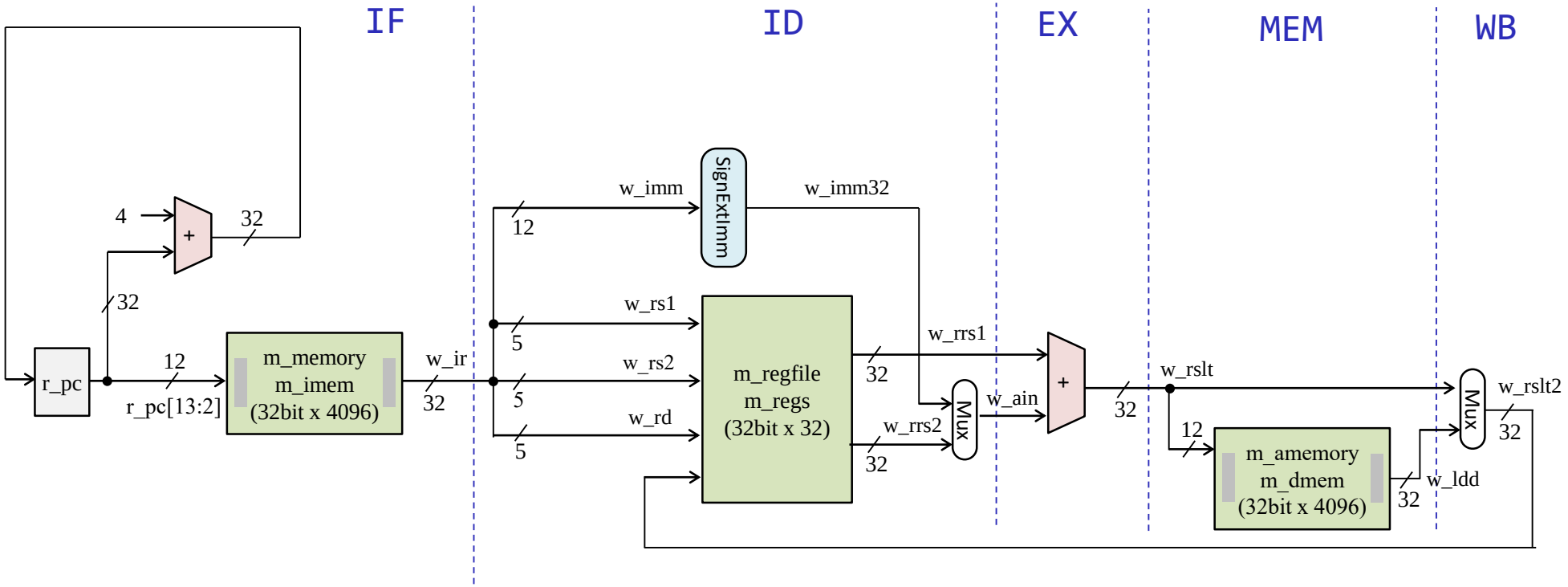
- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), ライトバック(WB) の処理をおこなう加算命令(add, addi)に対応したプロセッサのブロック図

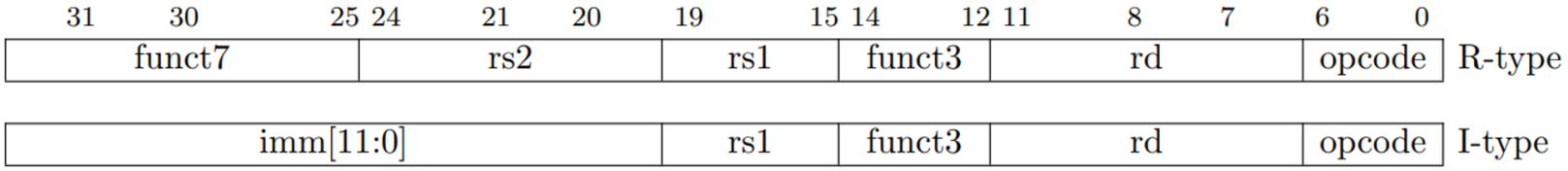




m_proc04 add, addi, lw を処理するプロセッサ

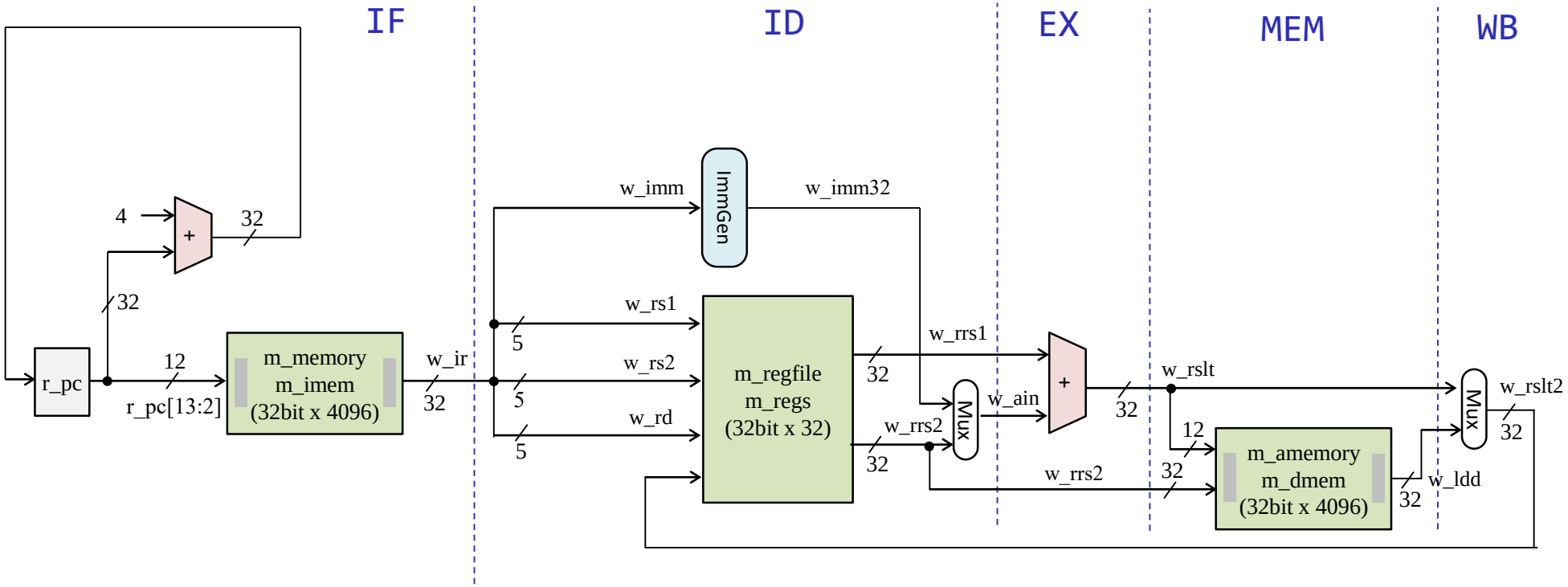
- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), メモリアクセス(MEM), ライトバック(WB) の処理をおこなうadd, addi, lw命令に対応したプロセッサ

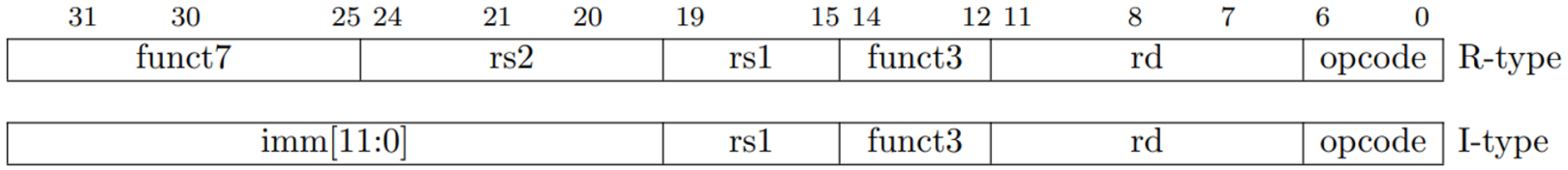




m_proc05 add, addi, lw, sw を処理するプロセッサ

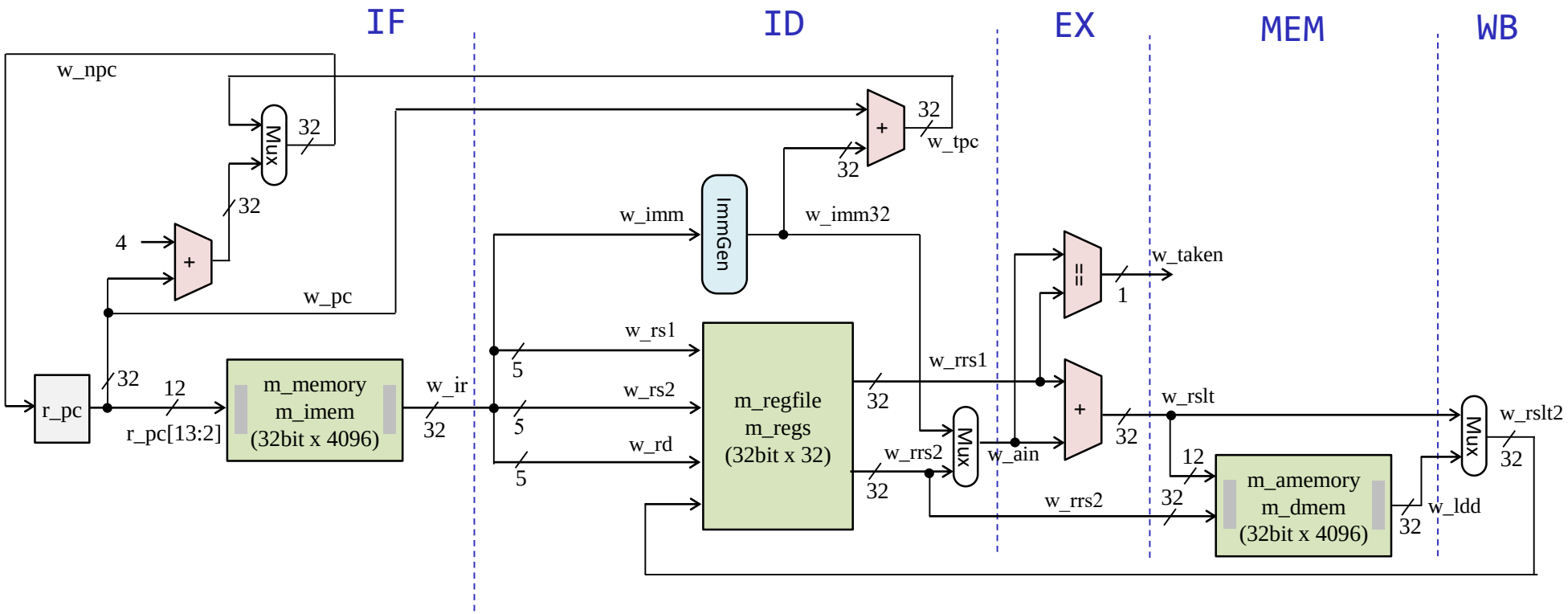
- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), メモリアクセス(MEM), ライトバック(WB) の処理をおこなうadd, addi, lw, sw命令に対応したプロセッサ

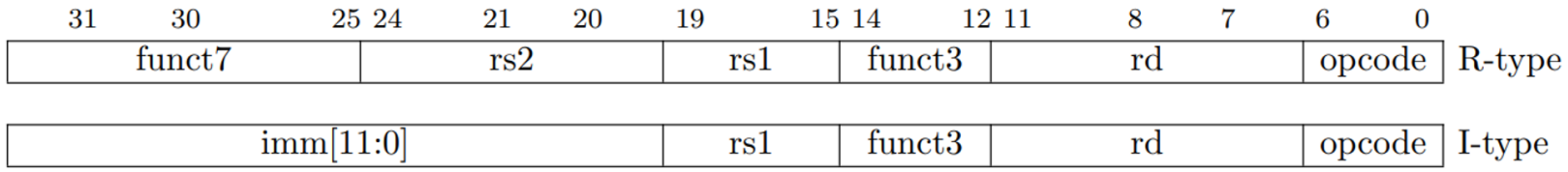




m_proc06 add, addi, lw, sw, beq を処理するプロセッサ

- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), メモリアクセス(MEM), ライトバック(WB) の処理をおこなうadd, addi, lw, sw, beq命令に対応したプロセッサ





RISC-V の32ビット基本命令セット RV32I

- LB (load byte)
- LH (load halfword)
- LW (load word)
- LBU (load byte, unsigned)
- LHU (load halfword, unsigned)
- SB (store byte(8-bit))
- SH (store halfword(16-bit))
- SW (store word(32-bit))

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
0000000	rs2	rs1	111	rd	0110011	AND
fm	pred	succ	rs1	000	rd	0001111	FENCE
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK

imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

little-endian, big-endian

In a little-endian configuration, multibyte stores write the least-significant register byte at the lowest memory byte address, followed by the other register bytes in ascending order of their significance. Loads similarly transfer the contents of the lesser memory byte addresses to the less-significant register bytes.

In a big-endian configuration, multibyte stores write the most-significant register byte at the lowest memory byte address, followed by the other register bytes in descending order of their significance. Loads similarly transfer the contents of the greater memory byte addresses to the less-significant register bytes.

Department of Computer Science
Course number: CSC.T341

コンピュータ論理設計
Computer Logic Design

9. シングルサイクルプロセッサの設計と実装
Design and Implementation of a Single Cycle Processor

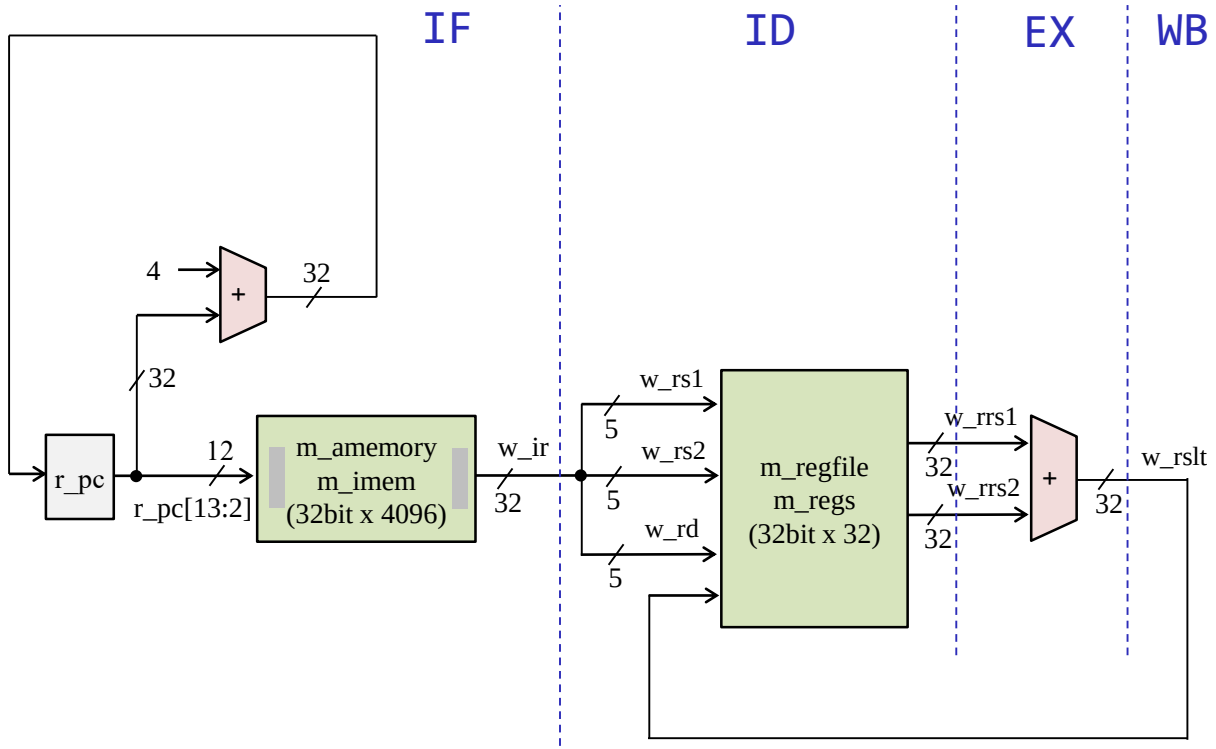
吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
講義：月曜日 10:45-12:25, 木曜日 10:45-12:25

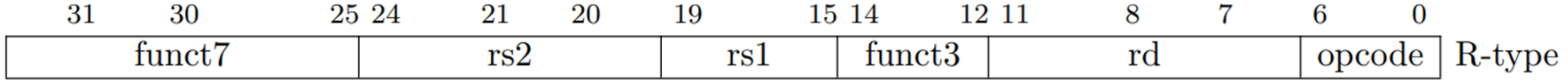
プロセッサが命令を処理するための基本的な5つのステップ

- IF (Instruction Fetch)
  メモリから命令をフェッチする.
- ID (Instruction Decode)
  命令をデコード(解読)しながら, レジスタの値を読み出す(Operand Fetch)
- EX (Execution)
  命令操作の実行またはアドレスの生成を行う.
- MEM (Memory Access)
  必要であれば, メモリ(データ・メモリ)のオペランドにアクセスする.
- WB (Write Back)
  必要であれば, 結果をレジスタに書き込む.

m_proc02 addを処理するシングルサイクルのプロセッサ

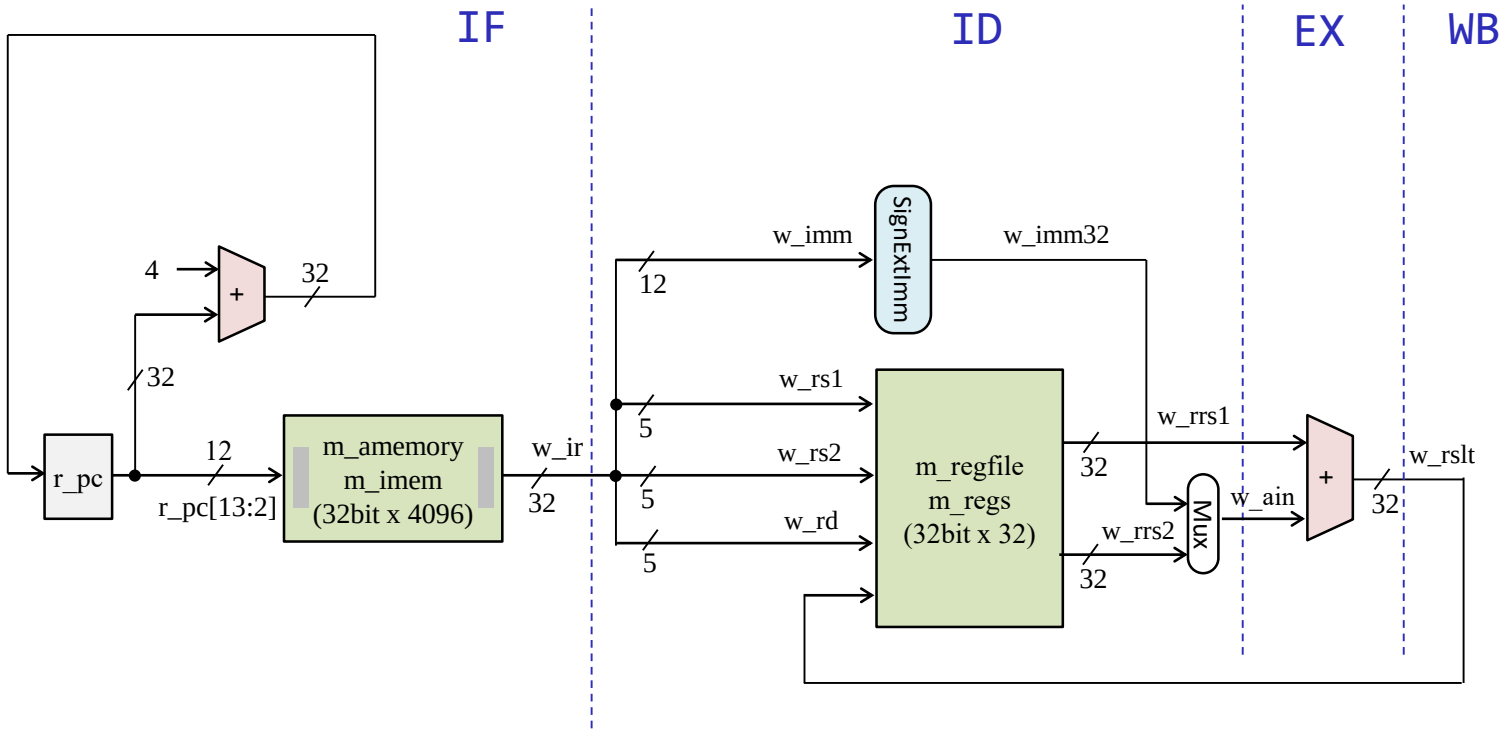
- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), ライトバック(WB) の処理をおこなう加算命令 (add) のみに対応したプロセッサのブロック図

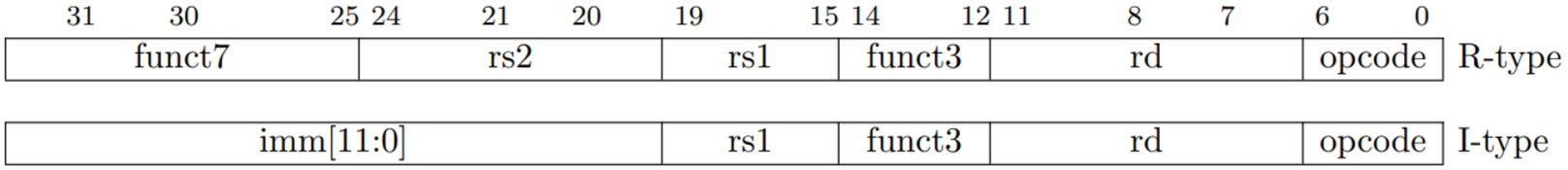




m_proc03 add と addi を処理するプロセッサ

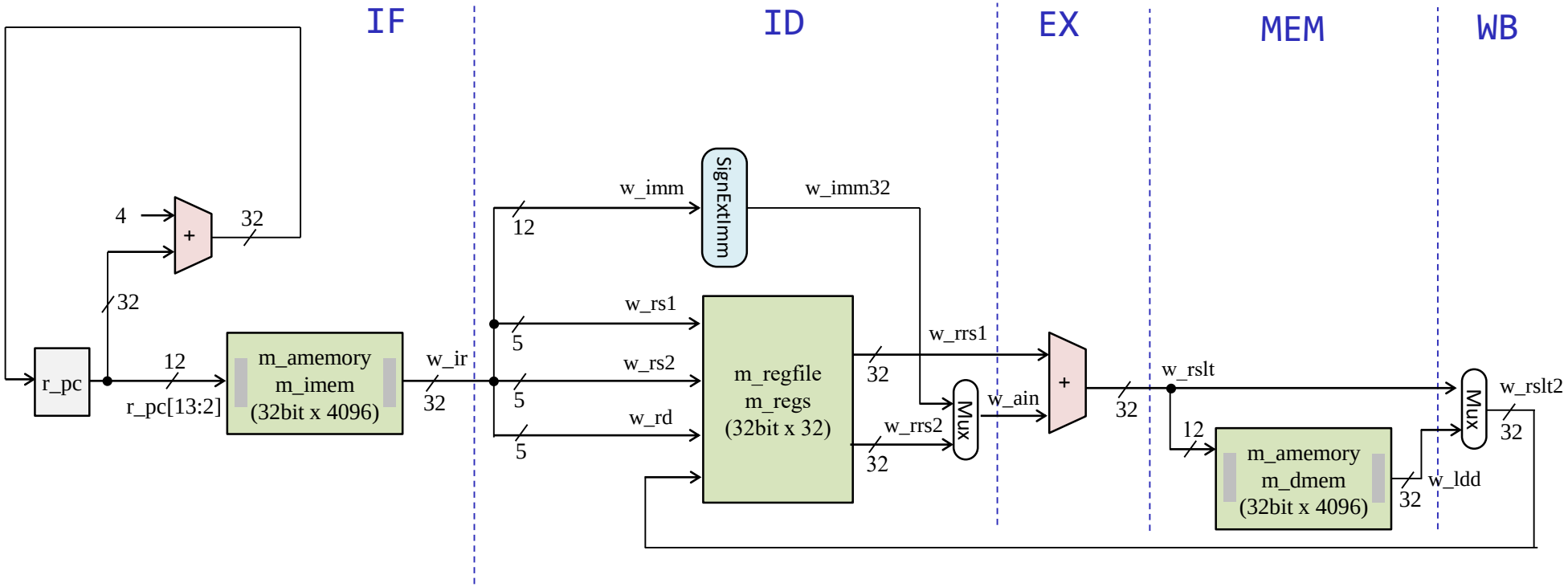
- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), ライトバック(WB) の処理をおこなう加算命令(add, addi)に対応したプロセッサのブロック図

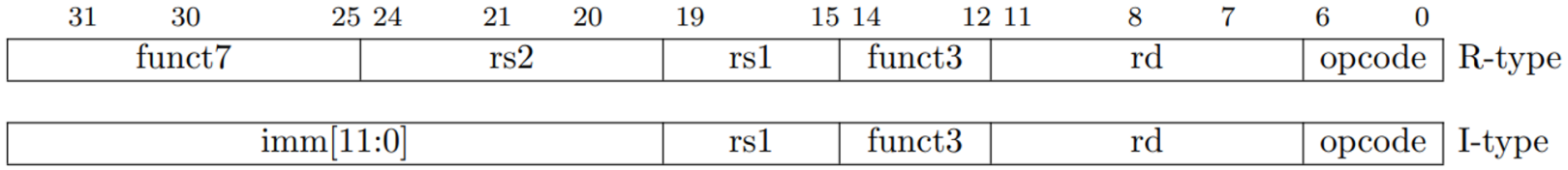




m_proc04 add, addi, lw を処理するプロセッサ

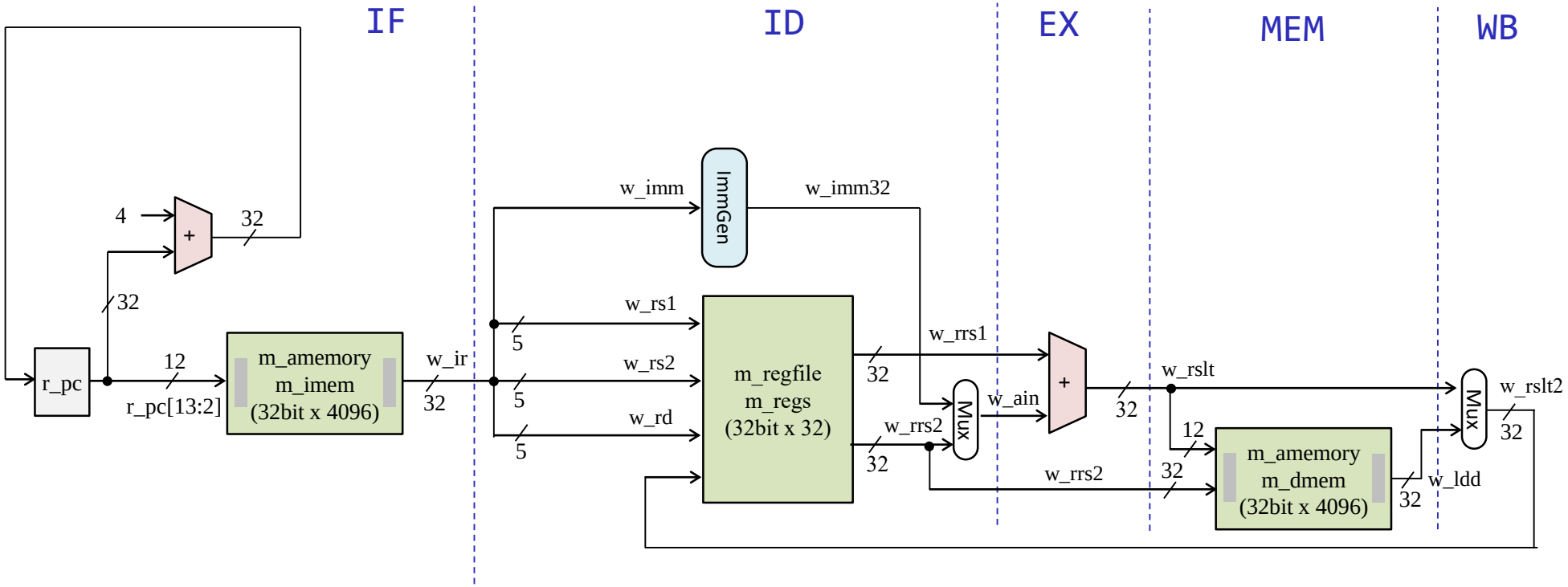
- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), メモリアクセス(MEM), ライトバック(WB) の処理をおこなうadd, addi, lw命令に対応したプロセッサ

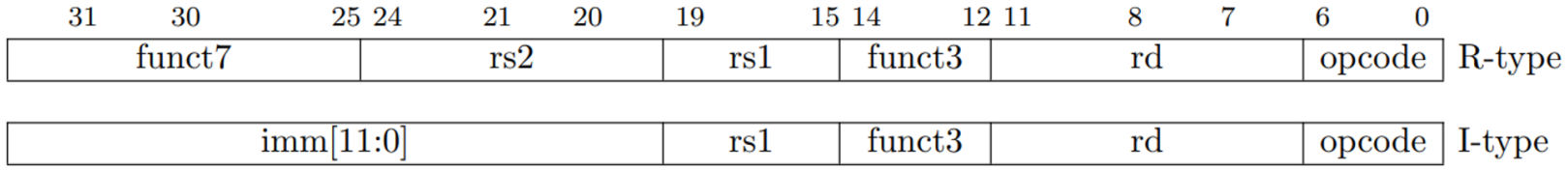




m_proc05 add, addi, lw, sw を処理するプロセッサ

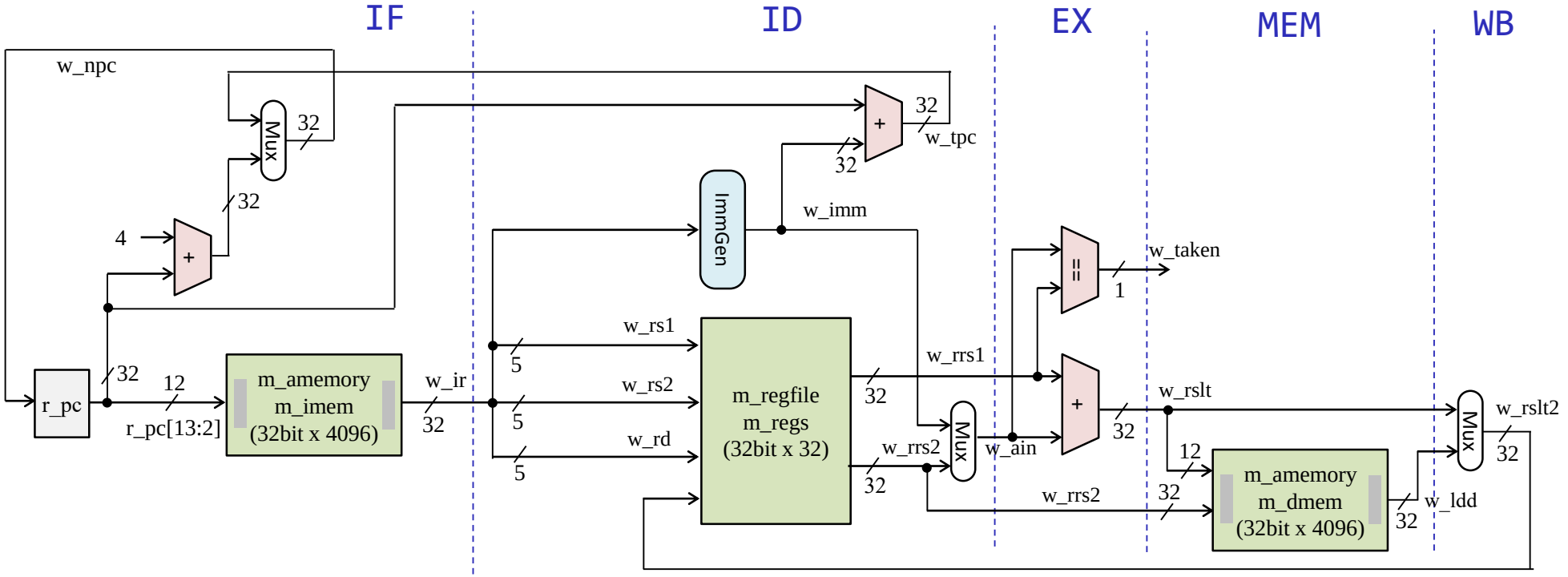
- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), メモリアクセス(MEM), ライトバック(WB) の処理をおこなうadd, addi, lw, sw命令に対応したプロセッサ

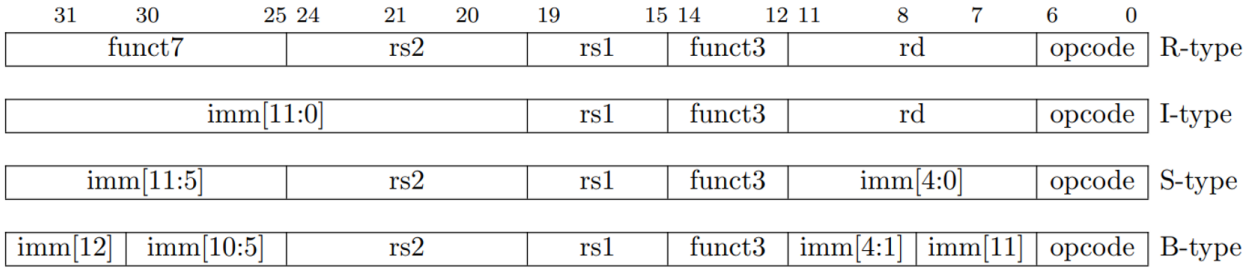




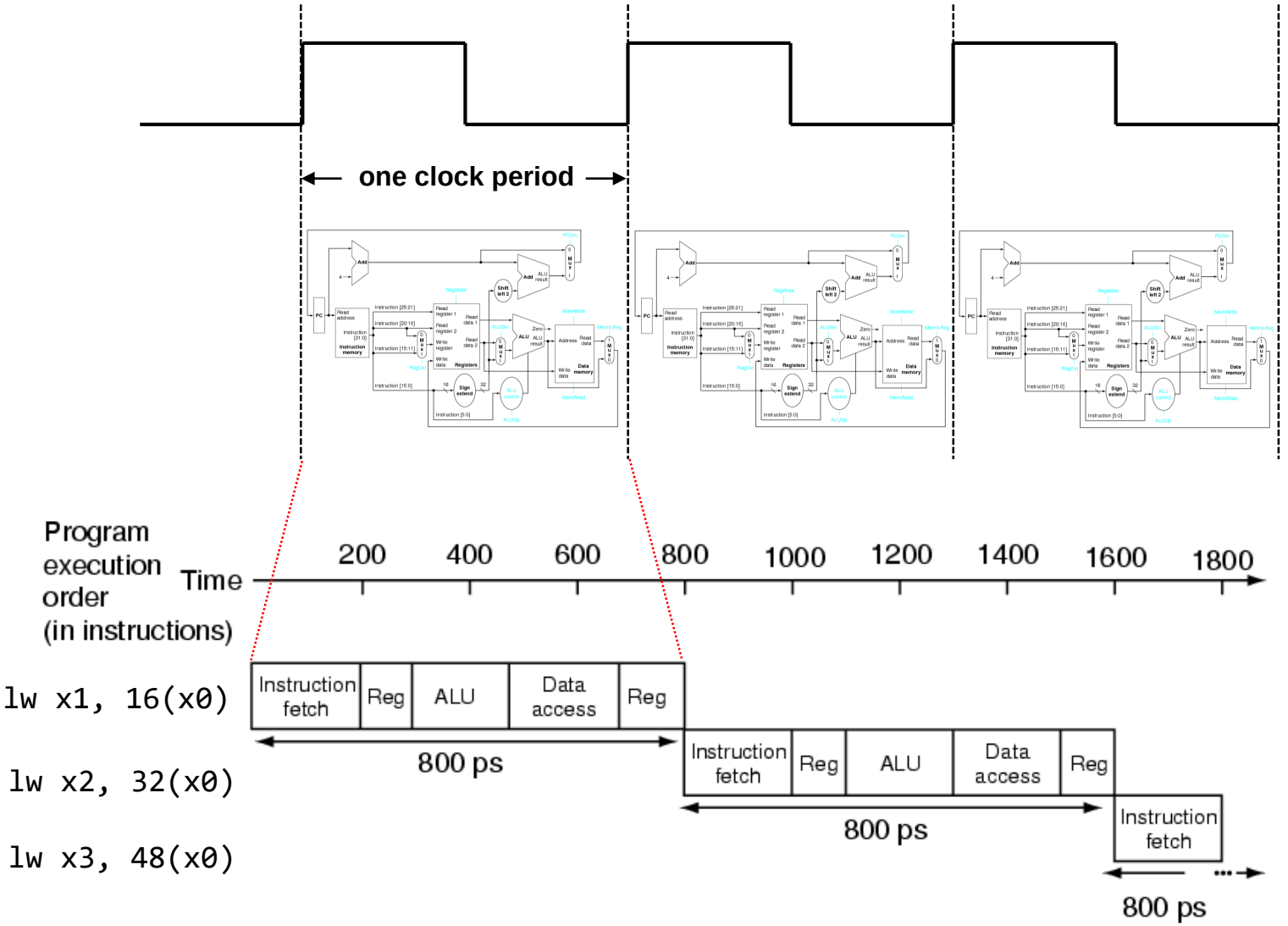
m_proc06 add, addi, lw, sw, beq を処理するプロセッサ

- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), メモリアクセス(MEM), ライトバック(WB) の処理をおこなうadd, addi, lw, sw, beq命令に対応したプロセッサ



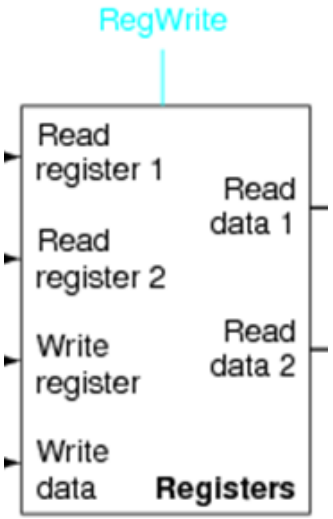


Single Cycle Processor (our baseline processor)



Register file, レジスタファイル m_regfile の実装

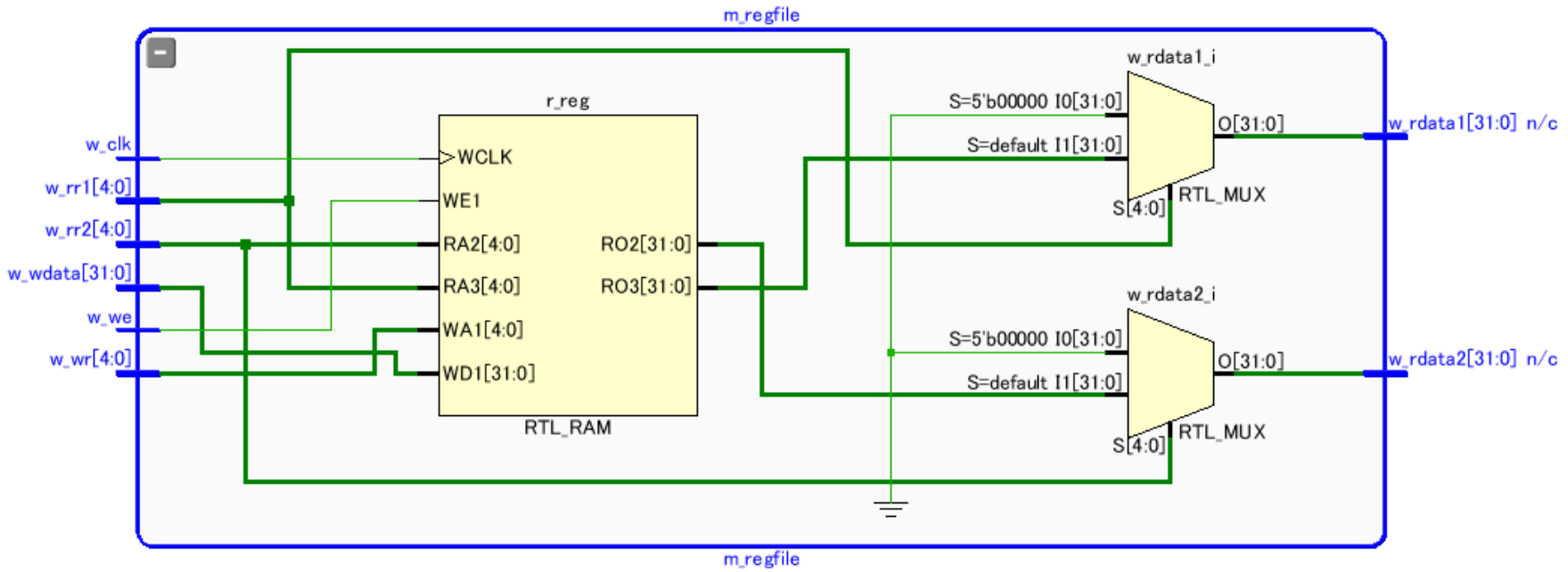
- Verilog HDLでは，ビット幅Bでワード数Wのメモリ m を reg [B-1:0] m [0:W-1] として宣言できる．
- w_rr1 で指定したレジスタの値を読み出し w_rdata1 に出力する．非同期の読み出し．
- w_rr2 で指定したレジスタの値を読み出し w_rdata2 に出力する．非同期の読み出し．
  - ただし，x0 (zero) の読み出しは，値0を出力する．
- posedge w_clk のタイミングで，w_we (write enable) が1の時に，w_wr (write register) で指定されたレジスタに w_wdata (write data) の値を書き込む．
- このモジュールではadd命令の動作確認のために x1 を 1 で，x2 を 2 で初期化している．



code112.v

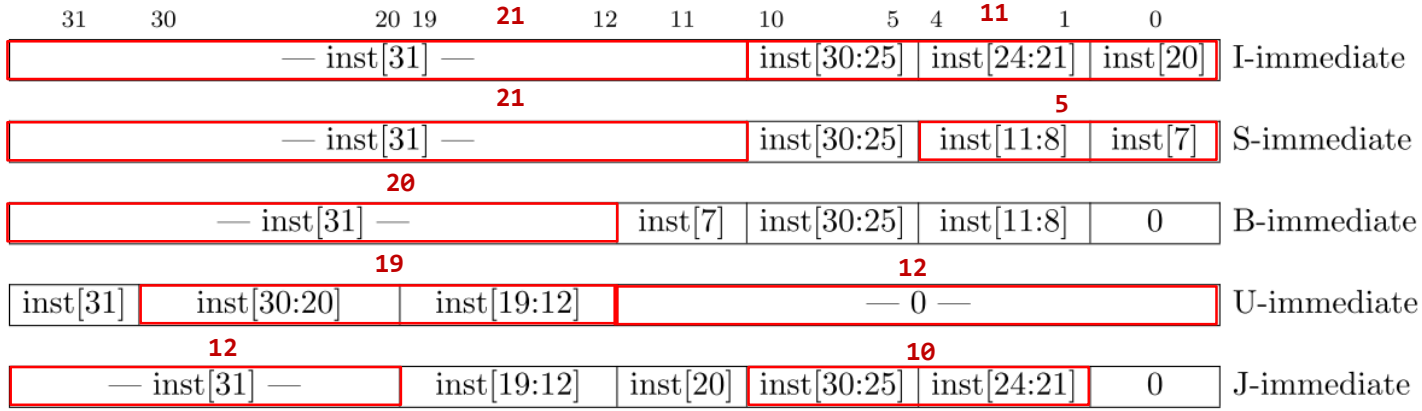
```
module m_regfile (w_clk, w_rr1, w_rr2, w_wr, w_we, w_wdata, w_rdata1, w_rdata2);
  input  wire        w_clk;
  input  wire [4:0]  w_rr1, w_rr2, w_wr;
  input  wire [31:0] w_wdata;
  input  wire        w_we;
  output wire [31:0] w_rdata1, w_rdata2;

  reg [31:0] r[0:31];
  assign w_rdata1 = (w_rr1==0) ? 0 : r[w_rr1];
  assign w_rdata2 = (w_rr2==0) ? 0 : r[w_rr2];
  always @(posedge w_clk) if(w_we) r[w_wr] <= w_wdata;

  initial r[1] = 1;
  initial r[2] = 2;
endmodule
```

Register file, module m_regfile



m_immgen RISC-Vの即値を出力するモジュール

- 32ビットの命令を入力 w_i として，RISC-Vの即値 r_imm を出力するモジュール
- reg を使っているが，これは組合せ回路となる．



code160.v

```
module m_immgen(w_i, r_imm); // module immediate generator
  input  wire [31:0] w_i;    // instruction
  output reg  [31:0] r_imm;  // r_immediate

  always @(*) case (w_i[6:2])
    5'b11000: r_imm <= {{20{w_i[31]}}, w_i[7], w_i[30:25], w_i[11:8], 1'b0};   // B-type
    5'b01000: r_imm <= {{21{w_i[31]}}, w_i[30:25], w_i[11:7]};                 // S-type
    5'b11011: r_imm <= {{12{w_i[31]}}, w_i[19:12], w_i[20], w_i[30:21], 1'b0}; // J-type
    5'b01101: r_imm <= {w_i[31:12], 12'b0};                                    // U-type
    5'b00101: r_imm <= {w_i[31:12], 12'b0};                                    // U-type
    default : r_imm <= {{21{w_i[31]}}, w_i[30:20]};                  // I-type & R-type
  endcase
endmodule
```

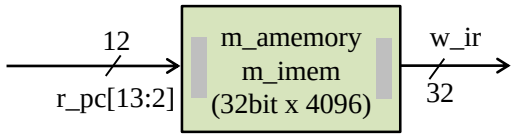
m_amemory 非同期式メモリの記述とシミュレーション

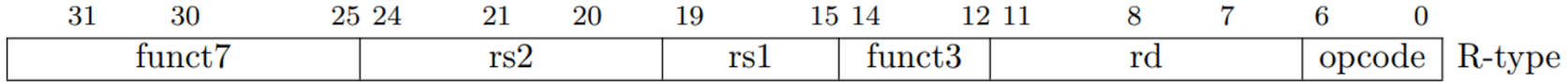
- Verilog HDLでは，ビット幅Bでワード数Wのメモリ m を reg [B-1:0] m [0:W-1] として宣言できる.
- 読み出す動作でクロック信号を利用しないメモリを非同期メモリ (asynchronous memory) と呼ぶ.
- 非同期式メモリの記述例を示す．シミュレーションでの読み出しの遅延を 20nsec とした．w_addr で指定されたアドレスの内容を読み出す．posedge w_clk のタイミングで，w_we (write enable) が1の時に，w_addr で指定されたアドレスに w_din (data in) の値を書き込む.
- このコードをシミュレーションして，波形を確認すること.

```
module m_amemory (w_clk, w_addr, w_we, w_din, w_dout);
  input  wire w_clk, w_we;
  input  wire [11:0] w_addr;
  input  wire [31:0] w_din;
  output wire [31:0] w_dout;

  reg [31:0] cm_ram [0:4095]; // 4K word (4096 x 32bit) memory
  always @(posedge w_clk) if (w_we) cm_ram[w_addr] <= w_din;
  assign #20 w_dout = cm_ram[w_addr];

  initial begin
    cm_ram[0]={7'd0, 5'd0, 5'd0, 3'd0, 5'd0, 7'b0110011}; // add x0, x0, x0
    cm_ram[1]={7'd0, 5'd1, 5'd0, 3'd0, 5'd4, 7'b0110011}; // add x4, x0, x1
    cm_ram[2]={7'd0, 5'd2, 5'd1, 3'd0, 5'd5, 7'b0110011}; // add x5, x1, x2
    cm_ram[3]={7'd0, 5'd5, 5'd4, 3'd0, 5'd6, 7'b0110011}; // add x6, x4, x5
    cm_ram[4]={7'd0, 5'd0, 5'd0, 3'd0, 5'd0, 7'b0110011}; // add x0, x0, x0
  end
endmodule
```

```
module m_top ();
  reg r_clk=0; initial forever #50 r_clk = ~r_clk;
  reg [31:0] r_pc = 0;
  always @(posedge r_clk) r_pc <= #3 r_pc + 4;

  wire [31:0] w_data;
  m_amemory m (r_clk, r_pc[13:2], 1'd0, 32'd0, w_data);

  initial $dumpfile("main.vcd"); /* file name for GTKWave */
  initial $dumpvars(0, m_top);   /* module    for GTKWave */
  initial #1000 $finish;
  always@(*) #80 $write("%3d %d %x¥n", $time, r_pc, w_data);
endmodule
```

code111.v





m_amemory の修正 include “program.txt”

- 命令メモリとデータメモリとして用いる m_amemory の内容を program.txt で初期化.
- x30 への書き込みをレジスタに保存して, プロセッサの出力 w_led とする.
- このプログラムを実行して beq に到達した時点の w_led の値は 55 + 9 = 64 になる.

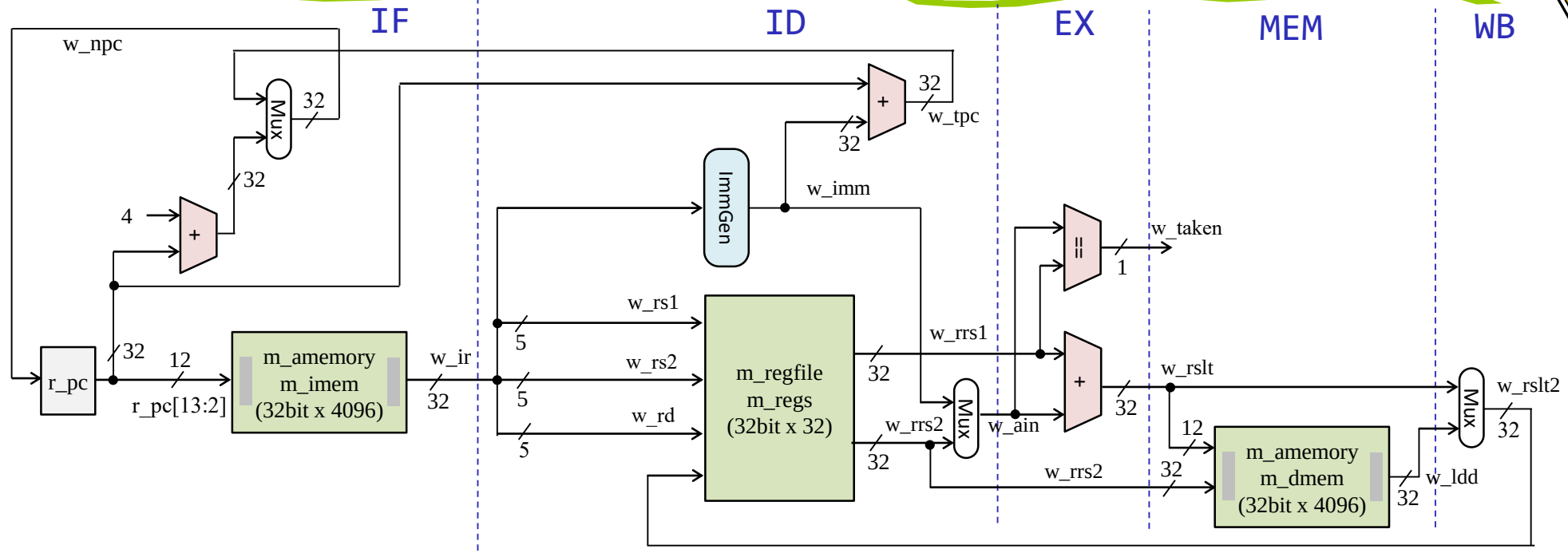
code160.v

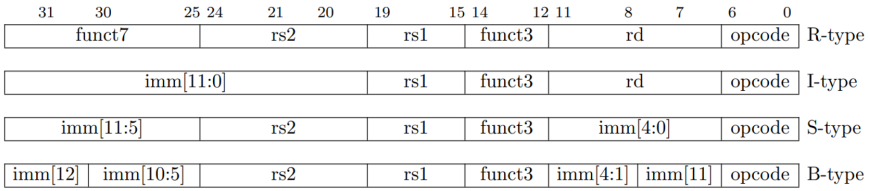
```
module m_amemory (w_clk, w_addr, w_we, w_din, w_dout);
  input  wire w_clk, w_we;
  input  wire [11:0] w_addr;
  input  wire [31:0] w_din;
  output wire [31:0] w_dout;
  reg [31:0]          cm_ram [0:4095]; // 4K word (4096 x 32bit) memory
  always @(posedge w_clk) if (w_we) cm_ram[w_addr] <= w_din;
  assign #20 w_dout = cm_ram[w_addr];
`include "program.txt"
endmodule
```

program.txt

```
initial begin
  cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //    add  x0, x0, x0  // NOP
  cm_ram[1]={12'd55,     5'd0, 3'b000, 5'd4, 7'b0010011}; //    addi x4, x0, 55  // x4 = 55
  cm_ram[2]={7'd0, 5'd4, 5'd0, 3'b010, 5'd16,7'b0100011}; //    sw   x4, 16(x0)  // m[16] = x4
  cm_ram[3]={12'd16,     5'd0, 3'b010, 5'd7, 7'b0000011}; //    lw   x7, 16(x0)  // x7 = m[16]
  cm_ram[4]={12’d9,      5'd0, 3'b000, 5'd2, 7'b0010011}; //    addi x2, x0, 9   // x2 = 9
  cm_ram[5]={7'd0, 5'd2, 5'd7, 3'b000, 5'd3, 7'b0110011}; //    add  x3, x7, x2  // x3 = x7 + x2
  cm_ram[6]={7'd0, 5'd3, 5'd0, 3'b000, 5'd30,7'b0110011}; //    add  x30,x0, x3  // led = x3
  cm_ram[7]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b1100011}; // L: beq  x0, x0, L
end
```

m_proc06 add, addi, lw, sw, beq を処理するプロセッサ

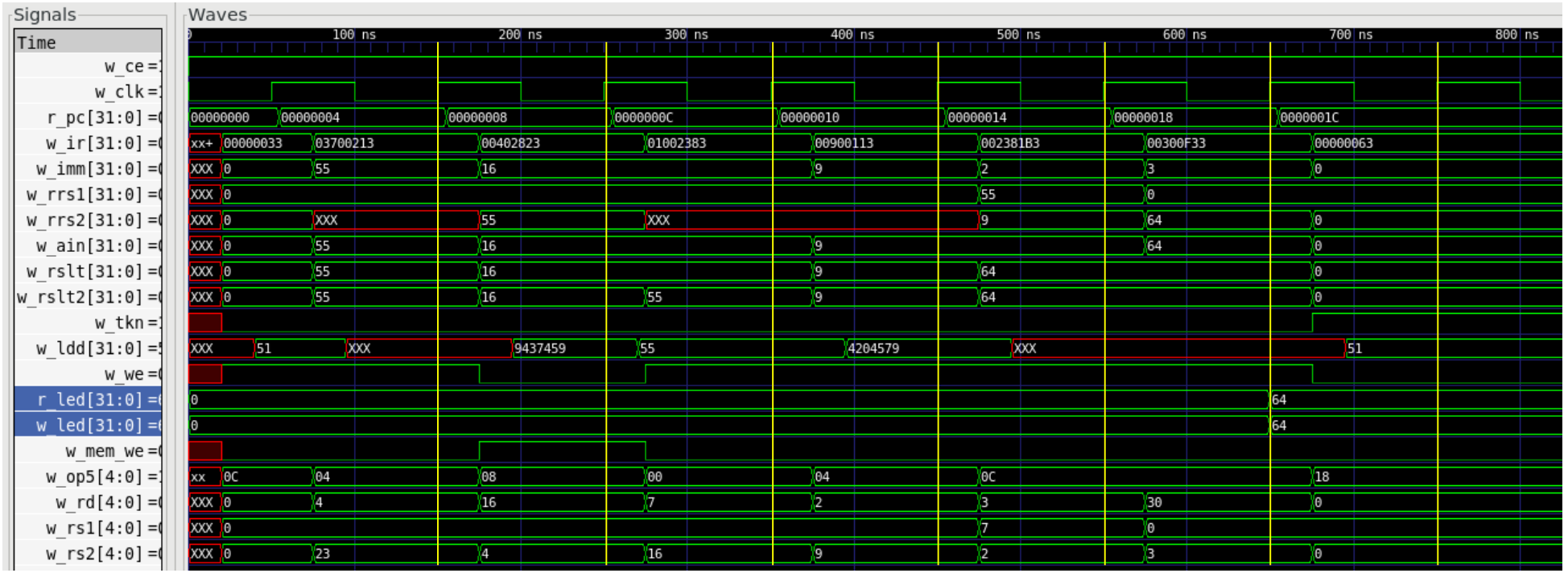


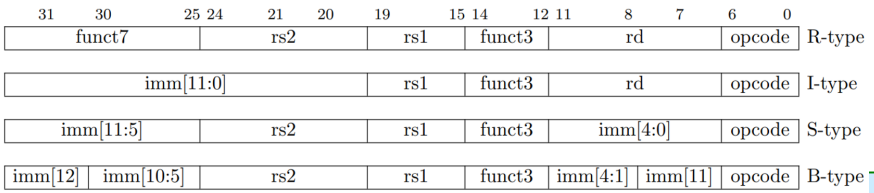


クロック毎の動作を確認する.

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //    add  x0, x0, x0  // NOP
cm_ram[1]={12'd55,     5'd0, 3'b000, 5'd4, 7'b0010011}; //    addi x4, x0, 55  // x4 = 55
cm_ram[2]={7'd0, 5'd4, 5'd0, 3'b010, 5'd16,7'b0100011}; //    sw   x4, 16(x0)  // m[16] = x4
cm_ram[3]={12'd16,     5'd0, 3'b010, 5'd7, 7'b0000011}; //    lw   x7, 16(x0)  // x7 = m[16]
cm_ram[4]={12'd9,      5'd0, 3'b000, 5'd2, 7'b0010011}; //    addi x2, x0, 9   // x2 = 9
cm_ram[5]={7'd0, 5'd2, 5'd7, 3'b000, 5'd3, 7'b0110011}; //    add  x3, x7, x2  // x3 = x7 + x2
cm_ram[6]={7'd0, 5'd3, 5'd0, 3'b000, 5'd30,7'b0110011}; //    add  x30,x0, x3  // led = x3
cm_ram[7]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b1100011}; // L: beq  x0, x0, L
```

m_proc06 add, addi, lw, sw, beq を処理するプロセッサ






```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //    add  x0, x0, x0  // NOP
cm_ram[1]={12'd55,     5'd0, 3'b000, 5'd4, 7'b0010011}; //    addi x4, x0, 55  // x4 = 55
cm_ram[2]={7'd0, 5'd4, 5'd0, 3'b010, 5'd16,7'b0100011}; //    sw   x4, 16(x0)  // m[16] = x4
cm_ram[3]={12'd16,     5'd0, 3'b010, 5'd7, 7'b0000011}; //    lw   x7, 16(x0)  // x7 = m[16]
cm_ram[4]={12'd9,      5'd0, 3'b000, 5'd2, 7'b0010011}; //    addi x2, x0, 9   // x2 = 9
cm_ram[5]={7'd0, 5'd2, 5'd7, 3'b000, 5'd3, 7'b0110011}; //    add  x3, x7, x2  // x3 = x7 + x2
cm_ram[6]={7'd0, 5'd3, 5'd0, 3'b000, 5'd30,7'b0110011}; //    add  x30,x0, x3  // led = x3
cm_ram[7]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b1100011}; // L: beq  x0, x0, L
```

m_proc06 add, addi, lw, sw, beq を処理するプロセッサ

m_top はシミュレーション用

```
/***** top module for simulation *****/
module m_top ();
   reg r_clk=0; initial forever #50 r_clk = ~r_clk;
   wire [31:0] w_led;

   initial $dumpfile("main.vcd");
   initial $dumpvars(0, m_top);

   m_proc06 p (r_clk, 1'b1, w_led);
   initial $write("time: r_pc     w_ir     w_rrs1  w_ain   r_rslt  r_led¥n");
   always@(posedge r_clk) $write("%4d: %x %x %07d %07d %07d %07d¥n", $time,
                          p.r_pc, p.w_ir, p.w_rrs1, p.w_ain, p.w_rslt, w_led);
   initial #1000 $finish;
endmodule

/***** main module for FPGA implementation *****/
/*
module m_main (w_clk, w_led);
   input  wire w_clk;
   output wire [3:0] w_led;

   wire w_clk2, w_locked;
   clk_wiz_0 clk_w0 (w_clk2, 0, w_locked, w_clk);

   wire [31:0] w_dout;
   m_proc06 p (w_clk2, w_locked, w_dout);

   vio_0 vio_00(w_clk2, w_dout);

   reg [3:0] r_led = 0;
   always @(posedge w_clk2)
     r_led <= {^w_dout[31:24], ^w_dout[23:16], ^w_dout[15:8], ^w_dout[7:0]};
   assign w_led = r_led;
endmodule
*/
```

code160.v

m_proc06 add, addi, lw, sw, beq を処理するプロセッサ

m_main はFPGA用

```
/***** top module for simulation *****/
// module m_top ();
//   reg r_clk=0; initial forever #50 r_clk = ~r_clk;
//   wire [31:0] w_led;
//
//   initial $dumpfile("main.vcd");
//   initial $dumpvars(0, m_top);
//
//   m_proc06 p (r_clk, 1'b1, w_led);
//   initial $write("time: r_pc     w_ir     w_rrs1  w_ain   r_rslt  r_led¥n");
//   always@(posedge r_clk) $write("%4d: %x %x %07d %07d %07d %07d¥n", $time,
//                          p.r_pc, p.w_ir, p.w_rrs1, p.w_ain, p.w_rslt, w_led);
//   initial #1000 $finish;
// endmodule

/***** main module for FPGA implementation *****/

module m_main (w_clk, w_led);
   input  wire w_clk;
   output wire [3:0] w_led;

   wire w_clk2, w_locked;
   clk_wiz_0 clk_w0 (w_clk2, 0, w_locked, w_clk);

   wire [31:0] w_dout;
   m_proc06 p (w_clk2, w_locked, w_dout);

   vio_0 vio_00(w_clk2, w_dout);

   reg [3:0] r_led = 0;
   always @(posedge w_clk2)
     r_led <= {^w_dout[31:24], ^w_dout[23:16], ^w_dout[15:8], ^w_dout[7:0]};
   assign w_led = r_led;
endmodule
```

code160.v

m_proc06 add, addi, lw, sw, beq を処理するプロセッサ

青色の部分を記述する.

code160.v

```
module m_proc06 (w_clk, w_ce, w_led);
  input  wire w_clk, w_ce;
  output wire [31:0] w_led;

  reg [31:0] r_pc = 0;
  wire [31:0] w_ir;
  wire [4:0]  w_op5 = w_ir[6:2];
  wire [4:0]  w_rs1 = w_ir[19:15];
  wire [4:0]  w_rs2 = w_ir[24:20];
  wire [4:0]  w_rd  = w_ir[11:7];
  wire w_we = w_ce & (w_op5==5'b01100 || w_op5==5'b00100 || w_op5==5'b00000);

  wire [31:0] w_imm, w_rrs1, w_rrs2, w_ain, w_rslt, w_ldd, w_rslt2;

  /* Please describe here by yourself */


  reg [31:0] r_led = 0;
  always @(posedge w_clk) if(w_we & w_rd==30) r_led <= w_rslt;
  assign w_led = r_led;
endmodule
```

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //    add  x0, x0, x0  // NOP
cm_ram[1]={12'd55,     5'd0, 3'b000, 5'd4, 7'b0010011}; //    addi x4, x0, 55  // x4 = 55
cm_ram[2]={7'd0, 5'd4, 5'd0, 3'b010, 5'd16,7'b0100011}; //    sw   x4, 16(x0)  // m[16] = x4
cm_ram[3]={12'd16,     5'd0, 3'b010, 5'd7, 7'b0000011}; //    lw   x7, 16(x0)  // x7 = m[16]
cm_ram[4]={12'd9,      5'd0, 3'b000, 5'd2, 7'b0010011}; //    addi x2, x0, 9   // x2 = 9
cm_ram[5]={7'd0, 5'd2, 5'd7, 3'b000, 5'd3, 7'b0110011}; //    add  x3, x7, x2  // x3 = x7 + x2
cm_ram[6]={7'd0, 5'd3, 5'd0, 3'b000, 5'd30,7'b0110011}; //    add  x30,x0, x3  // led = x3
cm_ram[7]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b1100011}; // L: beq  x0, x0, L
```

ヒント（m_proc06 の実装）

青色の部分を少し記述した.

code160.v

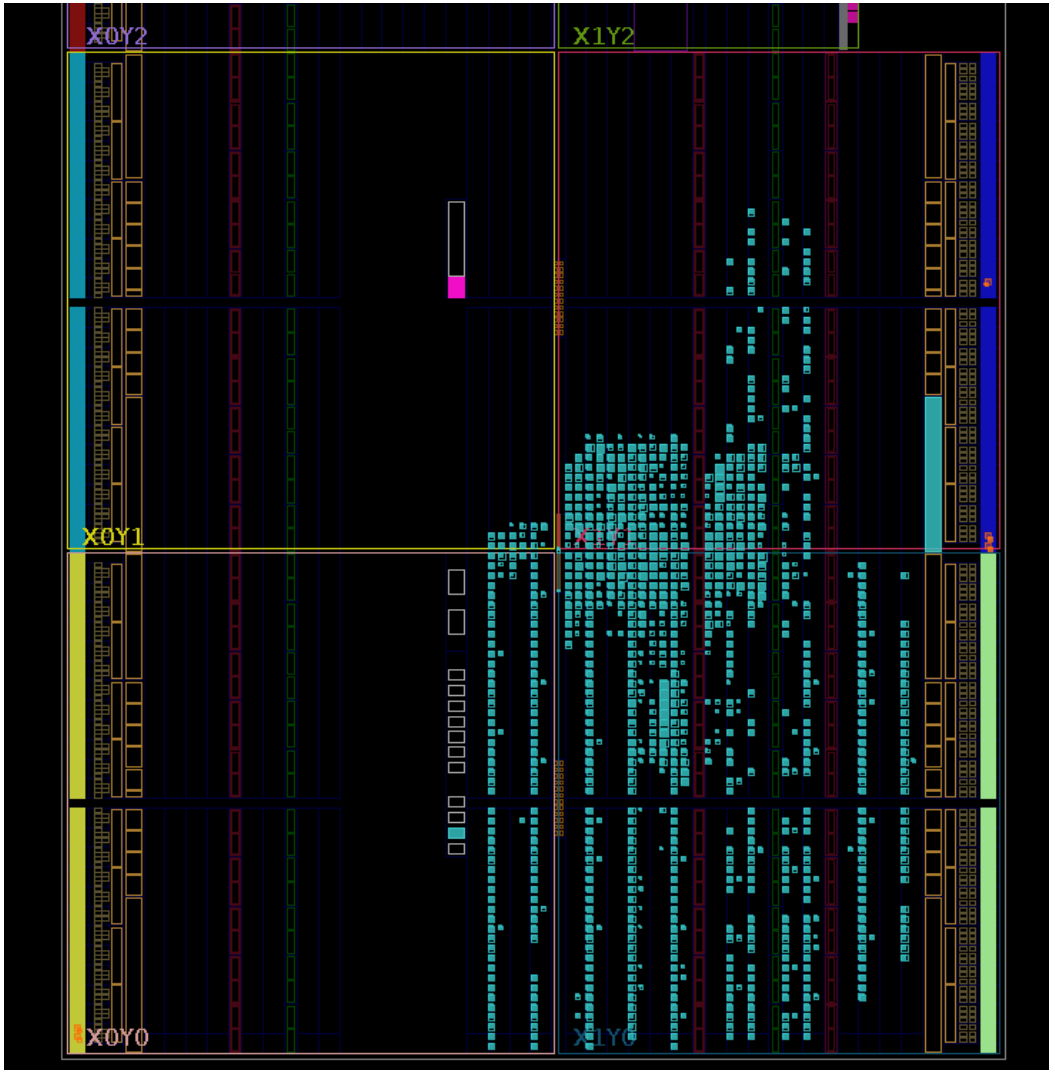
```
module m_proc06 (w_clk, w_ce, w_led);
  input  wire w_clk, w_ce;
  output wire [31:0] w_led;

  reg [31:0] r_pc = 0;
  wire [31:0] w_ir;
  wire [4:0]  w_op5 = w_ir[6:2];
  wire [4:0]  w_rs1 = w_ir[19:15];
  wire [4:0]  w_rs2 = w_ir[24:20];
  wire [4:0]  w_rd  = w_ir[11:7];
  wire w_we = w_ce & (w_op5==5'b01100 || w_op5==5'b00100 || w_op5==5'b00000);

  wire [31:0] w_imm, w_rrs1, w_rrs2, w_ain, w_rslt, w_ldd, w_rslt2;

  /* Please describe here by yourself */
  m_amemory m_imem (w_clk, r_pc[13:2], 1'd0, 32'd0, w_ir);
  m_immgen m_immgen0 (w_ir, w_imm);
  m_regfile m_regs (w_clk, w_rs1, w_rs2, w_rd, w_we, w_rslt2, w_rrs1, w_rrs2);
  assign w_ain = (w_op5==5'b01100) ? w_rrs2 : w_imm;
  assign w_rslt = w_rrs1 + w_ain;

  // about data memory here

  // about r_pc update here



  reg [31:0] r_led = 0;
  always @(posedge w_clk) if(w_we & w_rd==30) r_led <= w_rslt;
  assign w_led = r_led;
endmodule
```

m_proc06 の論理合成，配置配線の結果

Utilization Post-Synthesis | **Post-Implementation**

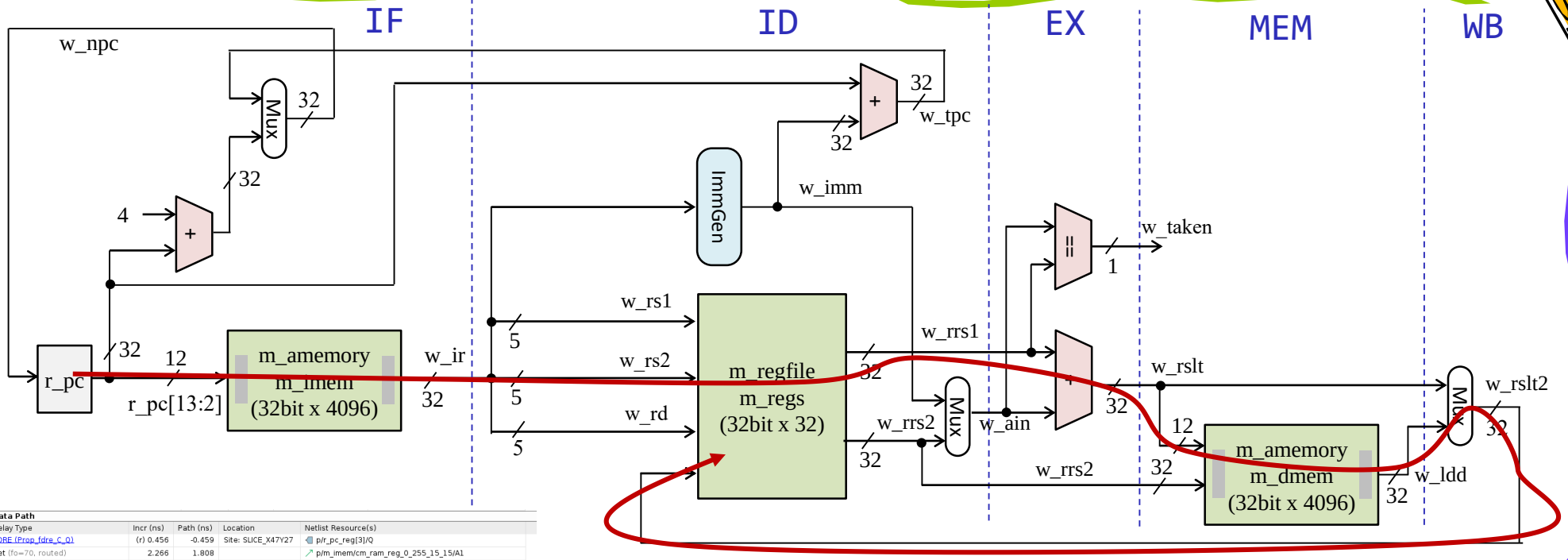
Graph | **Table**

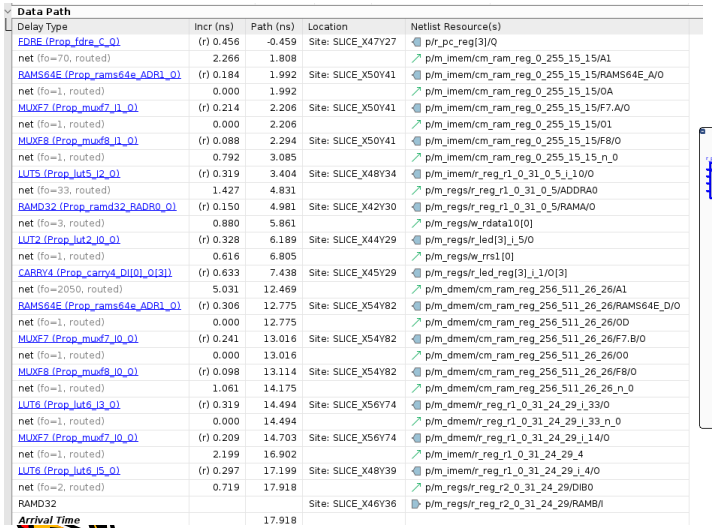
Resource	Utilization	Available	Utilization %
LUT	3095	20800	14.88
LUTRAM	2184	9600	22.75
FF	1199	41600	2.88
IO	5	210	2.38
BUFG	3	32	9.38
MMCM	1	5	20.00



この配置配線の結果は，実装によって変わる．

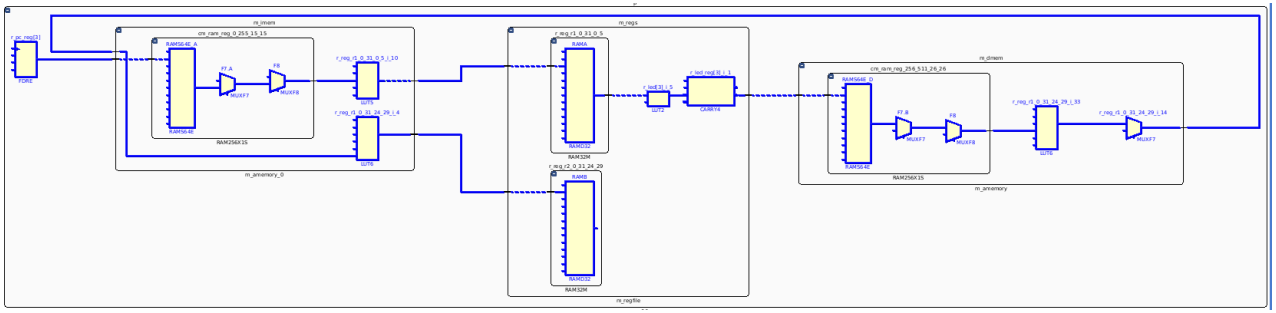
m_proc06 のクリティカルパスは





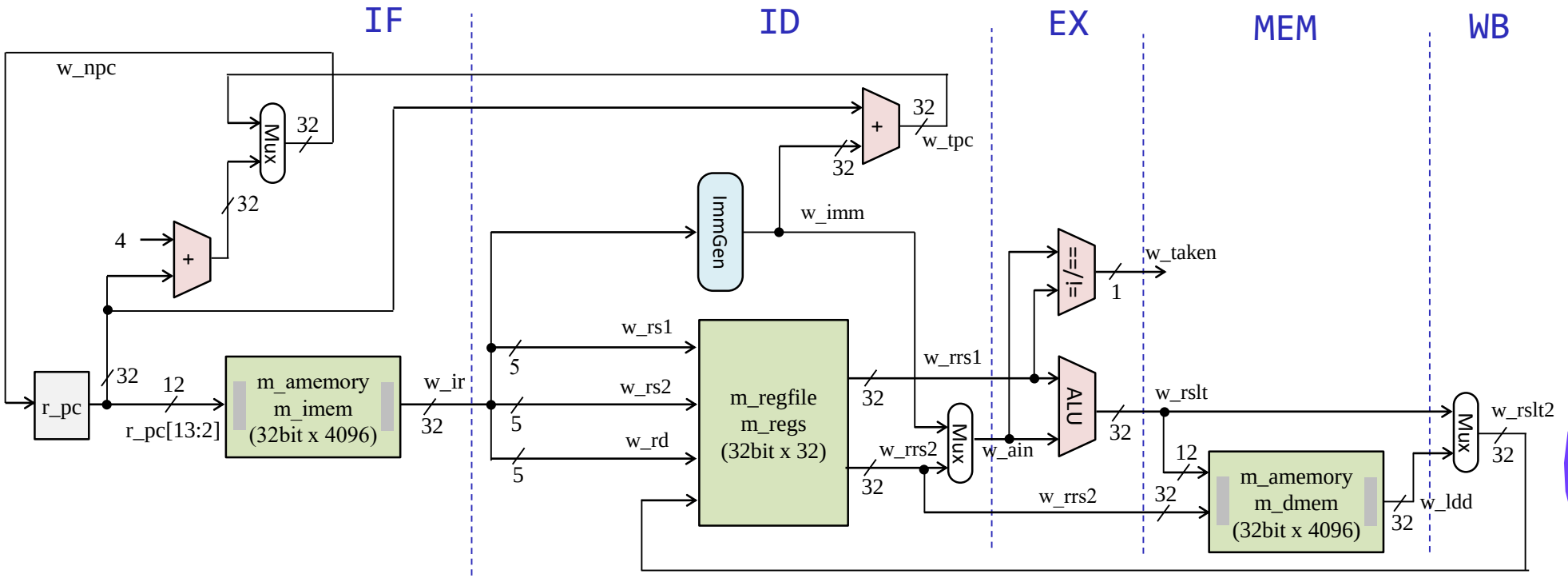
Data Path

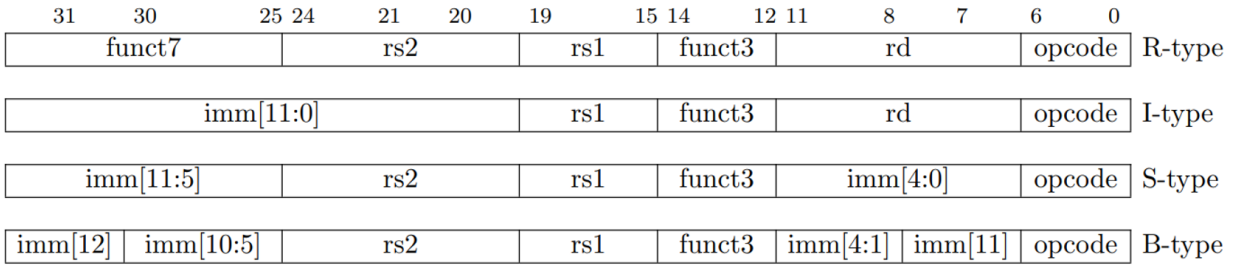
Delay Type	Incr (ns)	Path (ns)	Location	Netlist Resource(s)
FDRE (Prop_fdre_C_Q)	(r) 0.456	-0.459	Site: SLICE_X47Y27	p/r_pc_reg[3]/Q
net (fo=70, routed)	2.266	1.808		p/m_imem/cm_ram_reg_0_255_15_15/A1
RAMS64E (Prop_rams64e_ADR1_O)	(r) 0.184	1.992	Site: SLICE_X50Y41	p/m_imem/cm_ram_reg_0_255_15_15/RAMS64E_A/O
net (fo=1, routed)	0.000	1.992		p/m_imem/cm_ram_reg_0_255_15_15/0A
MUXF7 (Prop_muxf7_I1_O)	(r) 0.214	2.206	Site: SLICE_X50Y41	p/m_imem/cm_ram_reg_0_255_15_15/F7.A/O
net (fo=1, routed)	0.000	2.206		p/m_imem/cm_ram_reg_0_255_15_15/01
MUXF8 (Prop_muxf8_I1_O)	(r) 0.088	2.294	Site: SLICE_X50Y41	p/m_imem/cm_ram_reg_0_255_15_15/F8/O
net (fo=1, routed)	0.792	3.085		p/m_imem/cm_ram_reg_0_255_15_15_n_0
LUT5 (Prop_lut5_I2_O)	(r) 0.319	3.404	Site: SLICE_X48Y34	p/m_imem/r_reg_r1_0_31_0_5_i_10/O
net (fo=33, routed)	1.427	4.831		p/m_regs/r_reg_r1_0_31_0_5/ADDRA0
RAMD32 (Prop_ramd32_RADR0_O)	(r) 0.150	4.981	Site: SLICE_X42Y30	p/m_regs/r_reg_r1_0_31_0_5/RAMA/O
net (fo=3, routed)	0.880	5.861		p/m_regs/w_rdata10[0]
LUT2 (Prop_lut2_I0_O)	(r) 0.328	6.189	Site: SLICE_X44Y29	p/m_regs/r_led[3]_i_5/O
net (fo=1, routed)	0.616	6.805		p/m_regs/w_rrs1[0]
CARRY4 (Prop_carry4_DI[0]_O[3])	(r) 0.633	7.438	Site: SLICE_X45Y29	p/m_regs/r_led_reg[3]_i_1/O[3]
net (fo=2050, routed)	5.031	12.469		p/m_dmem/cm_ram_reg_256_511_26_26/A1
RAMS64E (Prop_rams64e_ADR1_O)	(r) 0.306	12.775	Site: SLICE_X54Y82	p/m_dmem/cm_ram_reg_256_511_26_26/RAMS64E_D/O
net (fo=1, routed)	0.000	12.775		p/m_dmem/cm_ram_reg_256_511_26_26/0D
MUXF7 (Prop_muxf7_I0_O)	(r) 0.241	13.016	Site: SLICE_X54Y82	p/m_dmem/cm_ram_reg_256_511_26_26/F7.B/O
net (fo=1, routed)	0.000	13.016		p/m_dmem/cm_ram_reg_256_511_26_26/00
MUXF8 (Prop_muxf8_I0_O)	(r) 0.098	13.114	Site: SLICE_X54Y82	p/m_dmem/cm_ram_reg_256_511_26_26/F8/O
net (fo=1, routed)	1.061	14.175		p/m_dmem/cm_ram_reg_256_511_26_26_n_0
LUT6 (Prop_lut6_I3_O)	(r) 0.319	14.494	Site: SLICE_X56Y74	p/m_dmem/r_reg_r1_0_31_24_29_i_33/O
net (fo=1, routed)	0.000	14.494		p/m_dmem/r_reg_r1_0_31_24_29_i_33_n_0
MUXF7 (Prop_muxf7_I0_O)	(r) 0.209	14.703	Site: SLICE_X56Y74	p/m_dmem/r_reg_r1_0_31_24_29_i_14/O
net (fo=1, routed)	2.199	16.902		p/m_imem/r_reg_r1_0_31_24_29_4
LUT6 (Prop_lut6_I5_O)	(r) 0.297	17.199	Site: SLICE_X48Y39	p/m_imem/r_reg_r1_0_31_24_29_i_4/O
net (fo=2, routed)	0.719	17.918		p/m_regs/r_reg_r2_0_31_24_29/DIB0
RAMD32			Site: SLICE_X46Y36	p/m_regs/r_reg_r2_0_31_24_29/RAMB/I
Arrival Time		17.918		



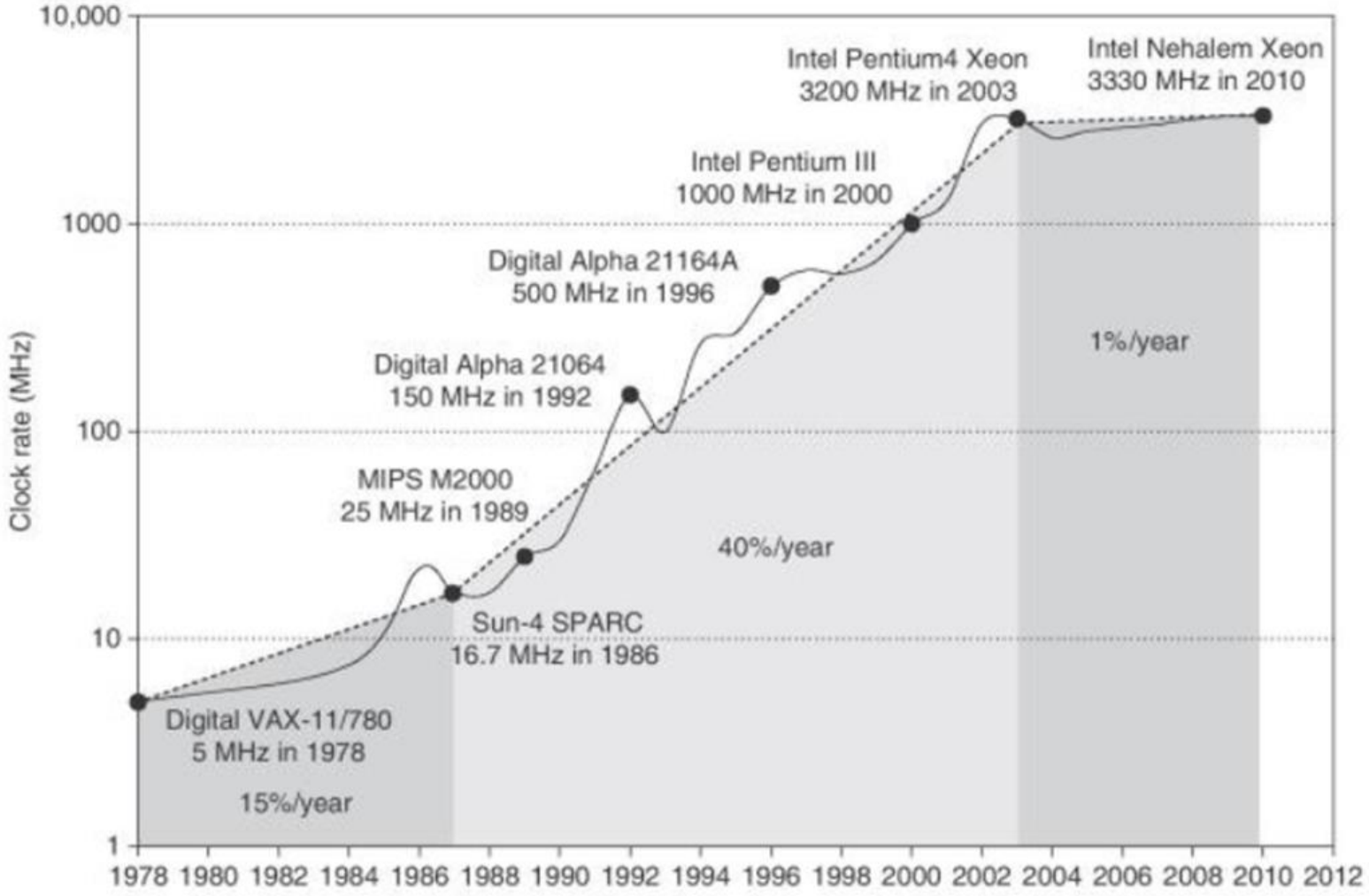
m_proc07 ベースラインのプロセッサ（シングルサイクル）

- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), メモリアクセス(MEM), ライトバック(WB) の処理をおこなうadd, addi, sll, srl, lw, sw, beq, bne命令に対応したプロセッサ





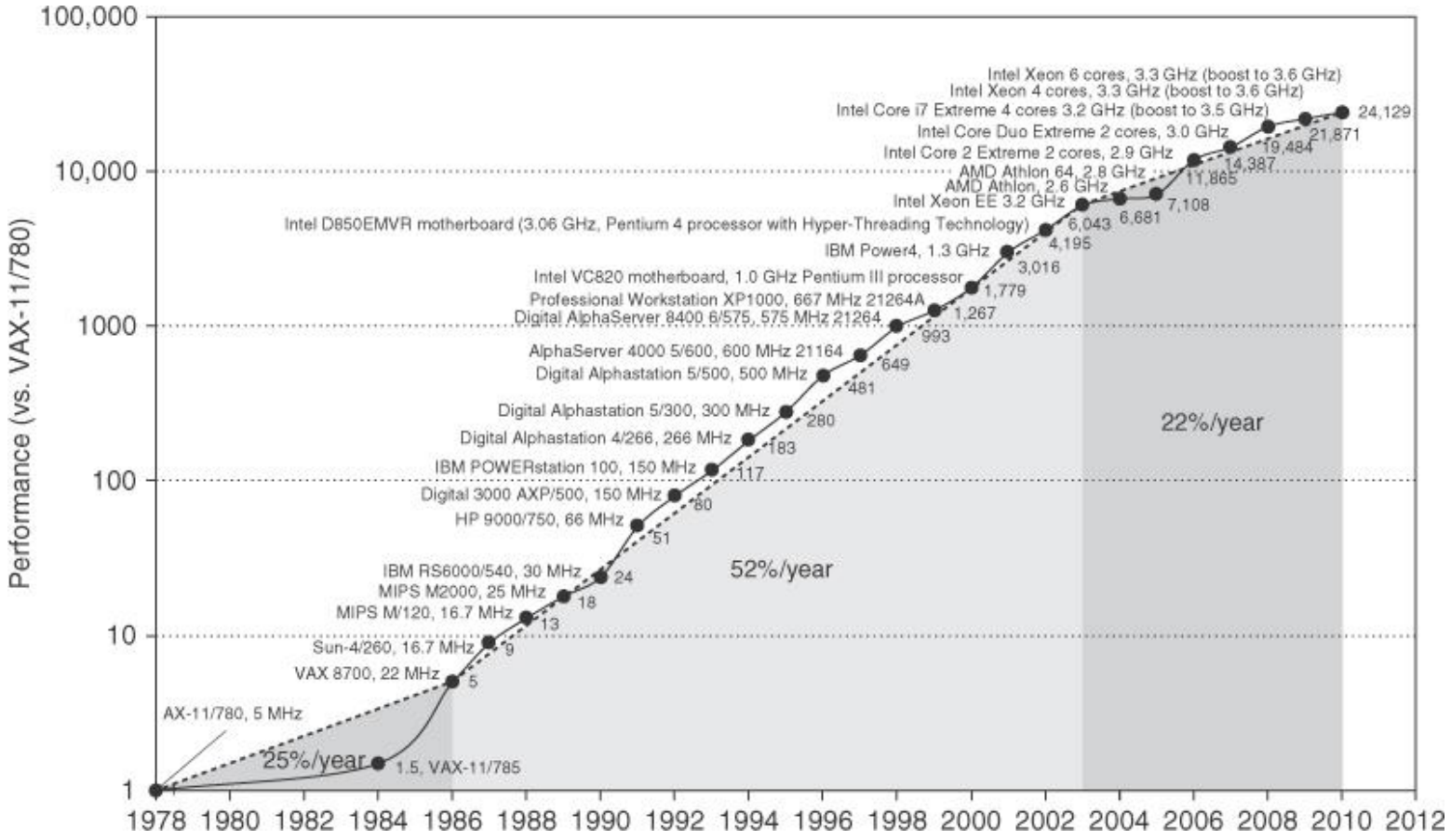
Growth in clock rate of microprocessors



From CAQA 5th edition

Growth in processor performance

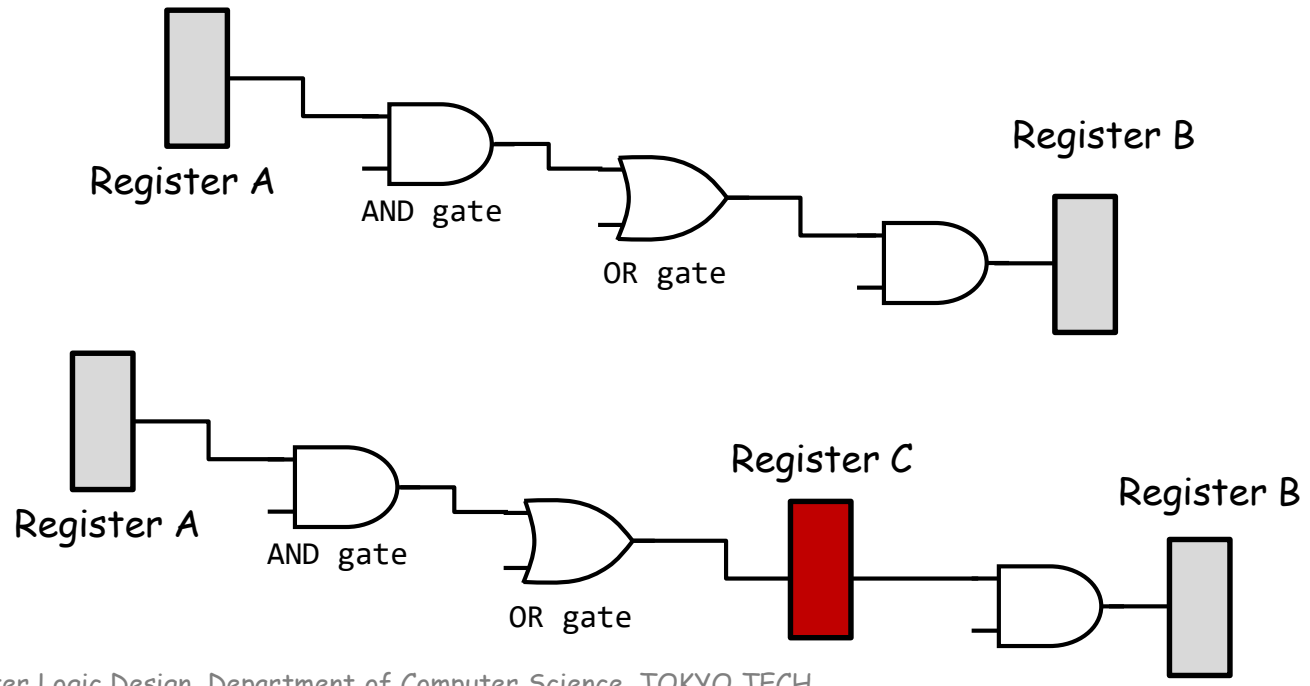
- Performance = f x IPC
  - f: frequency (clock rate)
  - IPC: retired machine Instructions Per Cycle



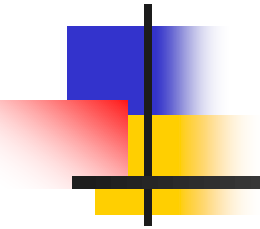
From CAQA 5th edition

Clock rate is mainly determined by

- Switching speed of gates (transistors)
- **The number of levels of gates**
  - The maximum number of gates cascaded in series in any combinational logics.
  - In this example, the number of levels of gates is 3.
- Wiring delay and fanout



Department of Computer Science
Course number: CSC.T341



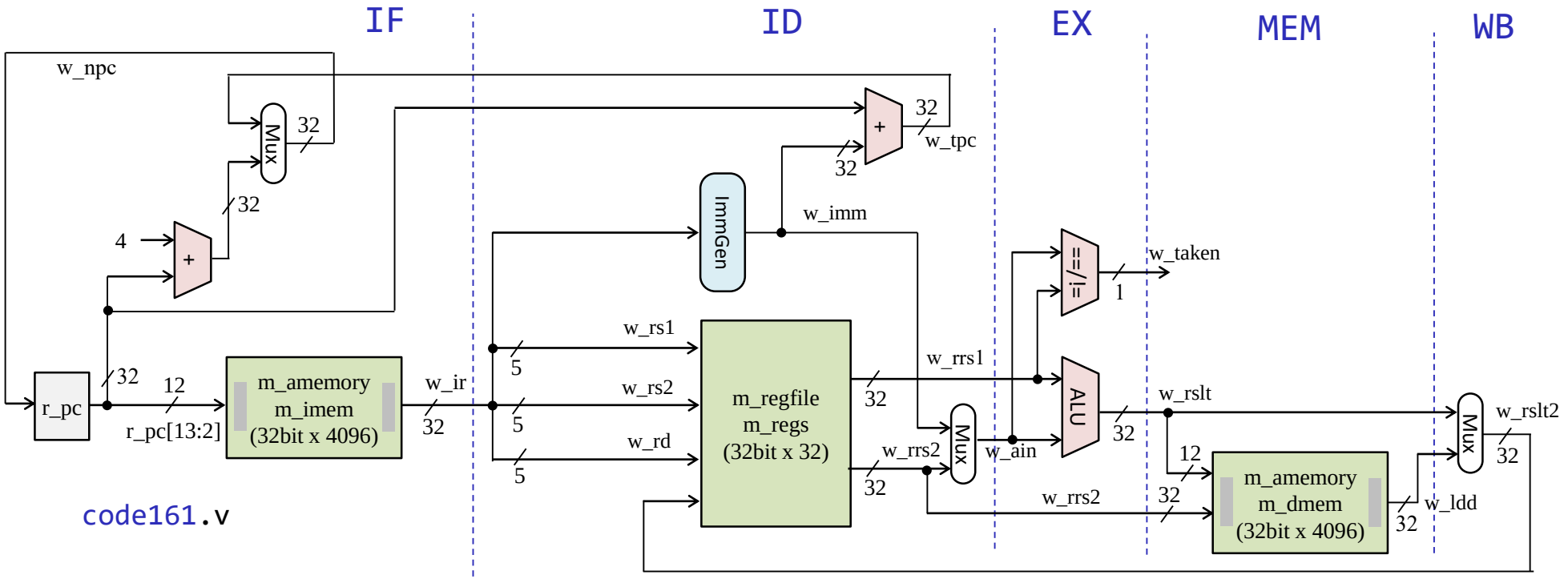
コンピュータ論理設計
Computer Logic Design

10. マルチサイクルプロセッサの設計と実装
Design and Implementation of a Multi-cycle Processor

吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
講義：月曜日 10:45-12:25, 木曜日 10:45-12:25

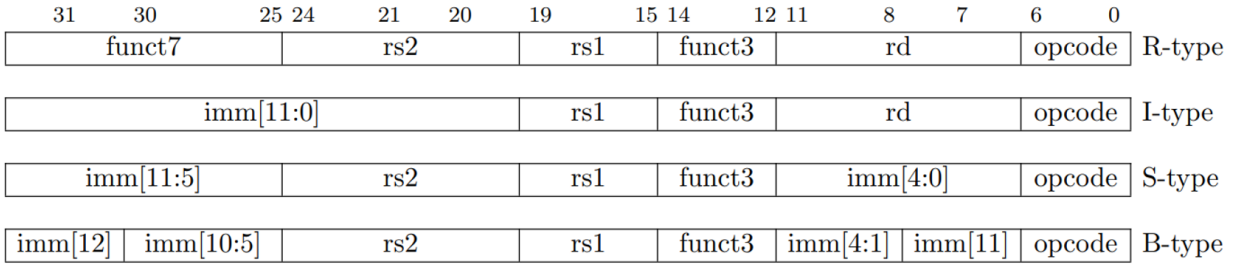
m_proc07 ベースラインのプロセッサ（シングルサイクル）

- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), メモリアクセス(MEM), ライトバック(WB) の処理をおこなうadd, addi, sll, srl, lw, sw, beq, bne命令に対応したプロセッサ



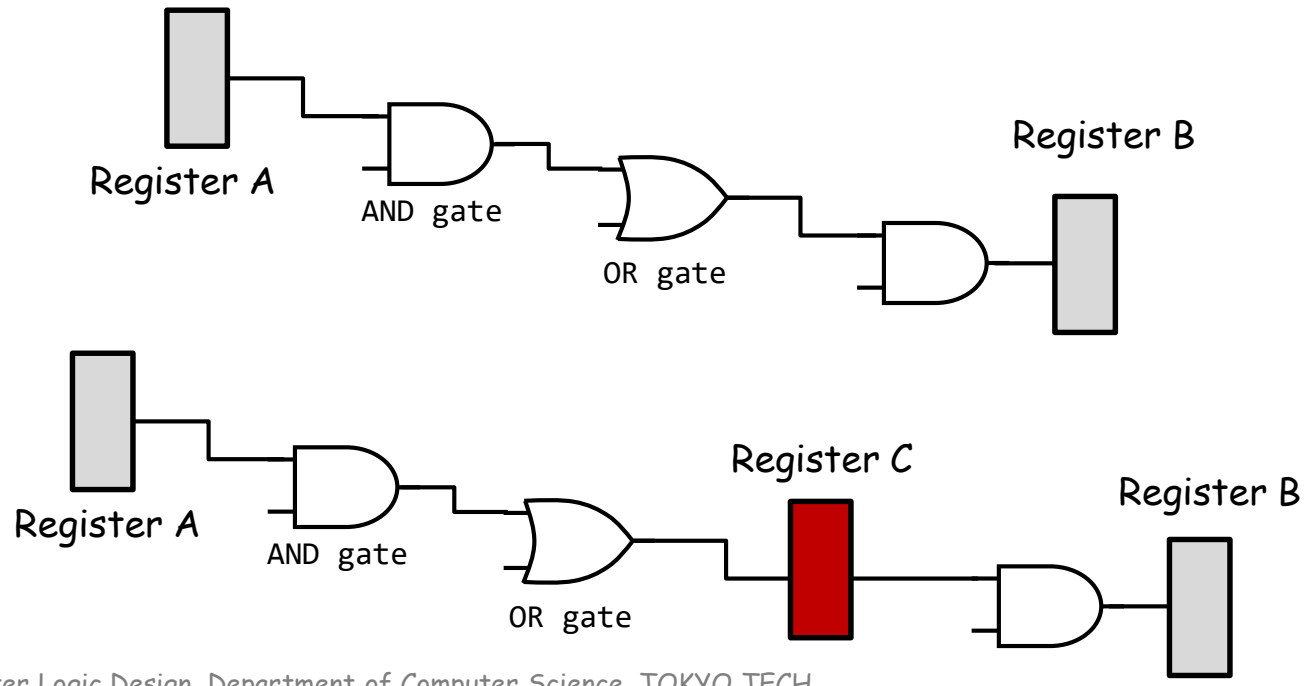
code161.v

f = 60MHz
IPC = 1.000
Perf = 60.00



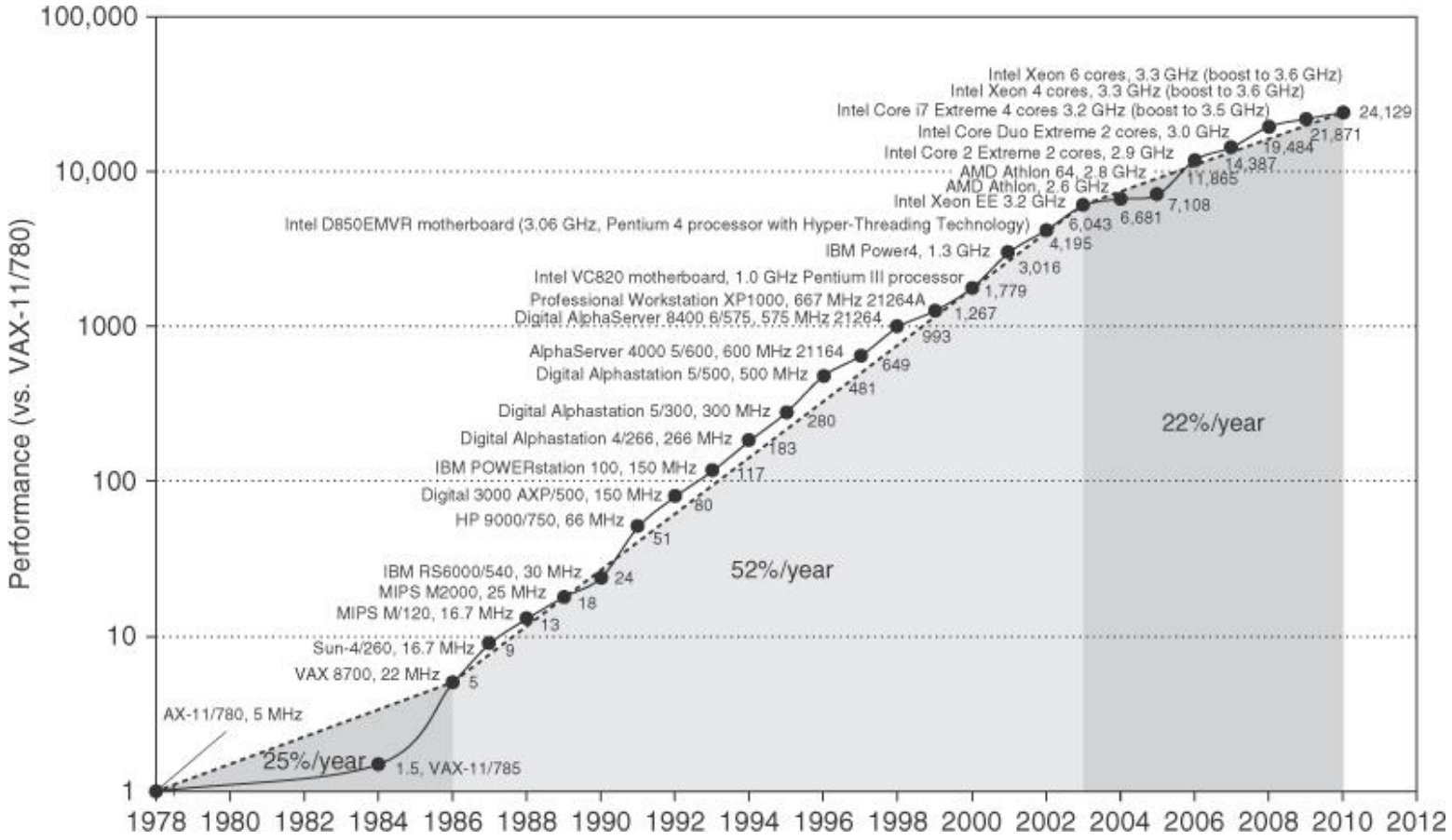
Clock rate is mainly determined by

- Switching speed of gates (transistors)
- **The number of levels of gates**
  - The maximum number of gates cascaded in series in any combinational logics.
  - In this example, the number of levels of gates is 3.
- Wiring delay and fanout



Growth in processor performance

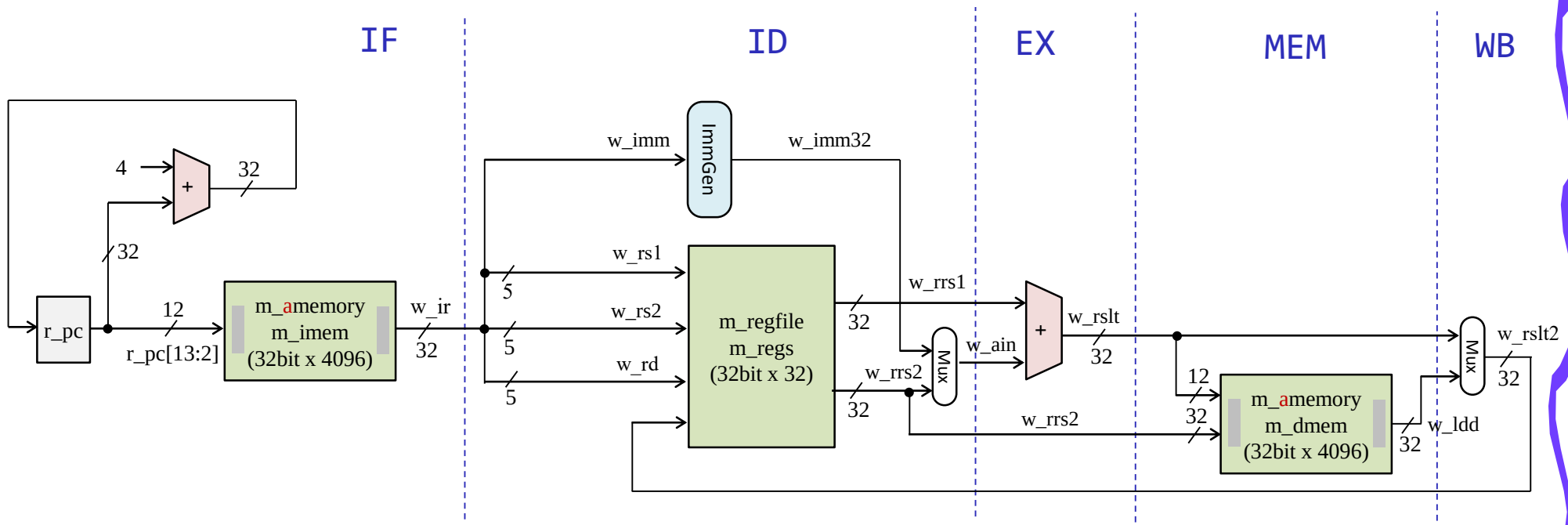
- Performance = f x IPC
  - f: frequency (clock rate)
  - IPC: retired machine Instructions Per Cycle

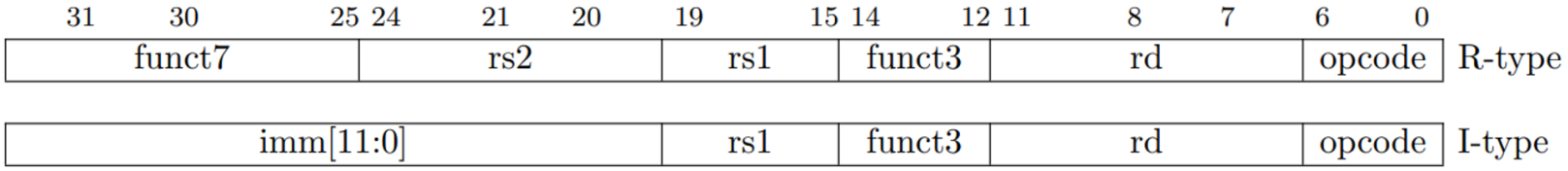


From CAQA 5th edition

m_proc05 add, addi, lw, sw を処理するプロセッサ

- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), メモリアクセス(MEM), ライトバック(WB) の処理をおこなうadd, addi, lw, sw命令に対応したプロセッサ



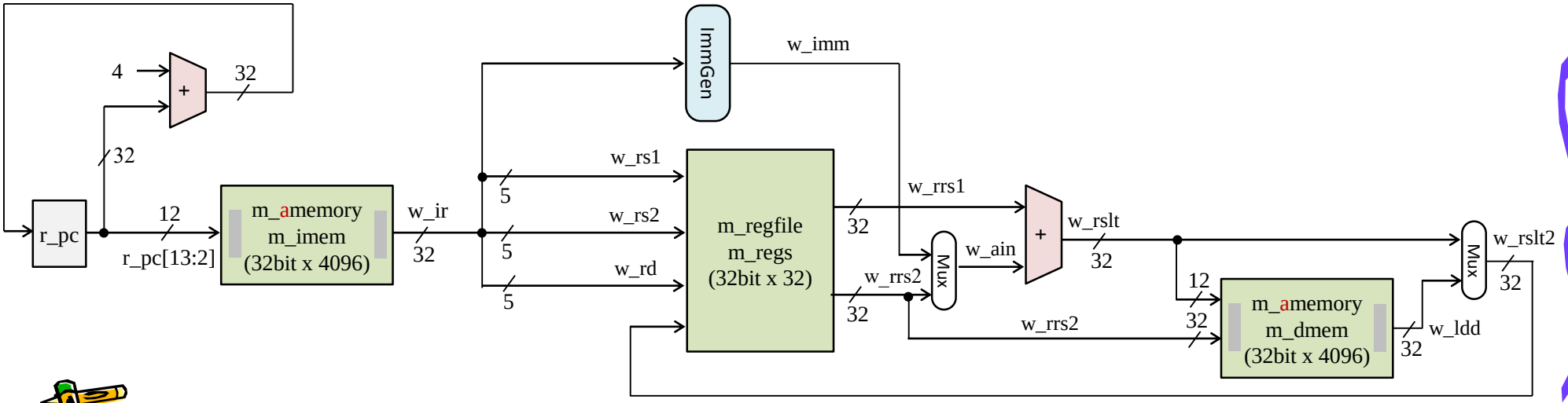


m_proc05 add, addi, lw, sw を処理するシングルサイクル版

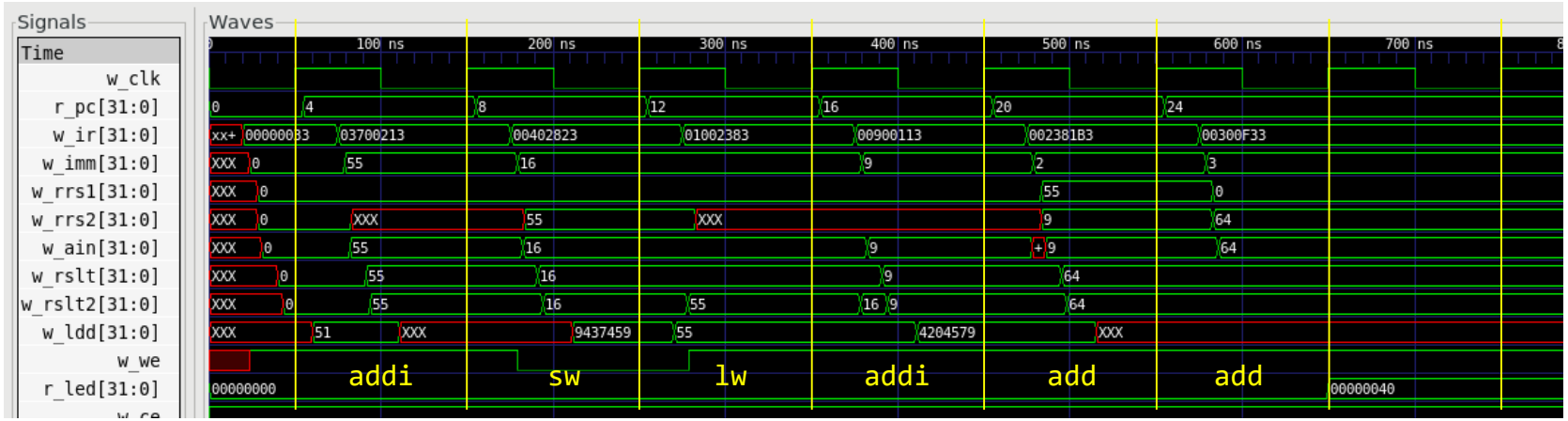
```
module m_proc05 (w_clk, w_ce, w_led);
  input  wire w_clk, w_ce;
  output wire [31:0] w_led;

  reg [31:0] r_pc = 0;
  wire [31:0] w_ir;
  wire [4:0]  w_op5 = w_ir[6:2];
  wire [4:0]  w_rs1 = w_ir[19:15];
  wire [4:0]  w_rs2 = w_ir[24:20];
  wire [4:0]  w_rd  = w_ir[11:7];
```

code171.v

```
  wire #4 w_we = w_ce & (w_op5==5'b01100 || w_op5==5'b00100 || w_op5==5'b00000);
  wire [31:0] w_imm, w_rrs1, w_rrs2, w_ain, w_rslt, w_ldd, w_rslt2;

  m_amemory m_imem (w_clk, r_pc[13:2], 1'd0, 32'd0, w_ir);
  m_immgen m_immgen0 (w_ir, w_imm);
  m_regfile m_regs (w_clk, w_rs1, w_rs2, w_rd, w_we, w_rslt2, w_rrs1, w_rrs2);
  assign #3 w_ain = (w_op5==5'b01100) ? w_rrs2 : w_imm;
  assign #9 w_rslt = w_rrs1 + w_ain;
  m_amemory m_dmem (w_clk, w_rslt[13:2], (w_op5==5'b01000), w_rrs2, w_ldd);
  assign #3 w_rslt2 = (w_op5==5'b00000) ? w_ldd : w_rslt;
  always @(posedge w_clk) #5 if(w_ce & r_pc!=24) r_pc <= r_pc + 4;

  reg [31:0] r_led = 0;
  always @(posedge w_clk) if(w_ce & w_we & w_rd==30) r_led <= w_rslt2;
  assign w_led = r_led;
endmodule
```

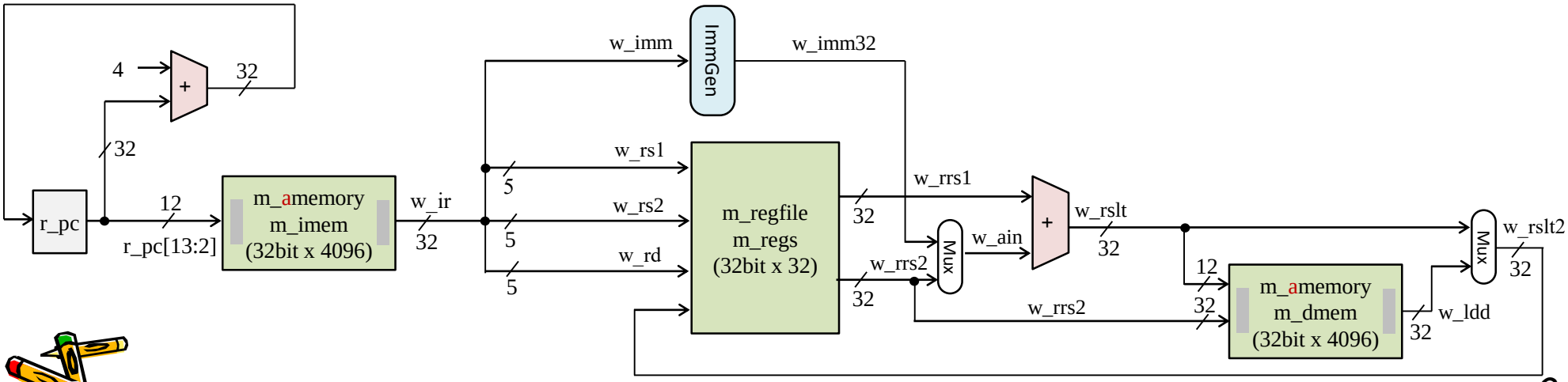


m_proc05 add, addi, lw, sw を処理するシングルサイクル版



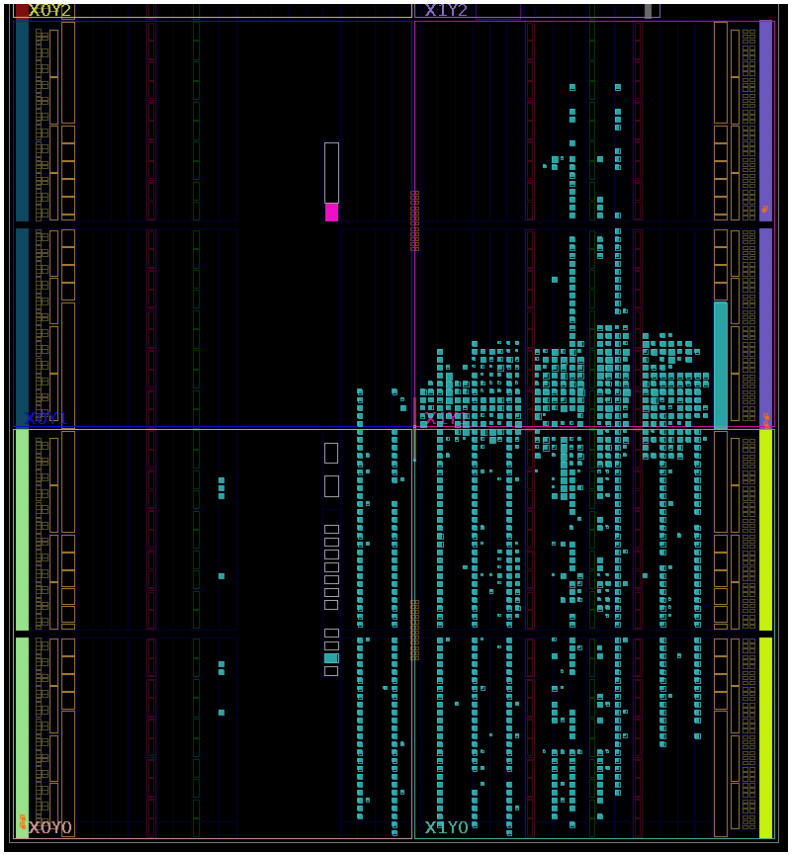

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //    add  x0, x0, x0  // NOP
cm_ram[1]={12'd55,     5'd0, 3'b000, 5'd4, 7'b0010011}; //    addi x4, x0, 55  // x4 = 55
cm_ram[2]={7'd0, 5'd4, 5'd0, 3'b010, 5'd16,7'b0100011}; //    sw   x4, 16(x0)  // m[16] = x4
cm_ram[3]={12'd16,     5'd0, 3'b010, 5'd7, 7'b0000011}; //    lw   x7, 16(x0)  // x7 = m[16]
cm_ram[4]={12'd9,      5'd0, 3'b000, 5'd2, 7'b0010011}; //    addi x2, x0, 9   // x2 = 9
cm_ram[5]={7'd0, 5'd2, 5'd7, 3'b000, 5'd3, 7'b0110011}; //    add  x3, x7, x2  // x3 = x7 + x2
cm_ram[6]={7'd0, 5'd3, 5'd0, 3'b000, 5'd30,7'b0110011}; //    add  x30,x0, x3  // led = x3
cm_ram[7]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b1100011}; // L: beq  x0, x0, L
```

code171.v
program.txt



m_proc05 add, addi, lw, sw を処理するシングルサイクル版

code171.v
main15.xdc
50MHz clock freq.



m_proc05 シングルサイクル版

50MHz

WNS	TNS	LUT	FF	LUTRAM	BRAM
		2458	**66**	**2164**	**0**
3.413	0.000	3080	1215	2184	0

60MHz

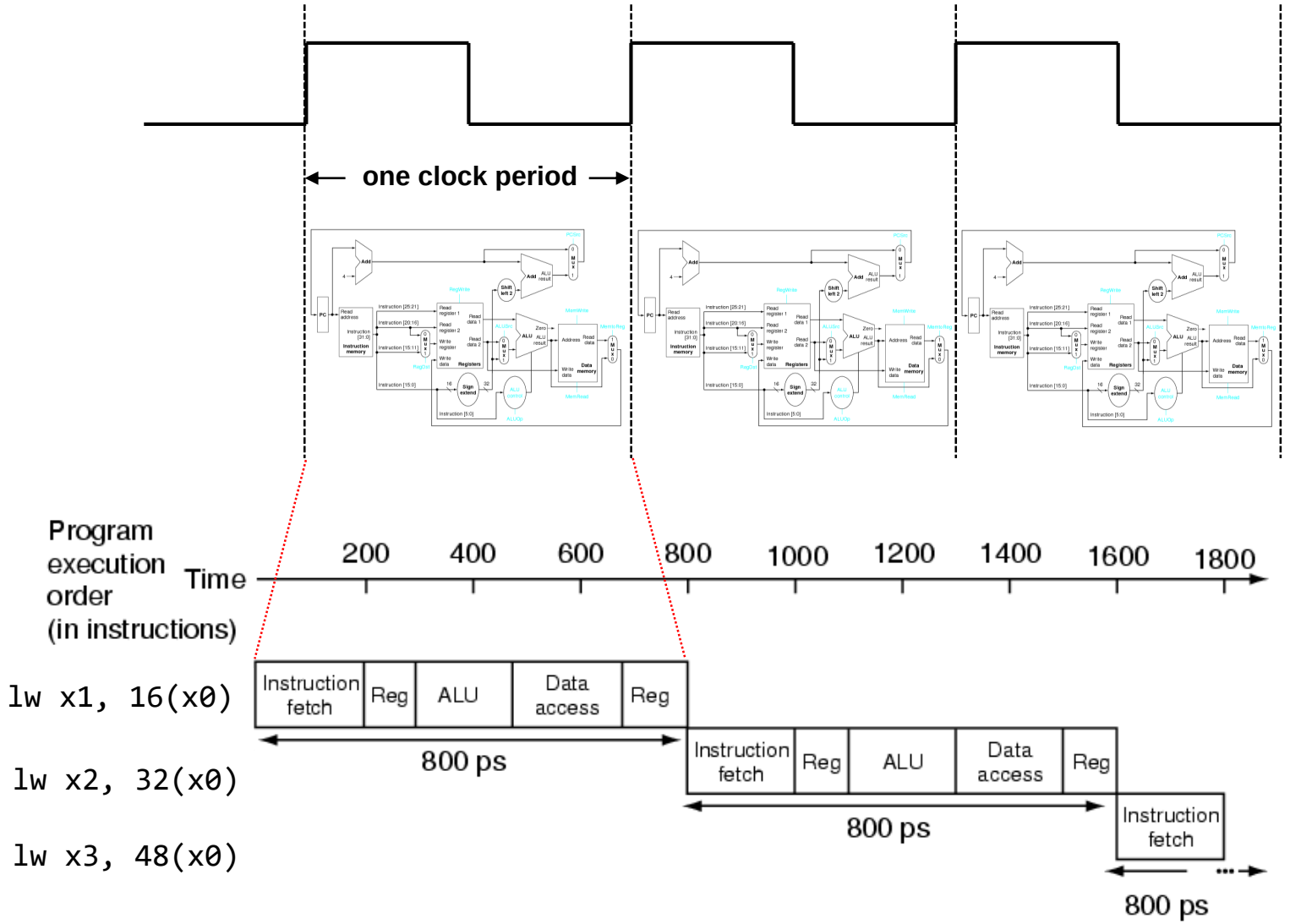
WNS	TNS	LUT	FF	LUTRAM	BRAM
		2458	**66**	**2164**	**0**
0.361	0.000	3081	1215	2184	0

70MHz

WNS	TNS	LUT	FF	LUTRAM	BRAM
		2458	**66**	**2164**	**0**
-0.023	-0.023	3081	1215	2184	0

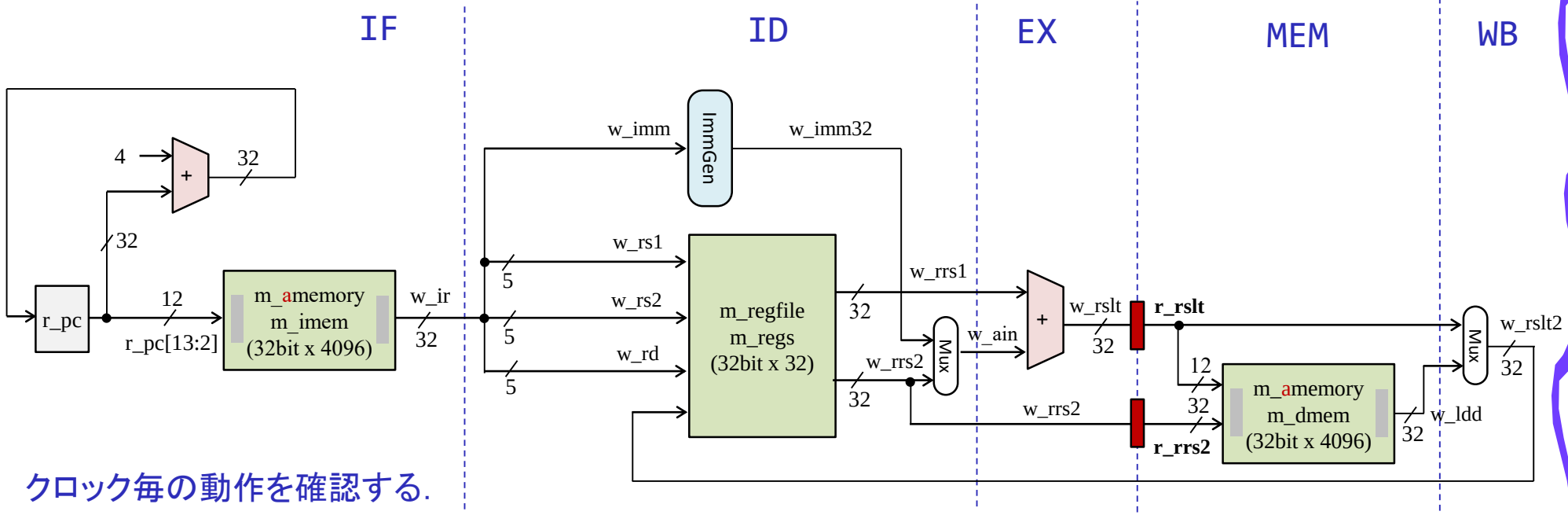
10MHz 単位で動作周波数を変化させて測定した場合の最高の動作周波数は 60MHz

Single Cycle Processor



m_proc08 (multi-cycle processor) マルチサイクル

- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX) を1サイクル,
  メモリアクセス(MEM), ライトバック(WB) の処理を1サイクルで処理するマルチサイクルのプロセッサ
- 2サイクルで1命令を処理する IPC (instructions per cycle) = 0.5 のプロセッサ, add, addi, lw, sw をサポート



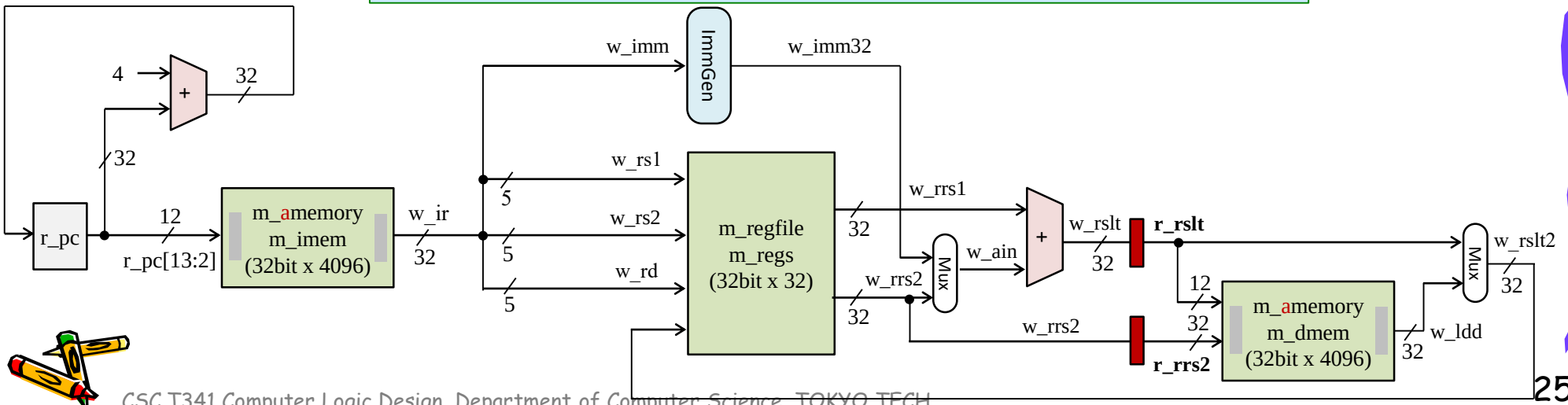
クロック毎の動作を確認する.

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //    add  x0, x0, x0  // NOP
cm_ram[1]={12'd55,     5'd0, 3'b000, 5'd4, 7'b0010011}; //    addi x4, x0, 55  // x4 = 55
cm_ram[2]={7'd0, 5'd4, 5'd0, 3'b010, 5'd16,7'b0100011}; //    sw   x4, 16(x0)  // m[16] = x4
cm_ram[3]={12'd16,     5'd0, 3'b010, 5'd7, 7'b0000011}; //    lw   x7, 16(x0)  // x7 = m[16]
cm_ram[4]={12'd9,      5'd0, 3'b000, 5'd2, 7'b0010011}; //    addi x2, x0, 9   // x2 = 9
cm_ram[5]={7'd0, 5'd2, 5'd7, 3'b000, 5'd3, 7'b0110011}; //    add  x3, x7, x2  // x3 = x7 + x2
cm_ram[6]={7'd0, 5'd3, 5'd0, 3'b000, 5'd30,7'b0110011}; //    add  x30,x0, x3  // led = x3
cm_ram[7]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b1100011}; // L: beq  x0, x0, L
```

m_proc08 (multi-cycle processor) マルチサイクル


```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //    add  x0, x0, x0  // NOP
cm_ram[1]={12'd55,     5'd0, 3'b000, 5'd4, 7'b0010011}; //    addi x4, x0, 55  // x4 = 55
cm_ram[2]={7'd0, 5'd4, 5'd0, 3'b010, 5'd16,7'b0100011}; //    sw   x4, 16(x0)  // m[16] = x4
cm_ram[3]={12'd16,     5'd0, 3'b010, 5'd7, 7'b0000011}; //    lw   x7, 16(x0)  // x7 = m[16]
cm_ram[4]={12'd9,      5'd0, 3'b000, 5'd2, 7'b0010011}; //    addi x2, x0, 9   // x2 = 9
cm_ram[5]={7'd0, 5'd2, 5'd7, 3'b000, 5'd3, 7'b0110011}; //    add  x3, x7, x2  // x3 = x7 + x2
cm_ram[6]={7'd0, 5'd3, 5'd0, 3'b000, 5'd30,7'b0110011}; //    add  x30,x0, x3  // led = x3
cm_ram[7]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b1100011}; // L: beq  x0, x0, L
```

code172.v
program.txt

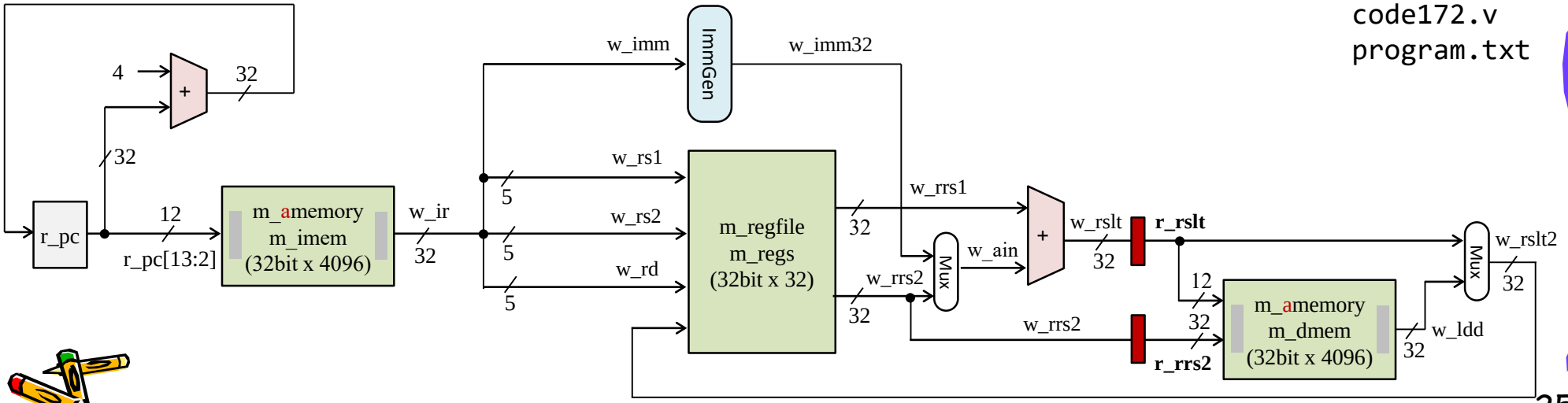


m_proc08 (multi-cycle processor) マルチサイクル

```
module m_proc08 (w_clk, w_ce, w_led);
  input  wire w_clk, w_ce;
  output wire [31:0] w_led;

  reg [31:0] r_pc = 0;
  wire [31:0] w_ir;
  wire [4:0]  w_op5 = w_ir[6:2];
  wire [4:0]  w_rs1 = w_ir[19:15];
  wire [4:0]  w_rs2 = w_ir[24:20];
  wire [4:0]  w_rd  = w_ir[11:7];

  reg [31:0] r_rslt, r_rrs2;

  reg r_state = 0;
  always @(posedge w_clk) #5
    if(w_ce) r_state <= r_state + 1;
```

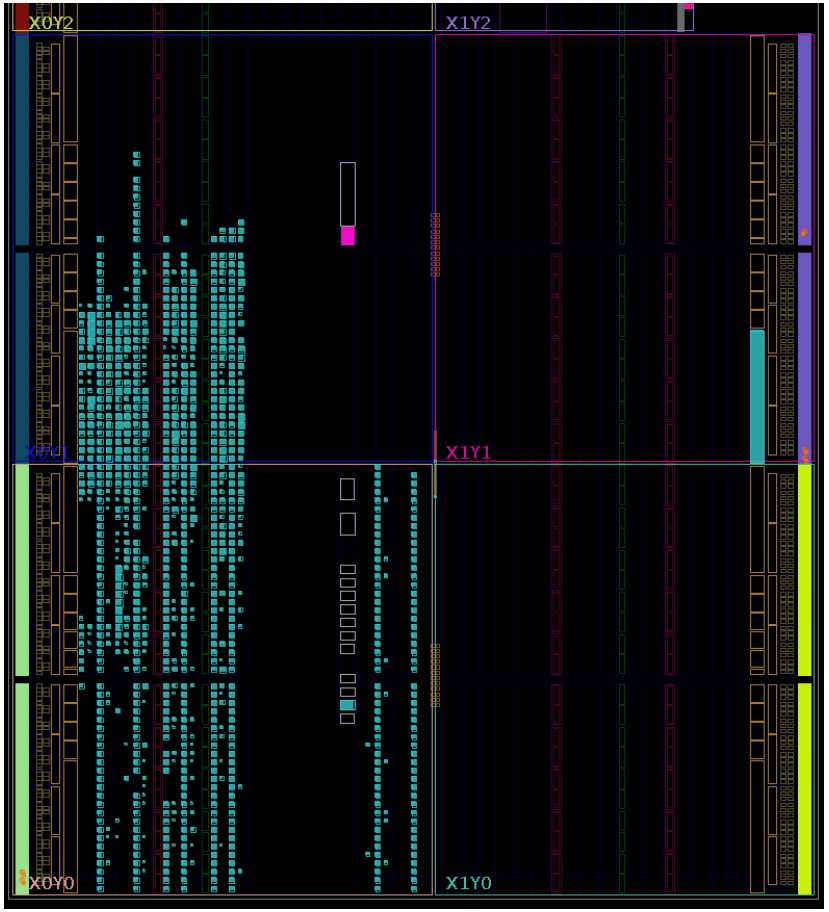
```
  wire #4 w_we = w_ce & (w_op5==5'b01100 || w_op5==5'b00100 || w_op5==5'b00000);

  wire [31:0] w_imm, w_rrs1, w_rrs2, w_ain, w_rslt, w_ldd, w_rslt2;

  always @(posedge w_clk) if(w_ce & r_state==0) r_rslt <= w_rslt;
  always @(posedge w_clk) if(w_ce & r_state==0) r_rrs2 <= w_rrs2;
  m_amemory m_imem (w_clk, r_pc[13:2], 1'd0, 32'd0, w_ir);
  m_immgen m_immgen0 (w_ir, w_imm);
  m_regfile m_regs (w_clk, w_rs1, w_rs2, w_rd, w_we & r_state, w_rslt2, w_rrs1, w_rrs2);
  assign #3 w_ain = (w_op5==5'b01100) ? w_rrs2 : w_imm;
  assign #9 w_rslt = w_rrs1 + w_ain;
  m_amemory m_dmem (w_clk, r_rslt[13:2], (w_op5==5'b01000 & r_state), r_rrs2, w_ldd);
  assign #3 w_rslt2 = (w_op5==5'b00000) ? w_ldd : r_rslt;
  always @(posedge w_clk) #5 if(w_ce & r_state & r_pc!=24) r_pc <= r_pc + 4;

  reg [31:0] r_led = 0;
  always @(posedge w_clk) if(w_ce & w_we & r_state & w_rd==30) r_led <= w_rslt2;
  assign w_led = r_led;
endmodule
```

code172.v
program.txt



m_proc08 (multi-cycle processor) マルチサイクル

code172.v (m_proc08)
main15.xdc
80MHz clock freq.



m_proc05 シングルサイクル版　**Perf = 60 x 1.0 = 60.0**

60MHz

WNS	TNS	LUT	FF	LUTRAM	BRAM
		2458	**66**	**2164**	**0**
0.361	0.000	3081	1215	2184	0

70MHz

WNS	TNS	LUT	FF	LUTRAM	BRAM
		2458	**66**	**2164**	**0**
-0.023	-0.023	3081	1215	2184	0

m_proc08 2段のマルチサイクル版　**Perf = 110 x 0.5 = 55.0**

90MHz

WNS	TNS	LUT	FF	LUTRAM	BRAM
		2461	**219**	**2164**	**0**
1.306	0.000	3084	1368	2184	0

100MHz

WNS	TNS	LUT	FF	LUTRAM	BRAM
		2461	**219**	**2164**	**0**
0.284	0.000	3083	1368	2184	0

110MHz

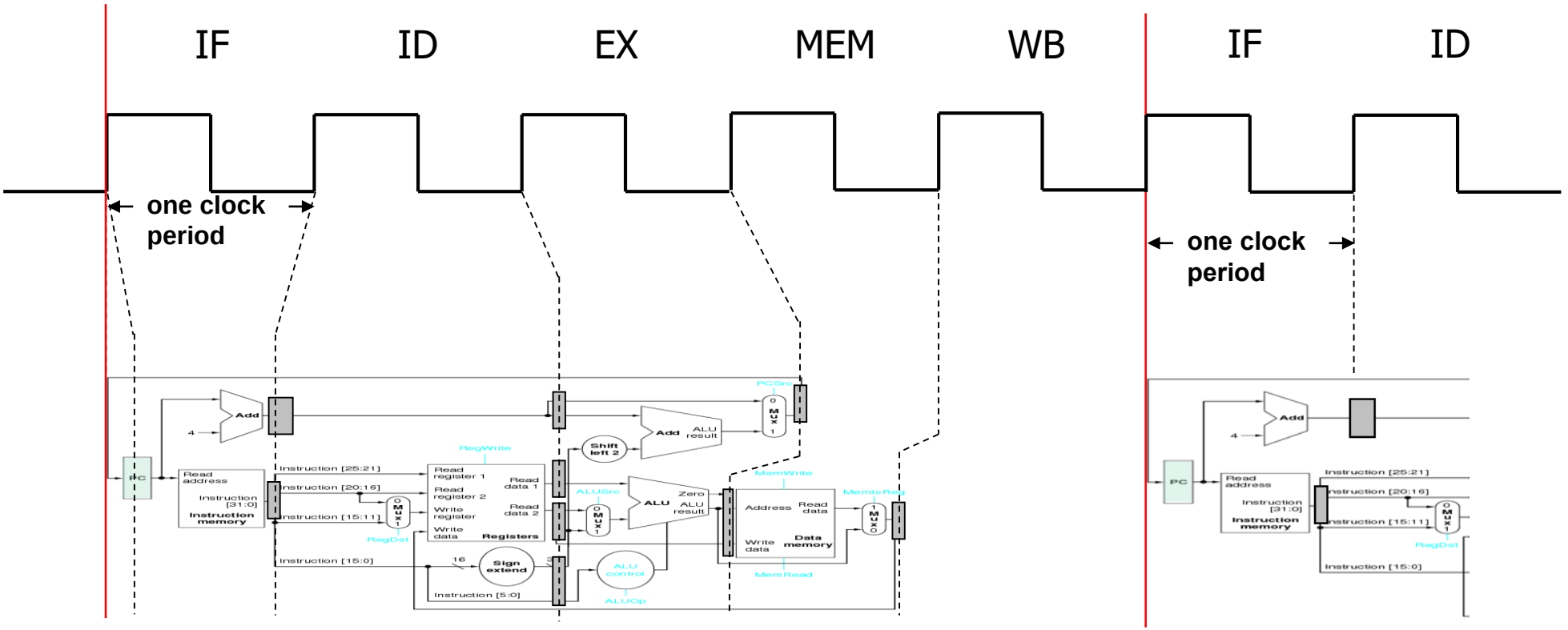
WNS	TNS	LUT	FF	LUTRAM	BRAM
		2461	**219**	**2164**	**0**
0.229	0.000	3084	1368	2184	0

120MHz

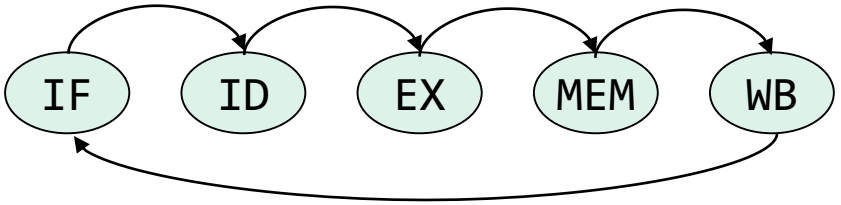
WNS	TNS	LUT	FF	LUTRAM	BRAM
		2461	**219**	**2164**	**0**
-0.027	-0.071	3091	1368	2184	0

10MHz単位で動作周波数を変化させて測定した場合の最高の動作周波数は 110MHz

マルチサイクルのプロセッサ Multi-cycle Processor



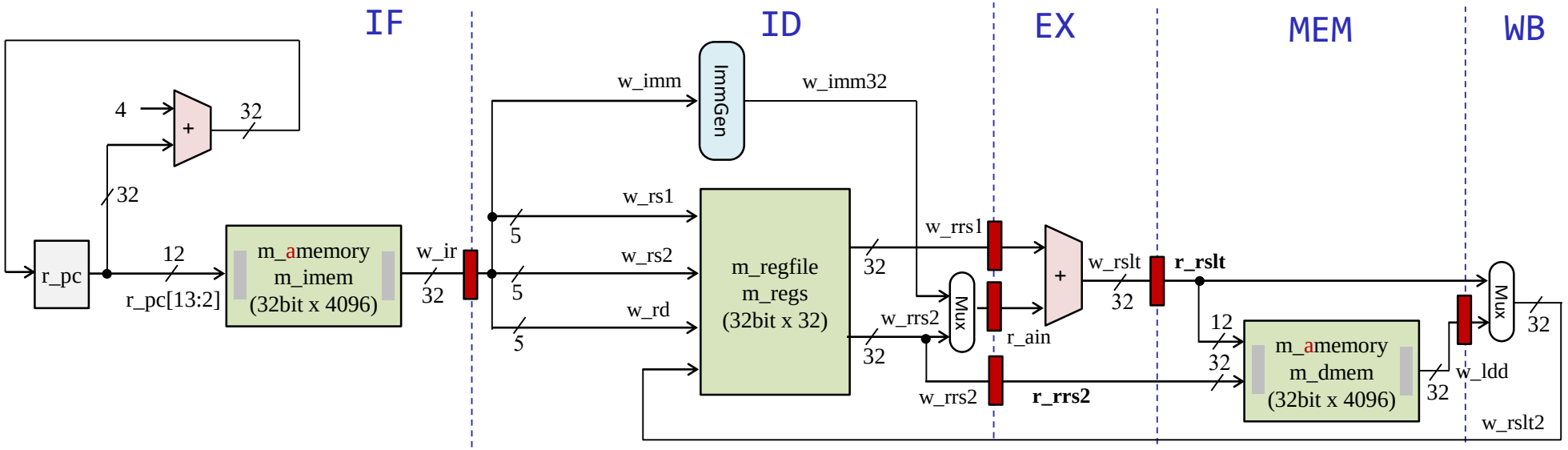
State Machine Diagram



0.1 x 3 + 0.1 x 5 + 0.8 x 4 = 4.0

m_proc09 (multi-cycle processor) マルチサイクル

- 5サイクルで1命令を処理する IPC = 0.2 のマルチサイクルのプロセッサ, add, addi, lw, sw をサポート
- 最も遅延の長い(長い時間を要する)ステップがプロセッサの動作周波数を決める.

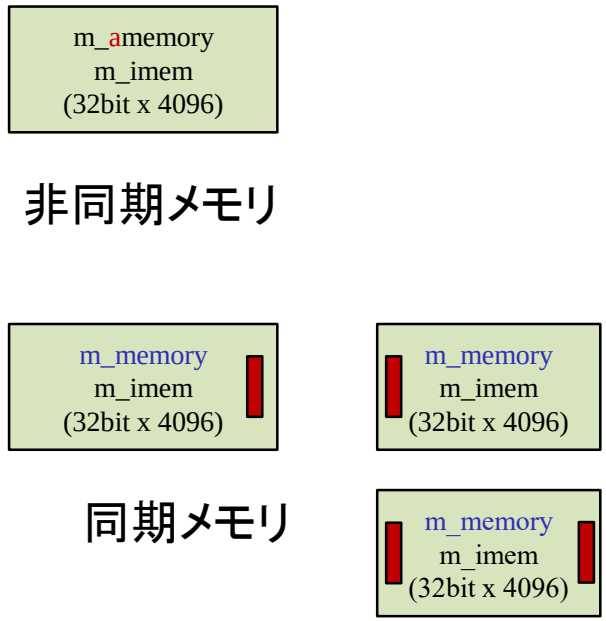


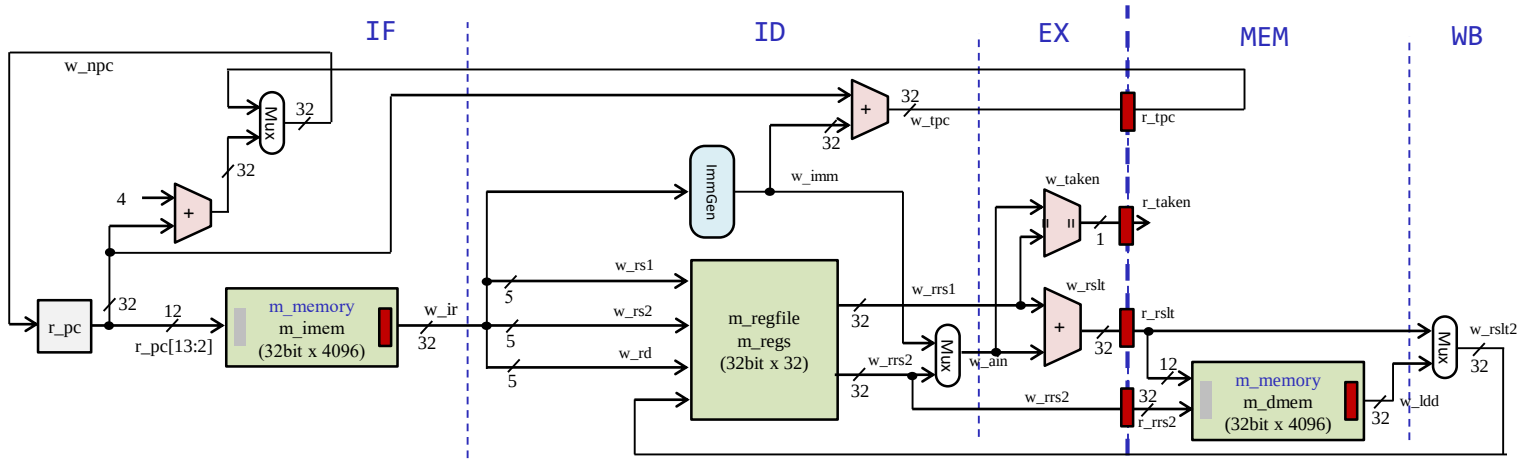
クロック毎の動作を確認する.

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //    add  x0, x0, x0  // NOP
cm_ram[1]={12'd55,     5'd0, 3'b000, 5'd4, 7'b0010011}; //    addi x4, x0, 55  // x4 = 55
cm_ram[2]={7'd0, 5'd4, 5'd0, 3'b010, 5'd16,7'b0100011}; //    sw   x4, 16(x0)  // m[16] = x4
cm_ram[3]={12'd16,     5'd0, 3'b010, 5'd7, 7'b0000011}; //    lw   x7, 16(x0)  // x7 = m[16]
cm_ram[4]={12'd9,      5'd0, 3'b000, 5'd2, 7'b0010011}; //    addi x2, x0, 9   // x2 = 9
cm_ram[5]={7'd0, 5'd2, 5'd7, 3'b000, 5'd3, 7'b0110011}; //    add  x3, x7, x2  // x3 = x7 + x2
cm_ram[6]={7'd0, 5'd3, 5'd0, 3'b000, 5'd30,7'b0110011}; //    add  x30,x0, x3  // led = x3
cm_ram[7]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b1100011}; // L: beq  x0, x0, L
```

非同期(asynchronous)メモリと同期(synchronous)メモリ

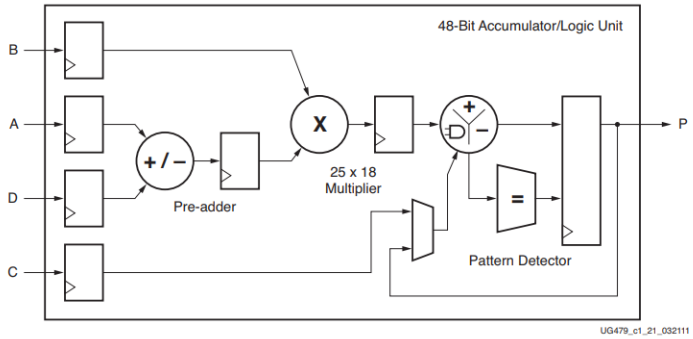
```
module m_amemory (w_clk, w_addr, w_we, w_din, w_dout); // asynchronous memory
  input  wire w_clk, w_we;
  input  wire [11:0] w_addr;
  input  wire [31:0] w_din;
  output wire [31:0] w_dout;
  reg [31:0]         cm_ram [0:4095]; // 4K word (4096 x 32bit) memory
  always @(posedge w_clk) if (w_we) cm_ram[w_addr] <= w_din;
  assign #20 w_dout = cm_ram[w_addr];
`include "program.txt"
endmodule

module m_memory (w_clk, w_addr, w_we, w_din, r_dout); // synchronous memory
  input  wire w_clk, w_we;
  input  wire [11:0] w_addr;
  input  wire [31:0] w_din;
  output reg  [31:0] r_dout;
  reg [31:0]         cm_ram [0:4095]; // 4K word (4096 x 32bit) memory
  always @(posedge w_clk) if (w_we) cm_ram[w_addr] <= w_din;
  always @(posedge w_clk) r_dout <= cm_ram[w_addr];
`include "program.txt"
endmodule
```



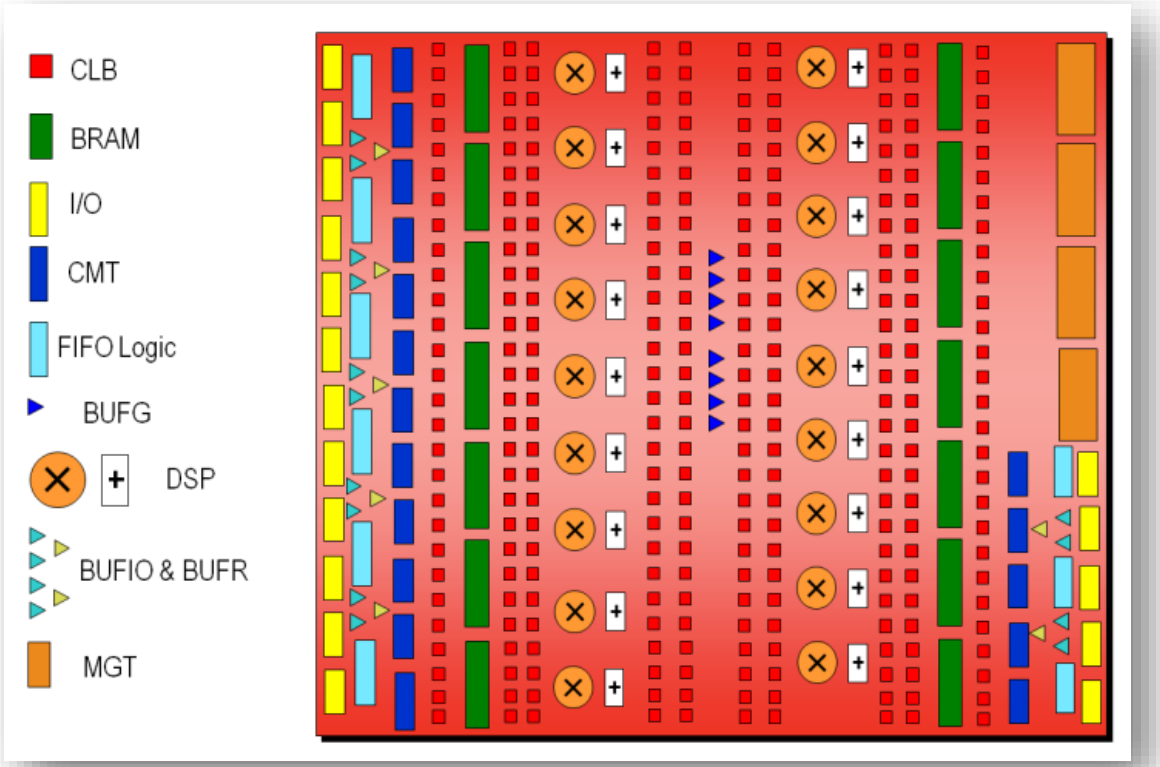


Artix-7 Architecture Overview

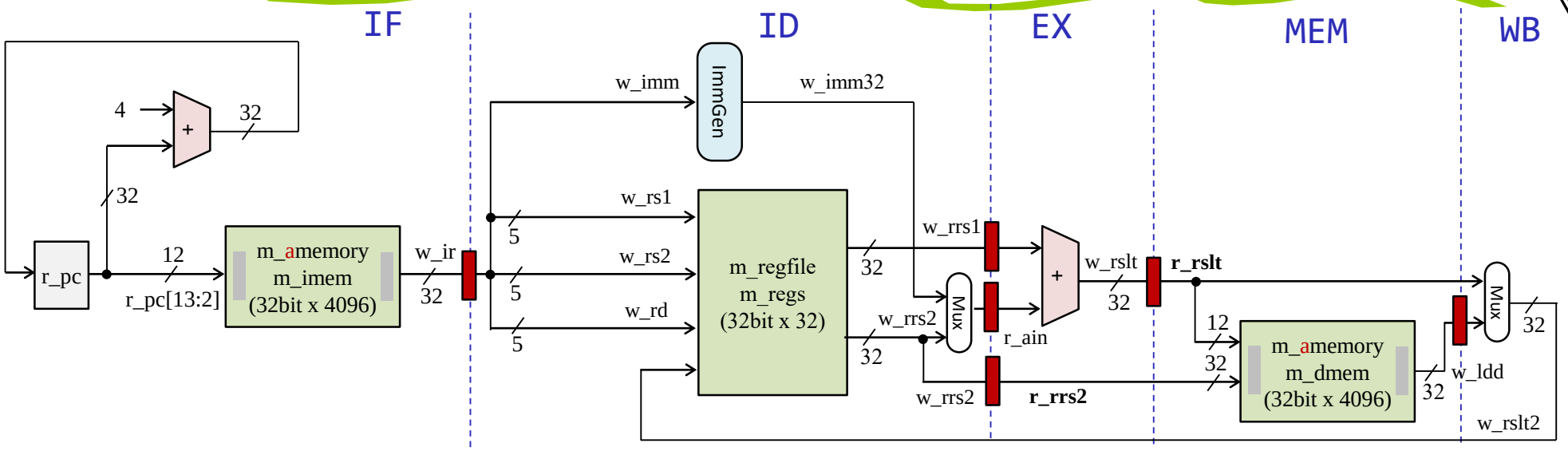
- CLB (Configurable Logic Block)
- BRAM (Block RAM, embedded memory, synchronous memory)
- DSP (Digital Signal Processing)
- CMT (Clock Management Tile)
- Routing fabric



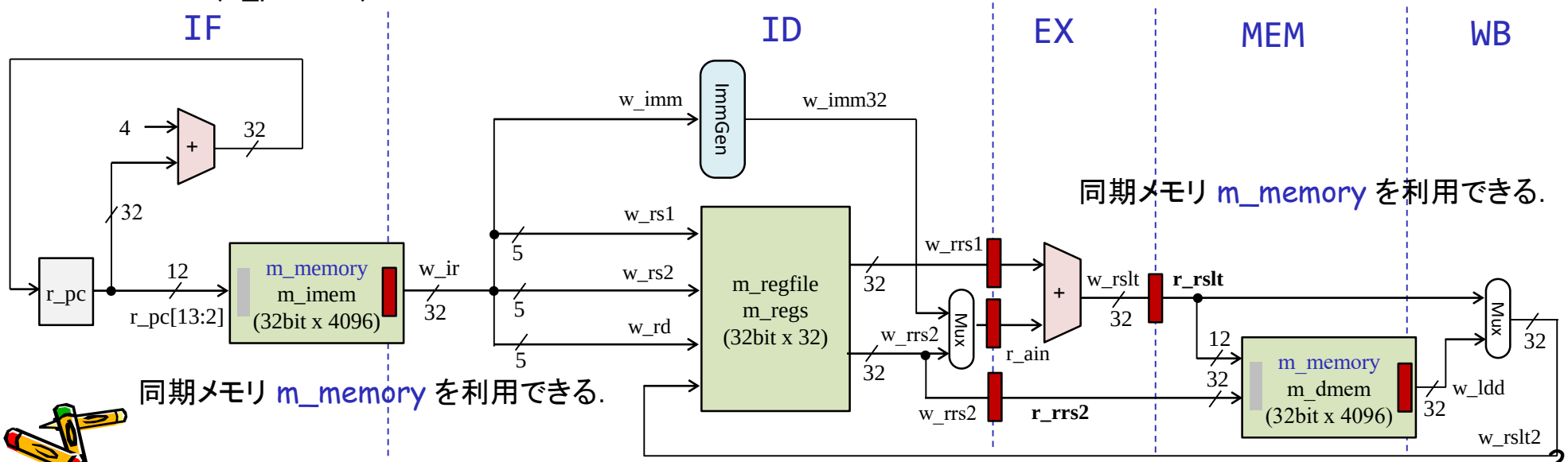
DSP Slice



m_proc10 (multi-cycle processor) マルチサイクル



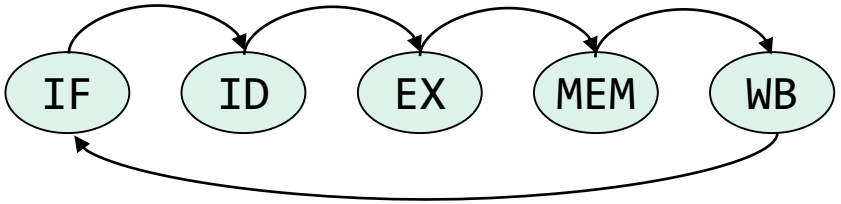
code174.v (m_proc10)



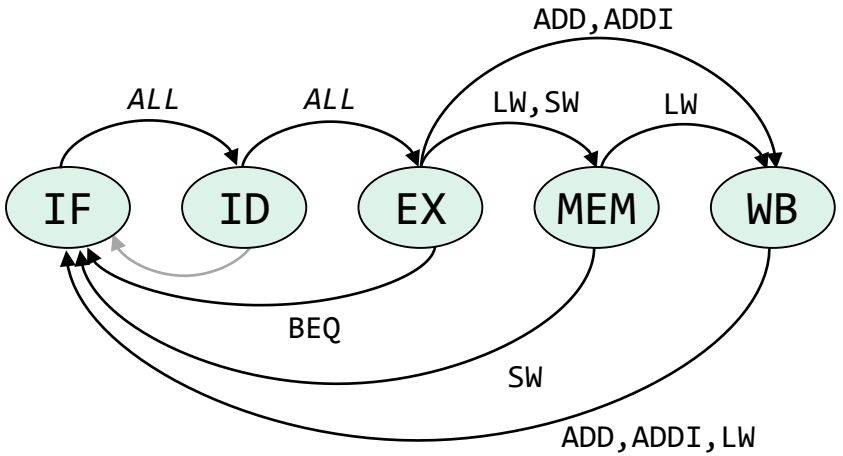
同期メモリ m_memory を利用できる.

同期メモリ m_memory を利用できる.

Optimization for Multi-cycle Processors



Simple State Machine Diagram

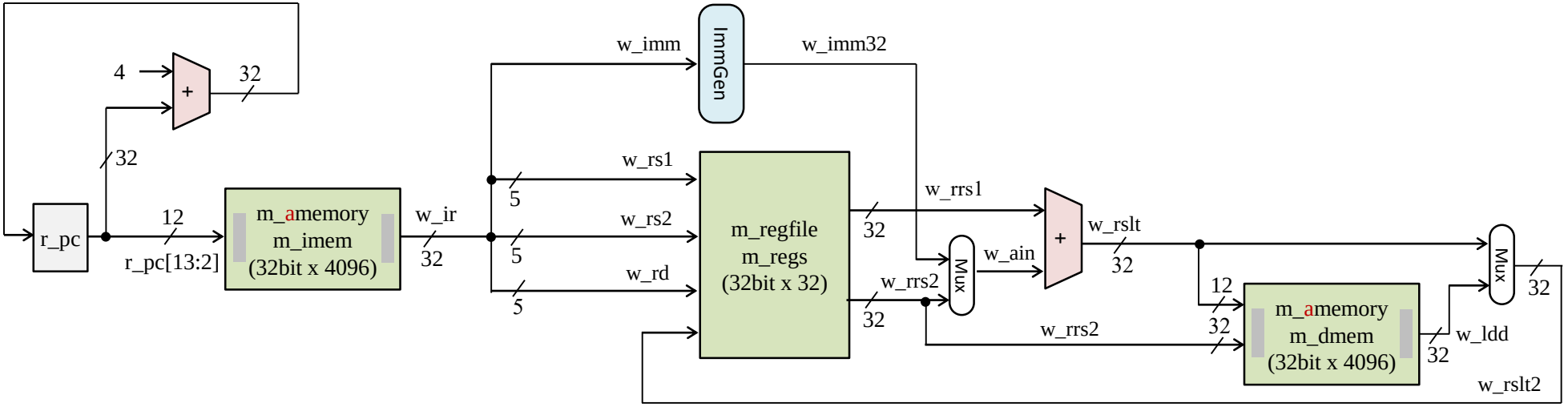


Optimized State Machine Diagram

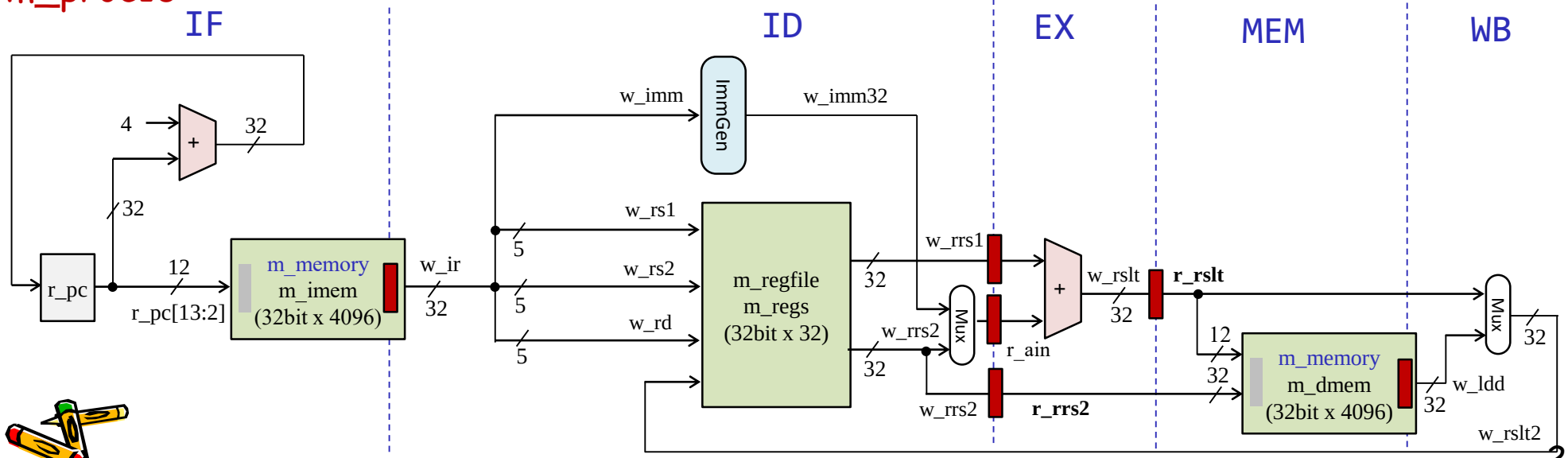
$0.1 \times 3 + 0.1 \times 5 + 0.8 \times 4 = 4.0$

Comparison of the two processor architectures

m_proc05 (code171.v)



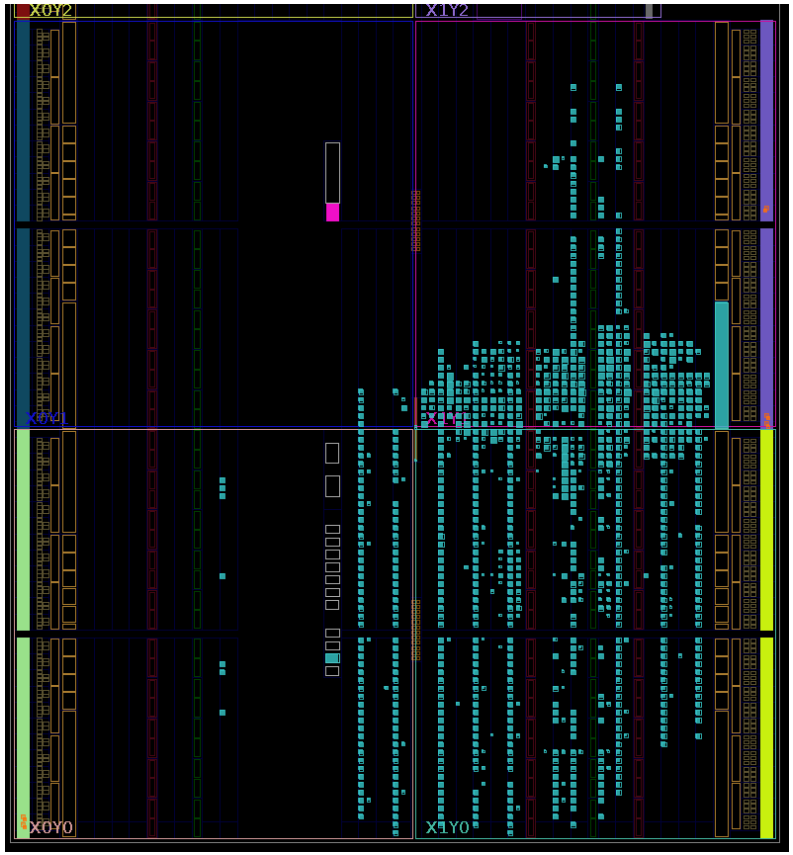
m_proc10

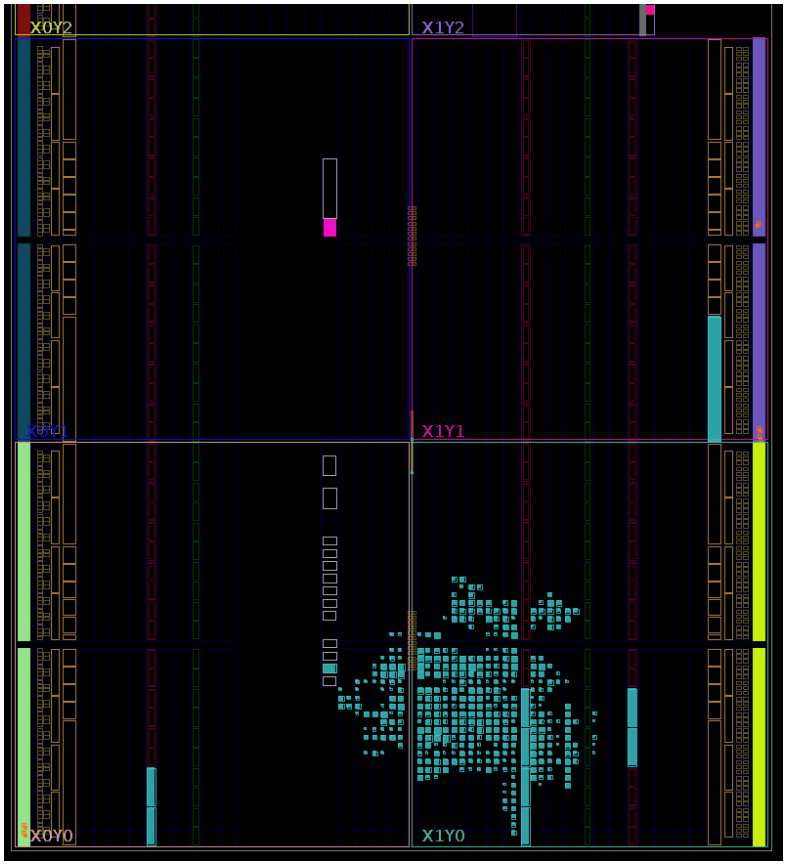


m_proc05 add, addi, lw, sw を処理するシングルサイクル版

code171.v
main15.xdc
50MHz clock freq.

左は，非同期メモリを使うようにm_proc05 を記述して分散メモリとなる場合
右は，同期メモリを使うようにm_proc05 を記述して BRAM となる場合

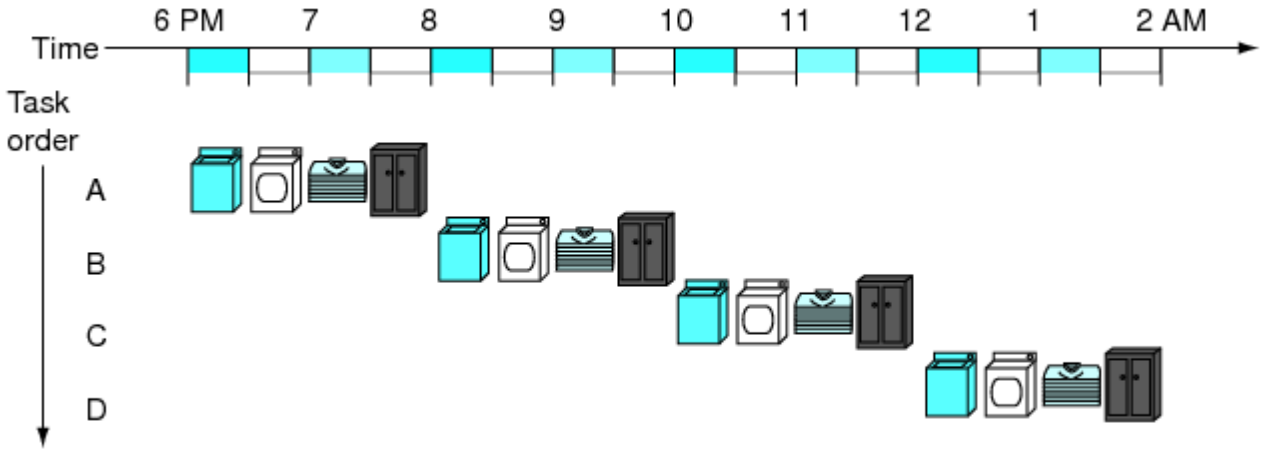




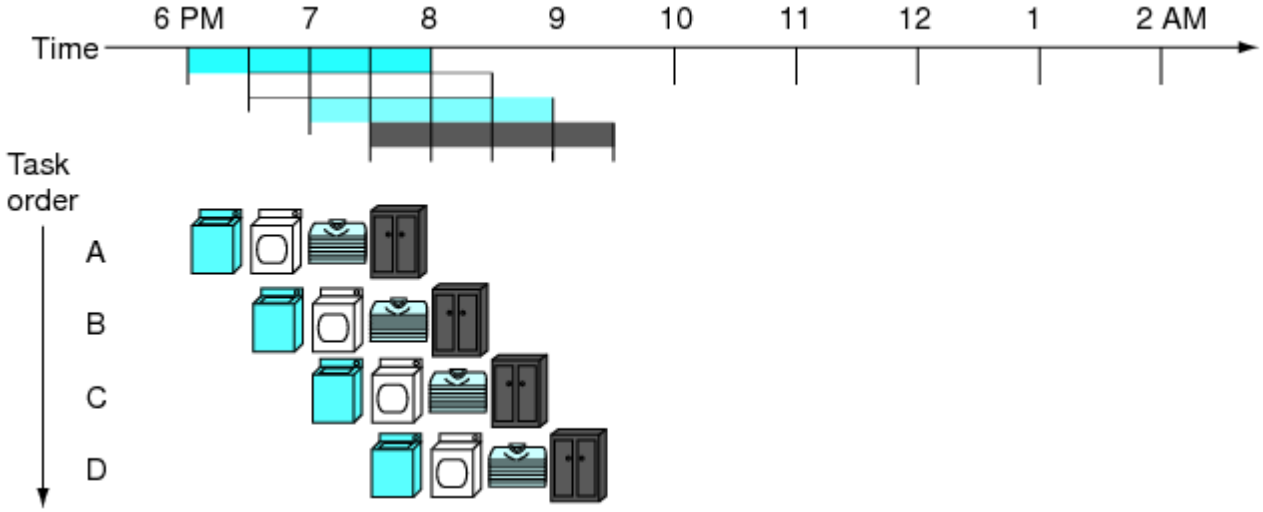
code171.v の命令メモリとデータメモリを m_memory に変更した版

An overview of pipelining (topic of the next lecture)

- Non pipelining (Multi-cycle)

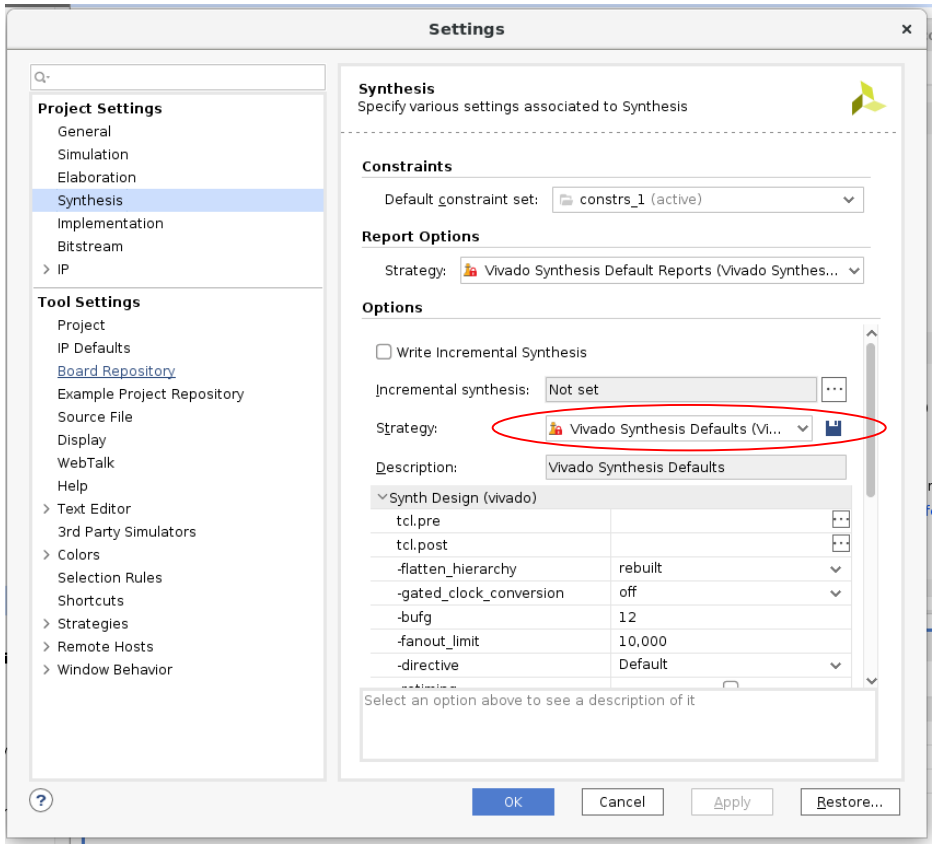


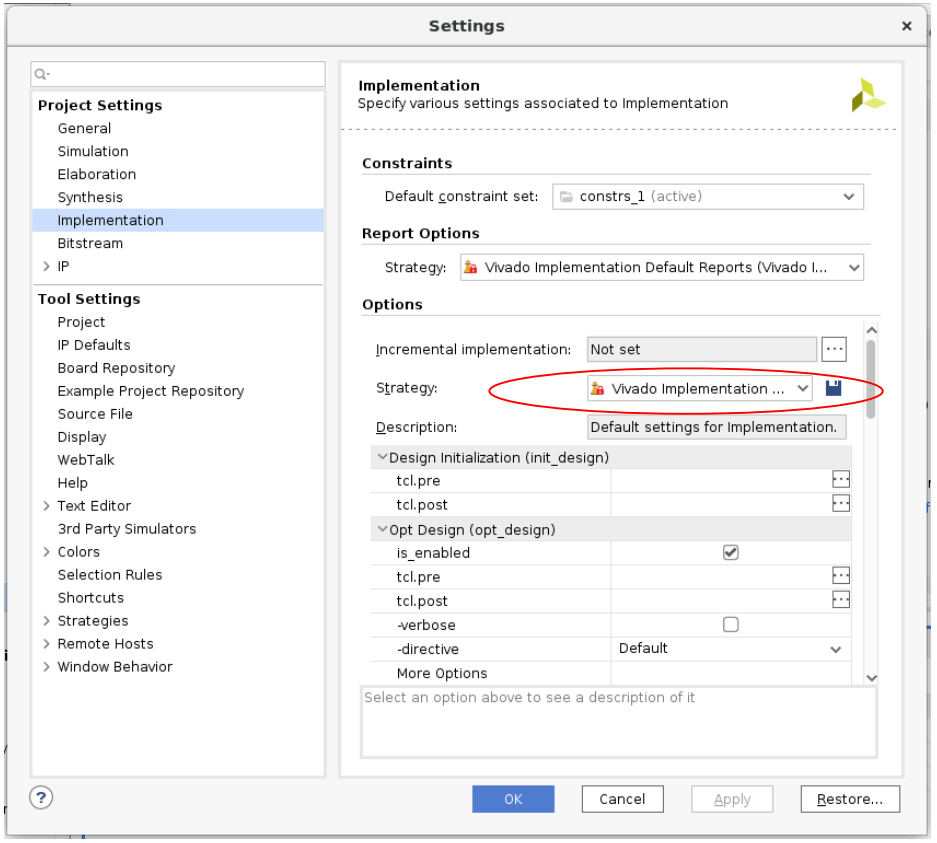
- Pipelining



補足：Vivado の最適化オプション

- Settings -> Synthesis -> Strategy : Vivado Synthesis Defaults
- Settings -> Implementation -> Strategy: Vivado Implementation Defaults





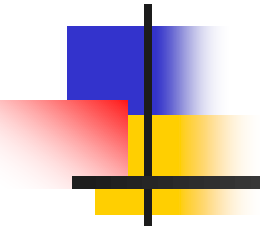
補足：Vivado の最適化オプション

- プロセッサの高性能化のための推奨オプションは次の通り．デザインコンテストでは，これらのVivadoのオプションを適切なものに変更してもよい．

- Settings -> Synthesis -> Strategy :
  Flow_PerfOptimized_high
- Settings -> Implementation -> Strategy:
  Performance_ExplorePostRoutePhysOpt

- ただし，これらの最適化オプションを変更すると，論理合成と配置配線の時間が長くなるので，適切に使い分ける必要がある．

Department of Computer Science
Course number: CSC.T341



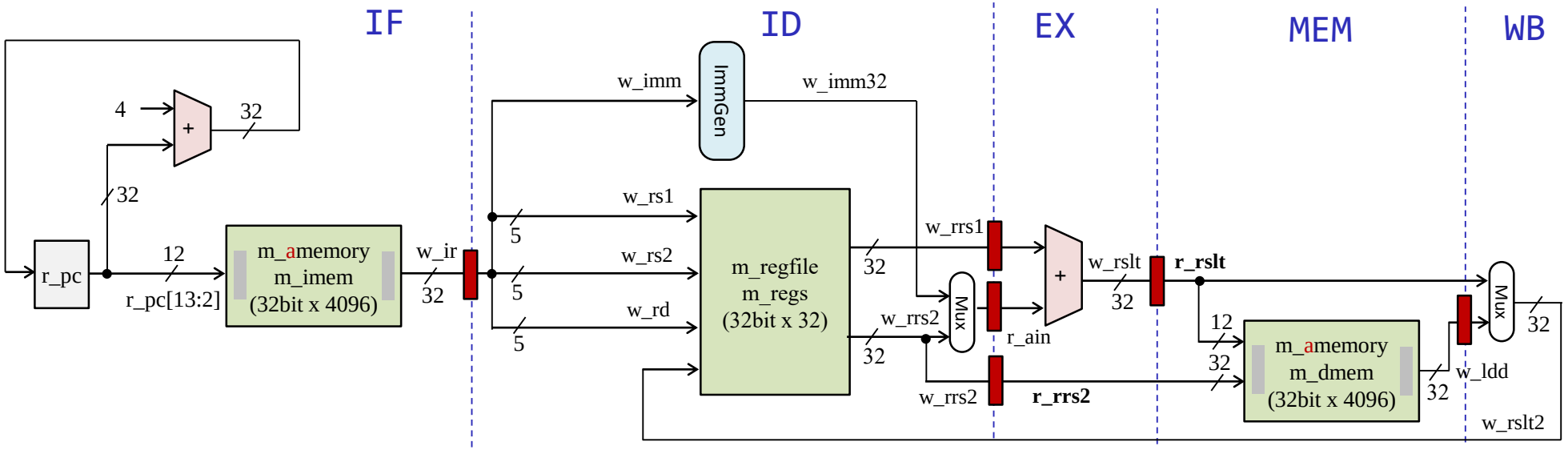
コンピュータ論理設計
Computer Logic Design

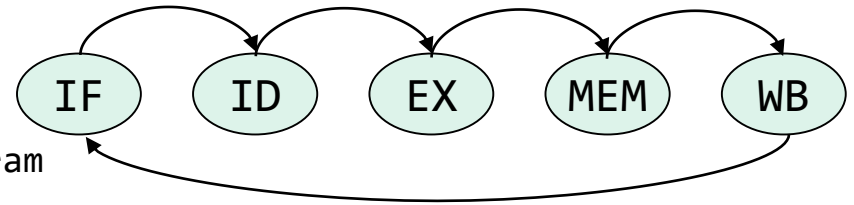
11. パイプラインプロセッサとハザード処理 (1)
Pipelining Processor and Hazards (1)

吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
講義：月曜日 10:45-12:25, 木曜日 10:45-12:25

m_proc09 (multi-cycle processor) マルチサイクル

- 5サイクルで1命令を処理する IPC = 0.2 のマルチサイクルのプロセッサ, add, addi, lw, sw をサポート
- 最も遅延の長い(長い時間を要する)ステップがプロセッサの動作周波数を決める.

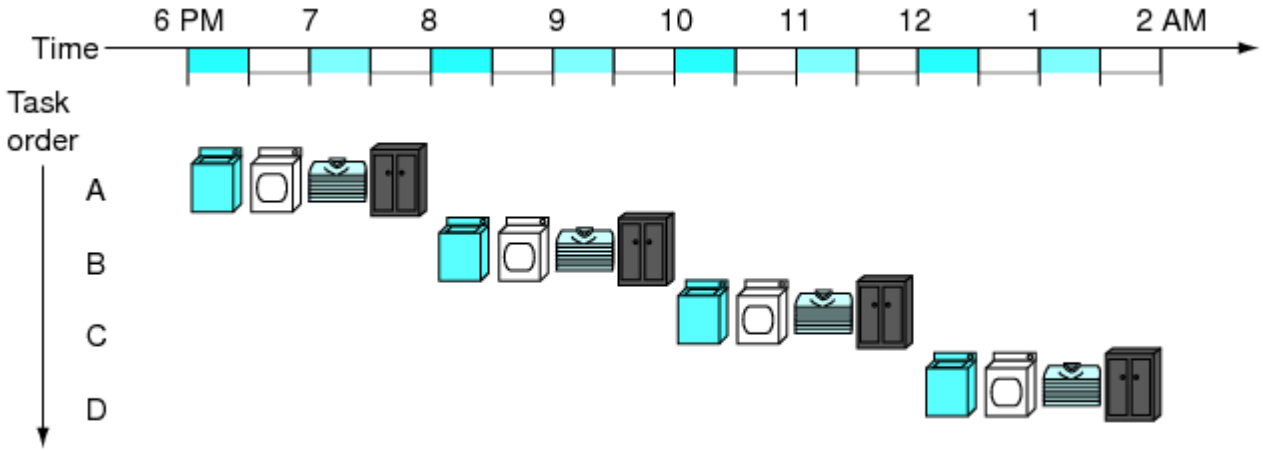




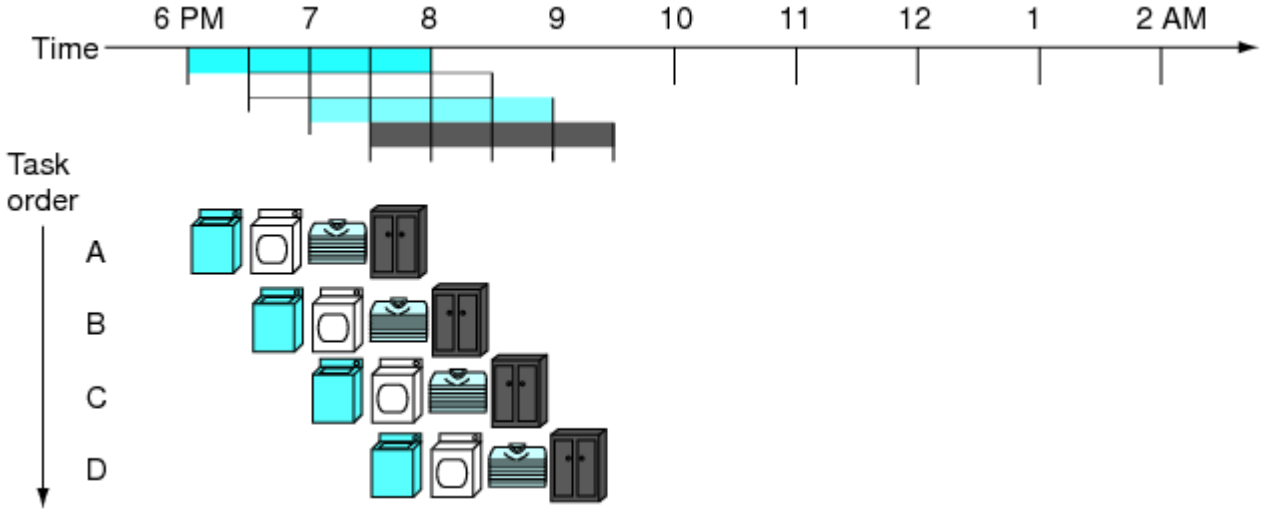
Simple State Machine Diagram

An Overview of Pipelining

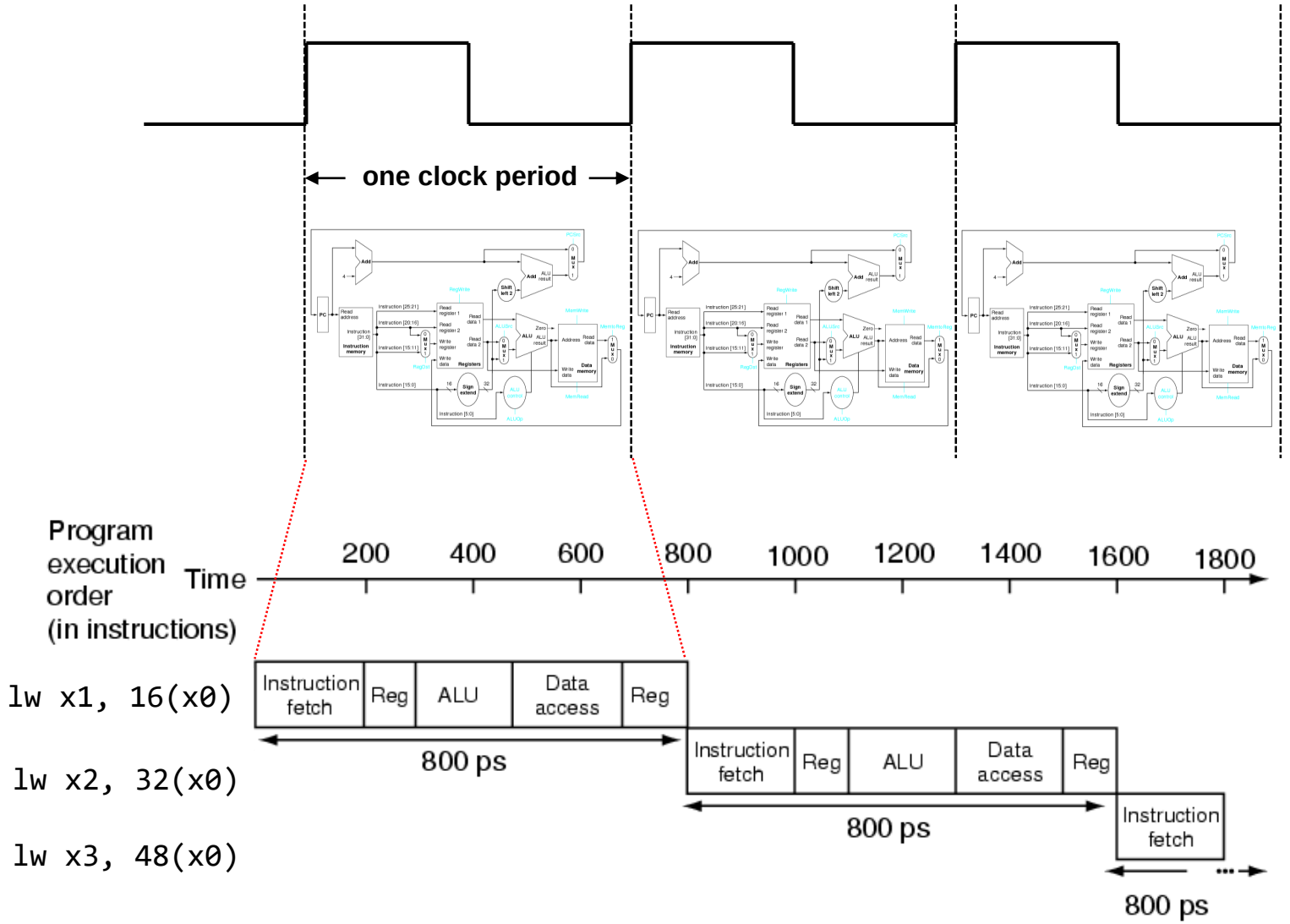
- Non pipelining (Multi-cycle)



- Pipelining

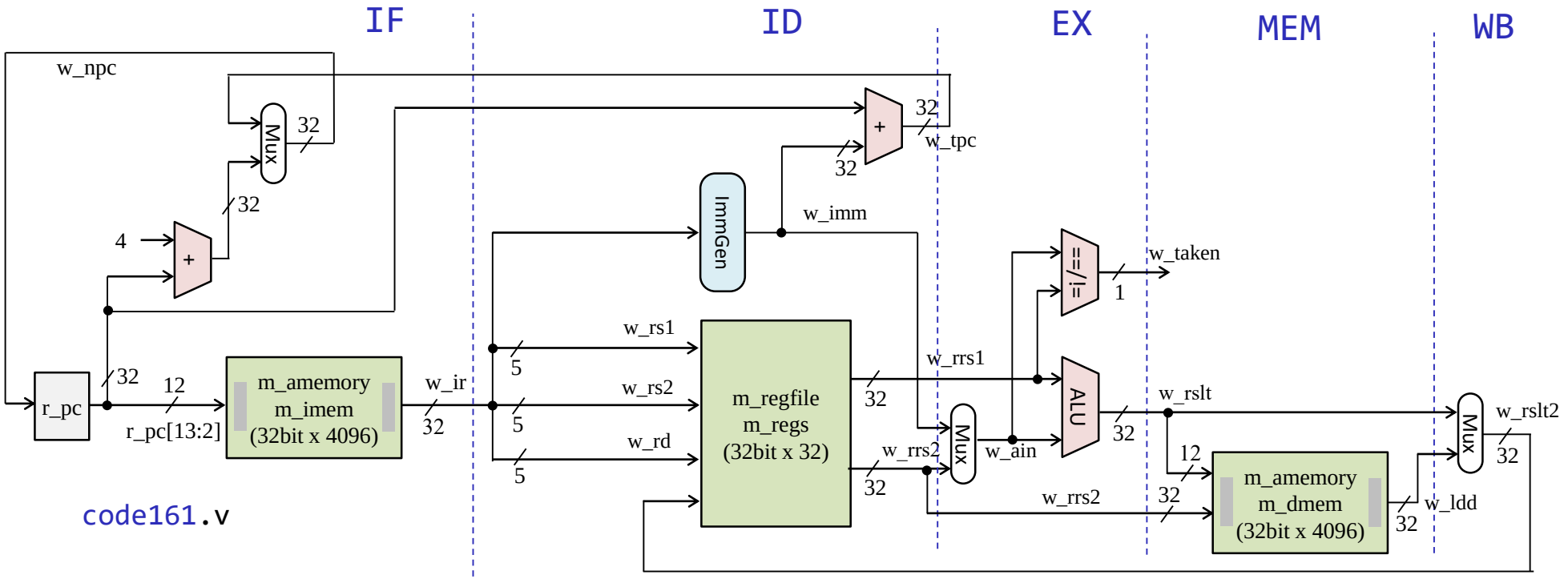


Single Cycle Processor



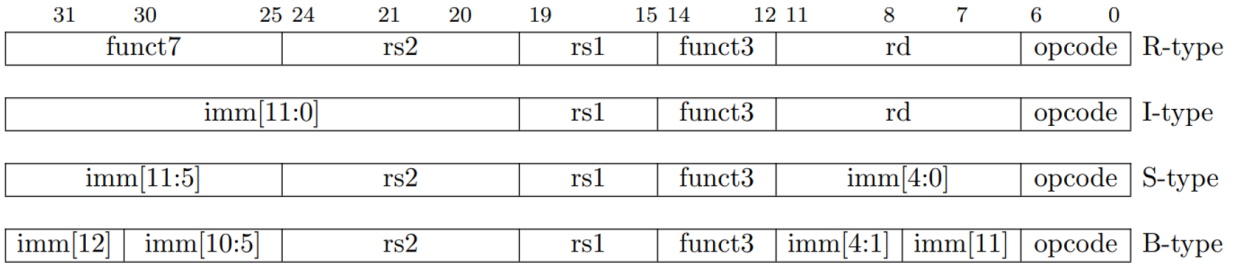
m_proc07 ベースラインのプロセッサ（シングルサイクル）

- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), メモリアクセス(MEM), ライトバック(WB) の処理をおこなうadd, addi, sll, srl, lw, sw, beq, bne命令に対応したプロセッサ

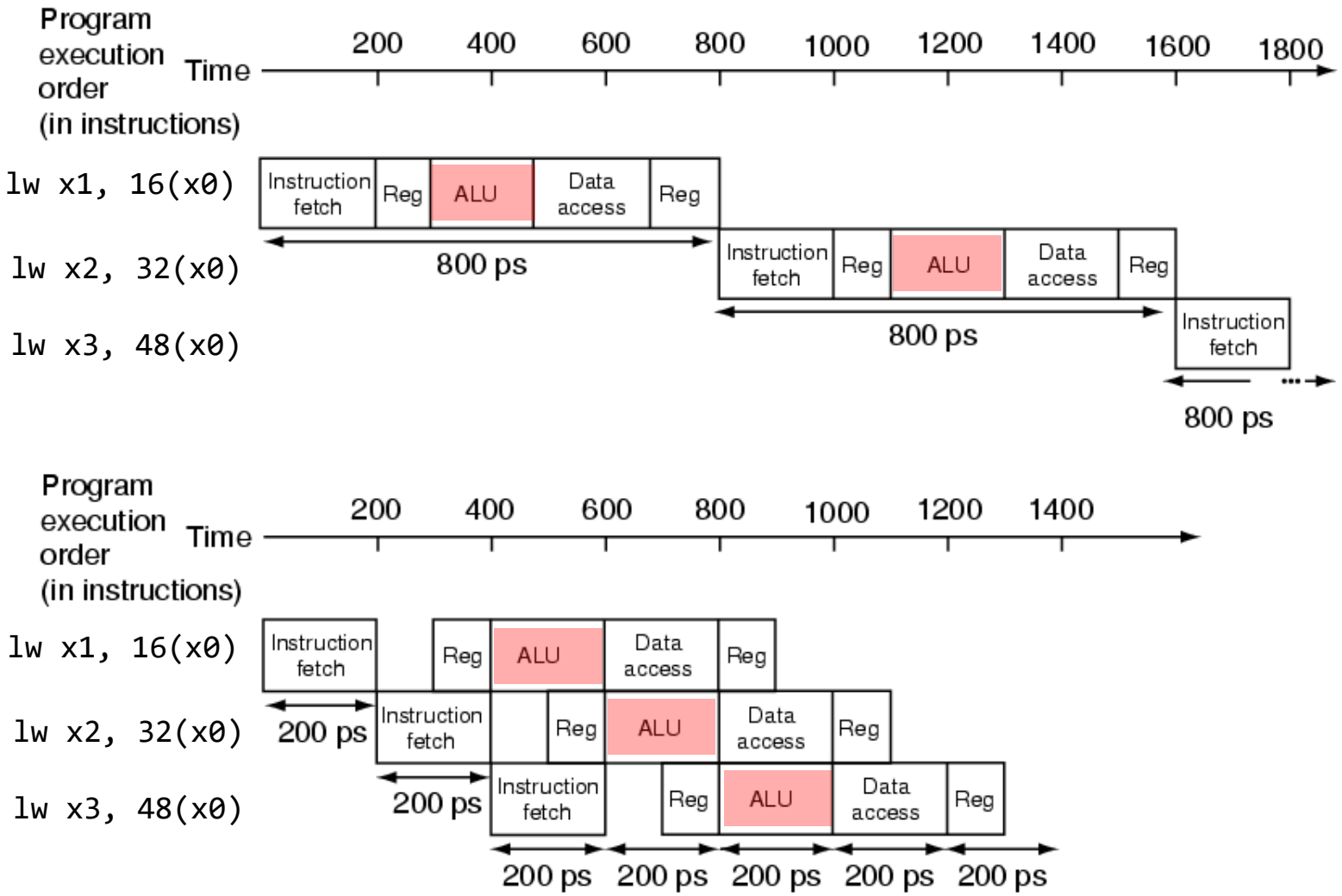


code161.v

```
f    = 60MHz
IPC  = 1.000
Perf = 60.00
```



Single-cycle versus pipelined execution



Hazards make pipelining hard

- 命令を適切なサイクルで実行できないような状況が存在する．これをハザード(hazard)と呼ぶ．
  - 構造ハザード (structural hazard)
    - オーバラップ実行する命令の組み合わせをハードウェアがサポートしていない場合．資源不足により生じる．
  - データ・ハザード(data hazard)
    - データの受け渡しの制約によって生じるハザード

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //    add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd4, 7'b0010011}; //    addi x4, x0, 3   // x4 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //    addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd5, 5'd4, 3'b000, 5'd30,7'b0110011}; //    add  x30,x4, x5  // led = x4 + x5
```

  - 制御ハザード(control hazard) 今回は，この対処方法を考える．
    - 分岐命令，ジャンプ命令によって生じるハザード

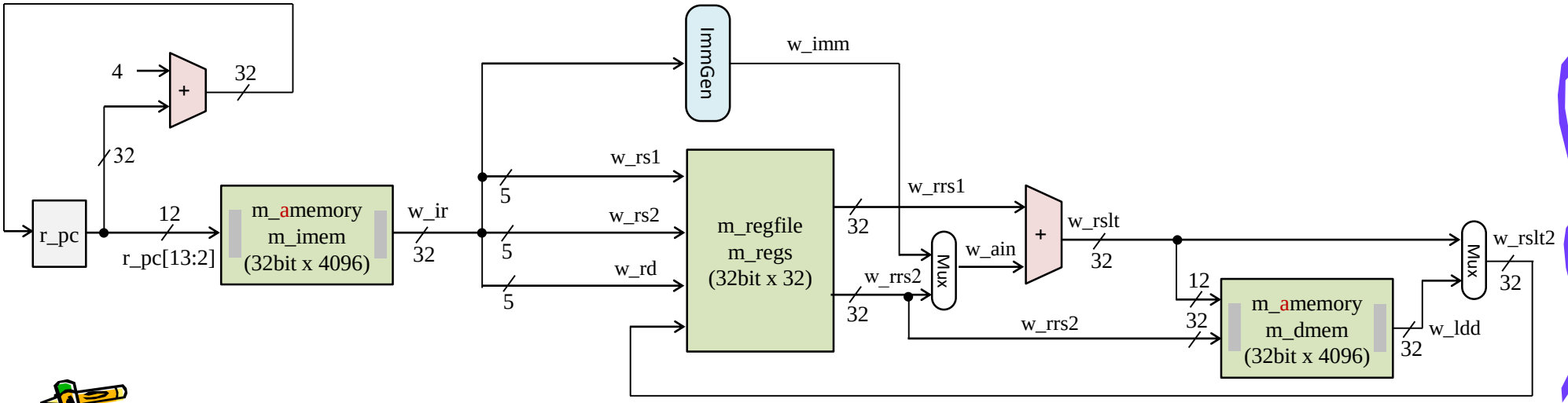
```
cm_ram[1]={12'd5,      5'd0, 3'b000, 5'd4, 7'b0010011};     //    addi x4, x0, 5   // x4 = 5
cm_ram[2]={12'd1,      5'd0, 3'b000, 5'd5, 7'b0010011};     //    addi x5, x0, 1   // x5 = 1
cm_ram[3]={12'd1,      5'd5, 3'b000, 5'd5, 7'b0010011};     // L1:addi x5, x5, 1   // x5 = x5 + 1
cm_ram[4]={7'h7f,5'd4, 5'd5, 3'b001, 5'b11101, 7'b1100011}; //    bne  x5, x4, L1  // goto L1 if x5!=x4
cm_ram[5]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011};     //    add  x30,x5, x0  // led = x5
```

m_proc05 add, addi, lw, sw を処理するシングルサイクル版

```
module m_proc05 (w_clk, w_ce, w_led);
  input  wire w_clk, w_ce;
  output wire [31:0] w_led;

  reg [31:0] r_pc = 0;
  wire [31:0] w_ir;
  wire [4:0]  w_op5 = w_ir[6:2];
  wire [4:0]  w_rs1 = w_ir[19:15];
  wire [4:0]  w_rs2 = w_ir[24:20];
  wire [4:0]  w_rd  = w_ir[11:7];
```

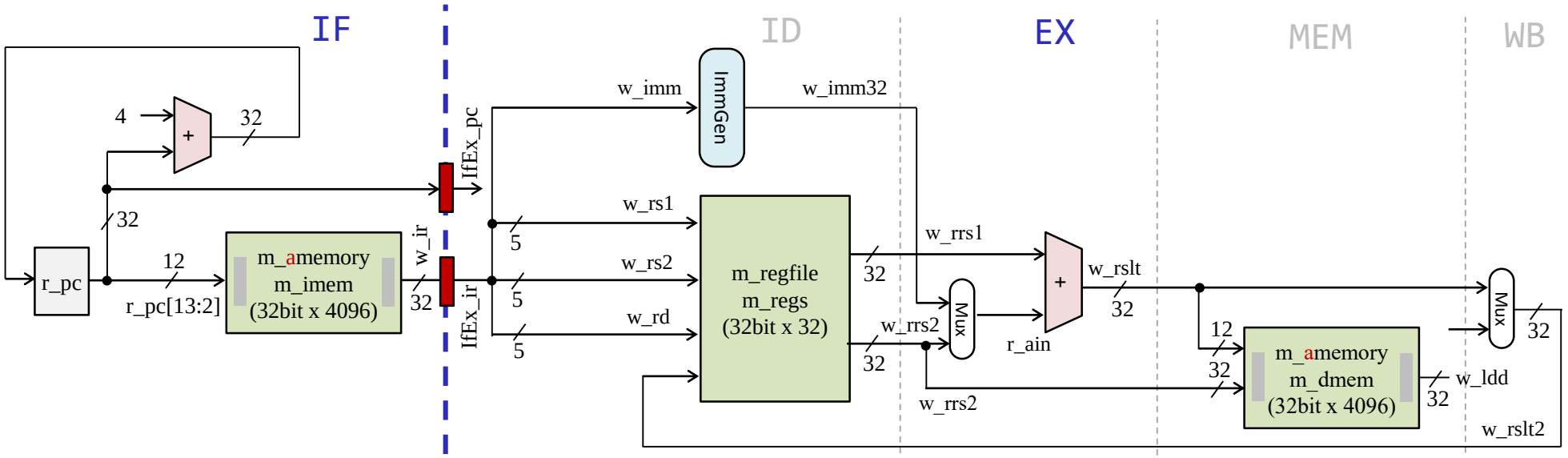
code171.v

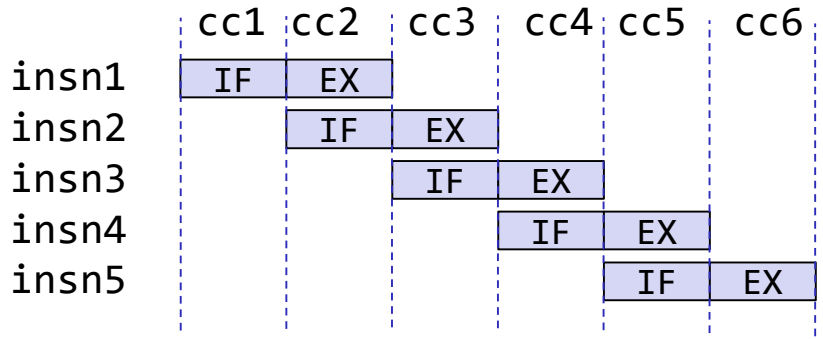
```
  wire #4 w_we = w_ce & (w_op5==5'b01100 || w_op5==5'b00100 || w_op5==5'b00000);
  wire [31:0] w_imm, w_rrs1, w_rrs2, w_ain, w_rslt, w_ldd, w_rslt2;

  m_amemory m_imem (w_clk, r_pc[13:2], 1'd0, 32'd0, w_ir);
  m_immgen m_immgen0 (w_ir, w_imm);
  m_regfile m_regs (w_clk, w_rs1, w_rs2, w_rd, w_we, w_rslt2, w_rrs1, w_rrs2);
  assign #3 w_ain = (w_op5==5'b01100) ? w_rrs2 : w_imm;
  assign #9 w_rslt = w_rrs1 + w_ain;
  m_amemory m_dmem (w_clk, w_rslt[13:2], (w_op5==5'b01000), w_rrs2, w_ldd);
  assign #3 w_rslt2 = (w_op5==5'b00000) ? w_ldd : w_rslt;
  always @(posedge w_clk) #5 if(w_ce & r_pc!=24) r_pc <= r_pc + 4;

  reg [31:0] r_led = 0;
  always @(posedge w_clk) if(w_ce & w_we & w_rd==30) r_led <= w_rslt2;
  assign w_led = r_led;
endmodule
```



m_proc11 2ステージのパイプラインプロセッサ

- IF を1ステージ，その他の ID, EX, MEM, WB を1ステージとするパイプラインプロセッサ
- add, addi, lw, sw を処理するプロセッサ



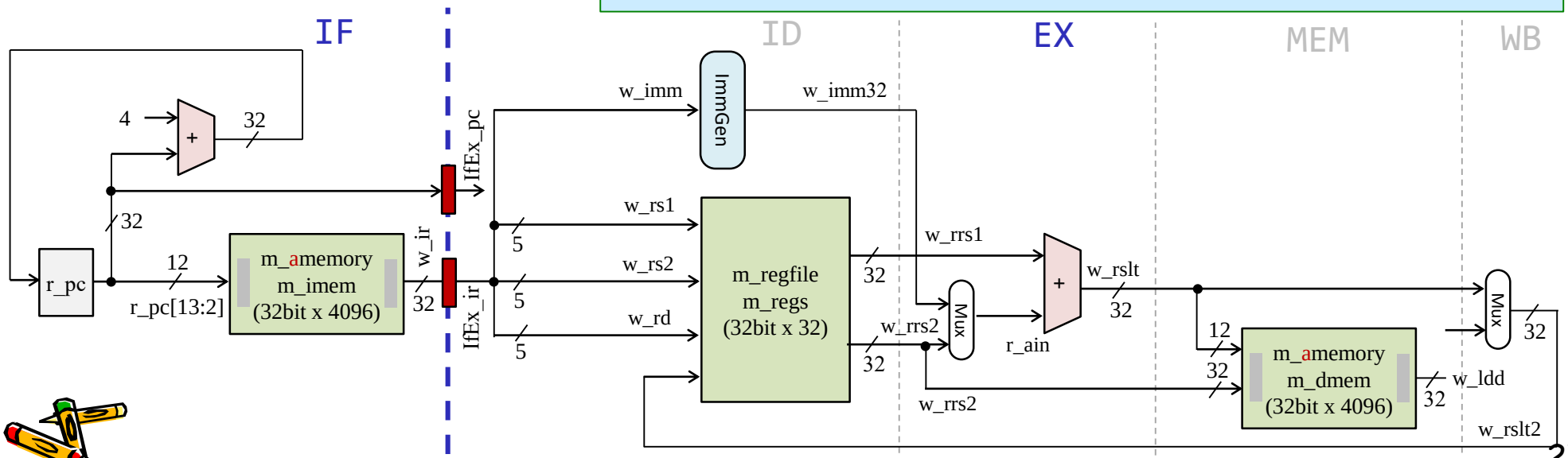


m_proc11 add, addi, lw, sw を処理する2段のパイプライン版

```
module m_proc11 (w_clk, w_ce, w_led);
  input  wire w_clk, w_ce;
  output wire [31:0] w_led;

  reg [31:0] r_pc = 0;
  reg [31:0] IfEx_pc=0;
  reg [31:0] IfEx_ir=0;

  wire [31:0] w_ir;
  wire [4:0]  w_op5 = IfEx_ir[6:2];
  wire [4:0]  w_rs1 = IfEx_ir[19:15];
  wire [4:0]  w_rs2 = IfEx_ir[24:20];
  wire [4:0]  w_rd  = IfEx_ir[11:7];
```

code181.v

```
  wire [31:0] w_imm, w_rrs1, w_rrs2, w_ain, w_rslt, w_ldd, w_rslt2;
  wire #4 w_we = w_ce & (w_op5==5'b01100 || w_op5==5'b00100 || w_op5==5'b00000);
  always @(posedge w_clk) #5 IfEx_pc <= r_pc;
  m_amemory m_imem (w_clk, r_pc[13:2], 1'd0, 32'd0, w_ir);
  always @(posedge w_clk) #5 IfEx_ir <= w_ir;

  m_immgen m_immgen0 (IfEx_ir, w_imm);
  m_regfile m_regs (w_clk, w_rs1, w_rs2, w_rd, w_we, w_rslt2, w_rrs1, w_rrs2);
  assign #3 w_ain = (w_op5==5'b01100) ? w_rrs2 : w_imm;
  assign #9 w_rslt = w_rrs1 + w_ain;
  m_amemory m_dmem (w_clk, w_rslt[13:2], (w_op5==5'b01000), w_rrs2, w_ldd);
  assign #3 w_rslt2 = (w_op5==5'b00000) ? w_ldd : w_rslt;
  always @(posedge w_clk) #5 if(w_ce && IfEx_ir!=32'h000f0033) r_pc <= r_pc + 4;

  reg [31:0] r_led = 0;
  always @(posedge w_clk) if(w_ce & w_we & w_rd==30) r_led <= w_rslt;
  assign w_led = r_led;
endmodule
```



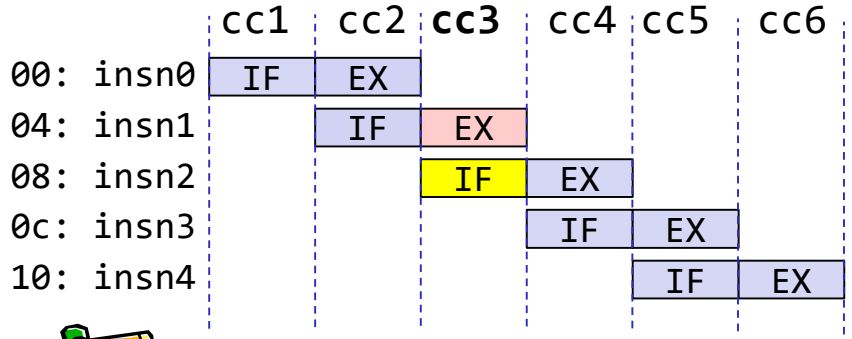
m_proc11 add, addi, lw, sw を処理する2段のパイプライン版

```
module m_proc11 (w_clk, w_ce, w_led);
  input  wire w_clk, w_ce;
  output wire [31:0] w_led;

  reg [31:0] r_pc = 0;
  reg [31:0] IfEx_pc=0;
  reg [31:0] IfEx_ir=0;

  wire [31:0] w_ir;
  wire [4:0]  w_op5 = IfEx_ir[6:2];
  wire [4:0]  w_rs1 = IfEx_ir[19:15];
  wire [4:0]  w_rs2 = IfEx_ir[24:20];
  wire [4:0]  w_rd  = IfEx_ir[11:7];
```

code181.v

```
  wire [31:0] w_imm, w_rrs1, w_rrs2, w_ain, w_rslt, w_ldd, w_rslt2;
  wire #4 w_we = w_ce & (w_op5==5'b01100 || w_op5==5'b00100 || w_op5==5'b00000);
  always @(posedge w_clk) #5 IfEx_pc <= r_pc;
  m_amemory m_imem (w_clk, r_pc[13:2], 1'd0, 32'd0, w_ir);
  always @(posedge w_clk) #5 IfEx_ir <= w_ir;

  m_immgen m_immgen0 (IfEx_ir, w_imm);
  m_regfile m_regs (w_clk, w_rs1, w_rs2, w_rd, w_we, w_rslt2, w_rrs1, w_rrs2);
  assign #3 w_ain = (w_op5==5'b01100) ? w_rrs2 : w_imm;
  assign #9 w_rslt = w_rrs1 + w_ain;
  m_amemory m_dmem (w_clk, w_rslt[13:2], (w_op5==5'b01000), w_rrs2, w_ldd);
  assign #3 w_rslt2 = (w_op5==5'b00000) ? w_ldd : w_rslt;
  always @(posedge w_clk) #5 if(w_ce && IfEx_ir!=32'h000f0033) r_pc <= r_pc + 4;

  reg [31:0] r_led = 0;
  always @(posedge w_clk) if(w_ce & w_we & w_rd==30) r_led <= w_rslt;
  assign w_led = r_led;
endmodule
```

```
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd4, 7'b0010011}; // 04   addi x4, x0, 3   // x4 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; // 08   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd5, 5'd4, 3'b000, 5'd30,7'b0110011}; // 0c   add  x30,x4, x5  // led = x4 + x5
```




```
clock: r_pc     IfEx_pc  IfEx_ir  w_rrs1   w_ain    r_rslt2  r_led
    1: 00000000 00000000 00000000 00000000 00000000 00000033 00000000
    2: 00000004 00000000 00000033 00000000 00000000 00000000 00000000
    3: 00000008 00000004 00300213 00000000 00000003 00000003 00000000
    4: 0000000c 00000008 00400293 00000000 00000004 00000004 00000000
    5: 00000010 0000000c 00520f33 00000003 00000004 00000007 00000000
    6: 00000014 00000010 00000033 00000000 00000000 00000000 00000007
```

Hazards make pipelining hard

- 命令を適切なサイクルで実行できないような状況が存在する．これをハザード(hazard)と呼ぶ．
  - 構造ハザード (structural hazard)
    - オーバラップ実行する命令の組み合わせをハードウェアがサポートしていない場合．資源不足により生じる．
  - データ・ハザード(data hazard)
    - データの受け渡しの制約によって生じるハザード

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //    add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd4, 7'b0010011}; //    addi x4, x0, 3   // x4 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //    addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd5, 5'd4, 3'b000, 5'd30,7'b0110011}; //    add  x30,x4, x5  // led = x4 + x5
```

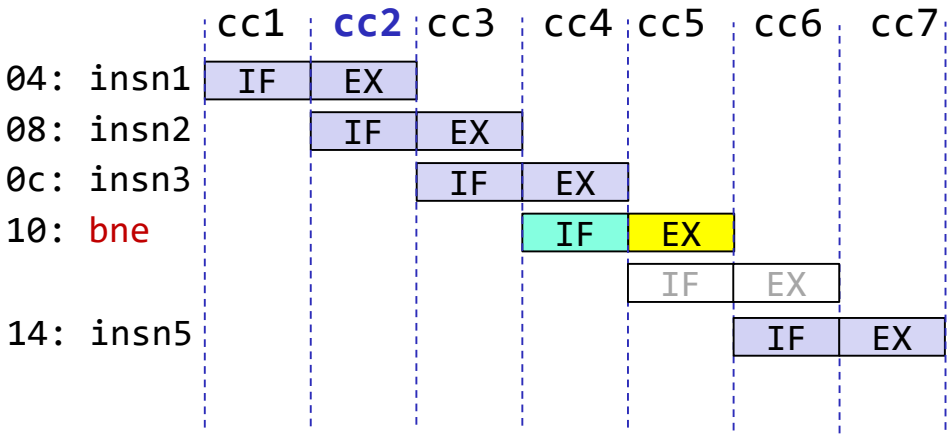
  - 制御ハザード(control hazard)
    - 分岐命令，ジャンプ命令によって生じるハザード

```
cm_ram[1]={12'd5,      5'd0, 3'b000, 5'd4, 7'b0010011};     //    addi x4, x0, 5   // x4 = 5
cm_ram[2]={12'd1,      5'd0, 3'b000, 5'd5, 7'b0010011};     //    addi x5, x0, 1   // x5 = 1
cm_ram[3]={12'd1,      5'd5, 3'b000, 5'd5, 7'b0010011};     // L1:addi x5, x5, 1   // x5 = x5 + 1
cm_ram[4]={7'h7f,5'd4, 5'd5, 3'b001, 5'b11101, 7'b1100011}; //    bne  x5, x4, L1  // goto L1 if x5!=x4
cm_ram[5]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011};     //    add  x30,x5, x0  // led = x5
```

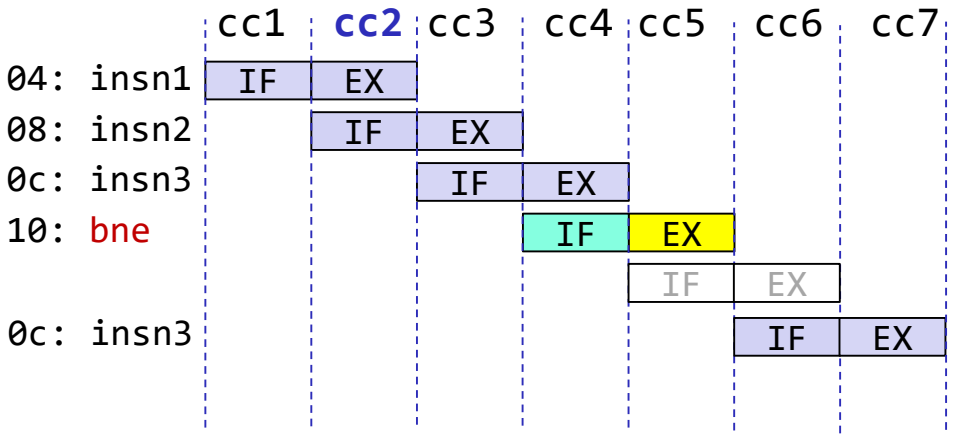
制御ハザードの対処：戦略1

- 分岐方向が判明するまで，分岐命令の後続命令のフェッチを止める（ストールさせる）.
  - このプロセッサ構成では，分岐命令の出現毎に1サイクルの無駄が生じる.

```
cm_ram[1]={12'd5,      5'd0, 3'b000, 5'd4, 7'b0010011};     // 04    addi x4, x0, 5   // x4 = 5
cm_ram[2]={12'd1,      5'd0, 3'b000, 5'd5, 7'b0010011};     // 08    addi x5, x0, 1   // x5 = 1
cm_ram[3]={12'd1,      5'd5, 3'b000, 5'd5, 7'b0010011};     // 0c L1:addi x5, x5, 1   // x5 = x5 + 1
cm_ram[4]={7'h7f,5'd4, 5'd5, 3'b001, 5'b11101, 7'b1100011}; // 10    bne  x5, x4, L1  // goto L1 if x5!=x4
cm_ram[5]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011};     // 14    add  x30,x5, x0  // led = x5
```



分岐が不成立の場合

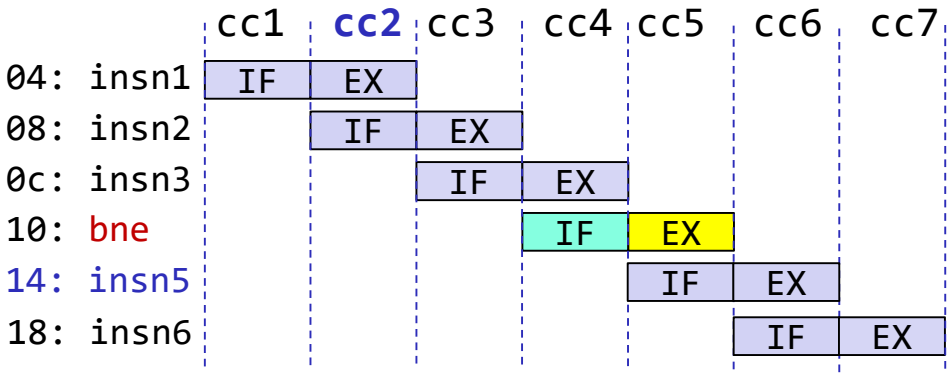


分岐が成立の場合

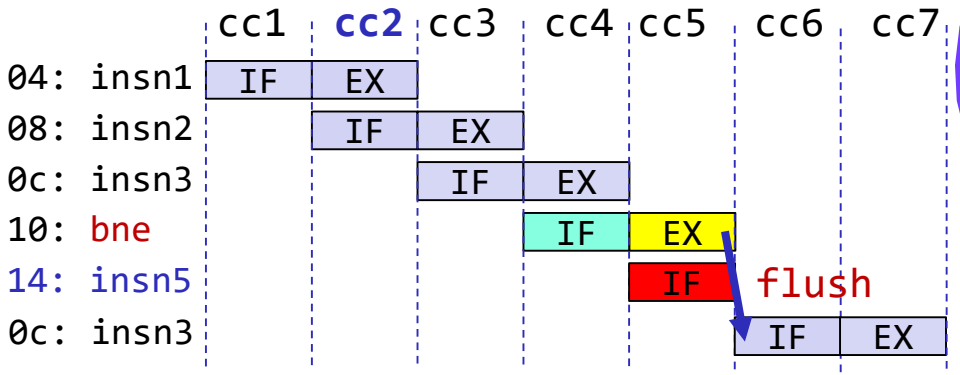
制御ハザードの対処：戦略2

- 分岐が不成立と仮定して分岐命令の後続命令の処理を進める.
  - その分岐が不成立の場合には，仮定が正しいので無駄は生じない.
  - この構成では，その分岐が成立の場合，間違ってフェッチした次の命令を削除(flush)して，正しい pc の命令のフェッチを再開する．1サイクルの無駄が発生する.

```
cm_ram[1]={12'd5,      5'd0, 3'b000, 5'd4, 7'b0010011};     // 04    addi x4, x0, 5   // x4 = 5
cm_ram[2]={12'd1,      5'd0, 3'b000, 5'd5, 7'b0010011};     // 08    addi x5, x0, 1   // x5 = 1
cm_ram[3]={12'd1,      5'd5, 3'b000, 5'd5, 7'b0010011};     // 0c L1:addi x5, x5, 1   // x5 = x5 + 1
cm_ram[4]={7'h7f,5'd4, 5'd5, 3'b001, 5'b11101, 7'b1100011}; // 10    bne  x5, x4, L1  // goto L1 if x5!=x4
cm_ram[5]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011};     // 14    add  x30,x5, x0  // led = x5
```



分岐が不成立の場合

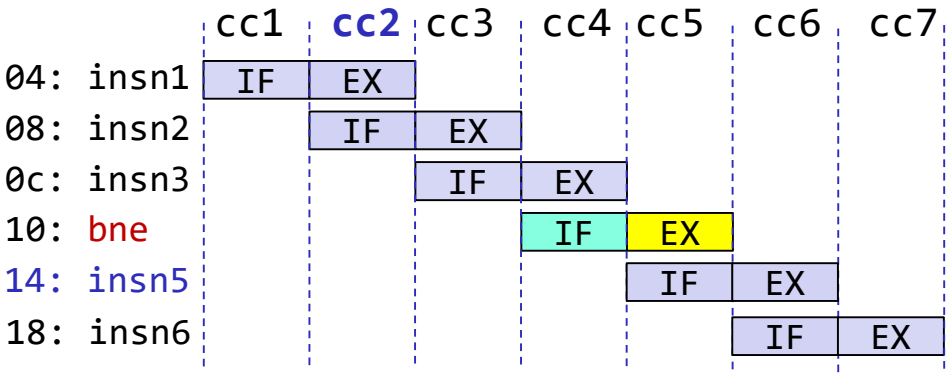


分岐が成立の場合

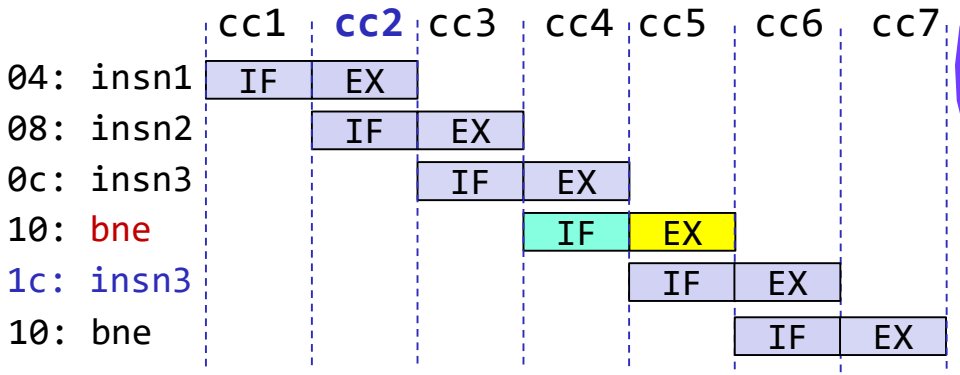
制御ハザードの対処：戦略3

- 分岐の成立／不成立を予測して，その予測が正しいと仮定して分岐命令の後続命令の処理を進める．
  - その分岐の結果が予測と同じ場合には，予測が成功で無駄は生じない．
  - この構成では，予測が失敗の場合，間違ってフェッチした次の命令を削除(flush)して，正しい pc の命令のフェッチを再開する．1サイクルの無駄が発生．

```
cm_ram[1]={12'd5,      5'd0, 3'b000, 5'd4, 7'b0010011};     // 04    addi x4, x0, 5   // x4 = 5
cm_ram[2]={12'd1,      5'd0, 3'b000, 5'd5, 7'b0010011};     // 08    addi x5, x0, 1   // x5 = 1
cm_ram[3]={12'd1,      5'd5, 3'b000, 5'd5, 7'b0010011};     // 0c L1:addi x5, x5, 1   // x5 = x5 + 1
cm_ram[4]={7'h7f,5'd4, 5'd5, 3'b001, 5'b11101, 7'b1100011}; // 10    bne  x5, x4, L1  // goto L1 if x5!=x4
cm_ram[5]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011};     // 14    add  x30,x5, x0  // led = x5
```



分岐が不成立と予測して成功の場合



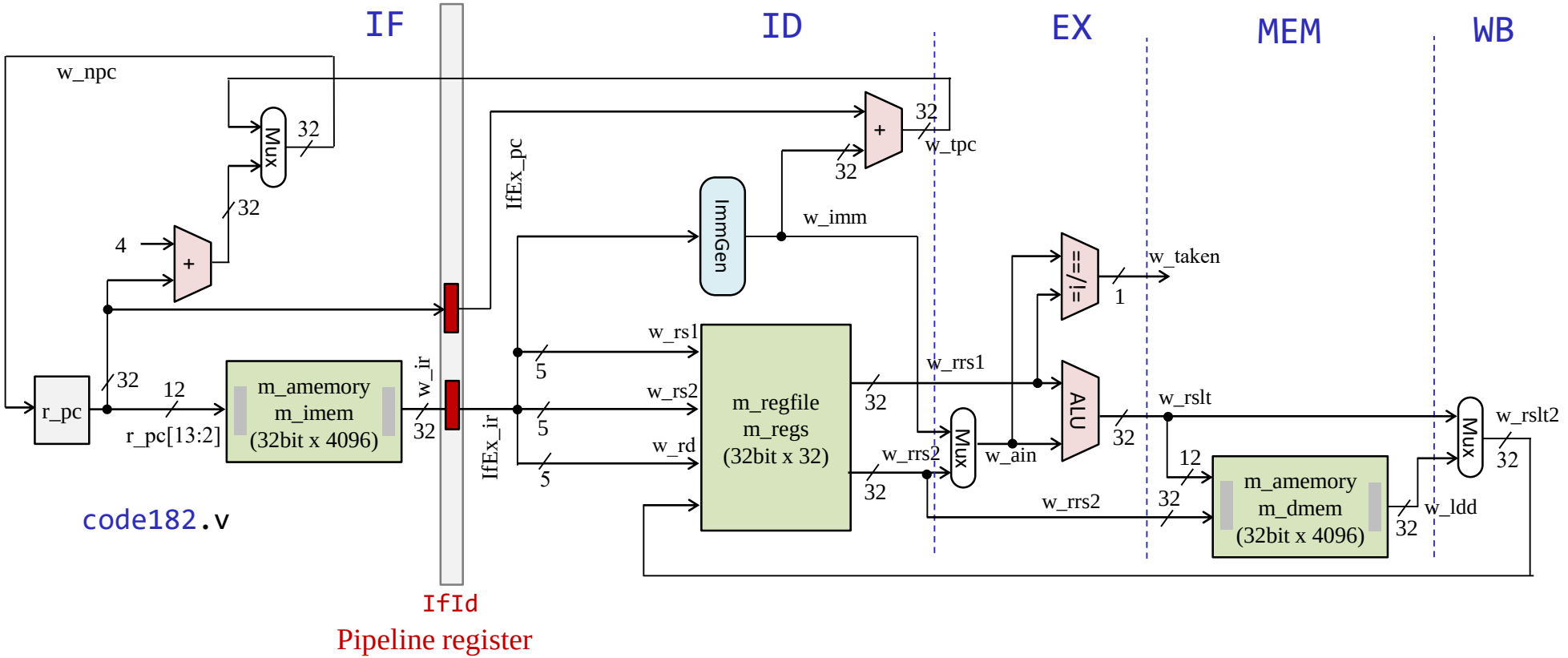
分岐が成立と予測して成功の場合

制御ハザードへの対処

- 戦略3の分岐予測を用いる方法は，コンピュータアーキテクチャの講義で扱う.
- この講義では，戦略2を採用する.
  - 分岐が不成立と仮定して分岐命令の後続命令の処理を進める.

m_proc13 2段のパイプライン版

- add, addi, sll, srl, lw, sw, beq, bne命令に対応したプロセッサ
- IF を1ステージ，その他の ID, EX, MEM, WB を1ステージとするパイプラインプロセッサ



m_proc13 2段のパイプライン版

- 分岐が成立 (w_tkn==1) の場合，間違ってフェッチした次の命令を削除 (flush) して，正しい pc の命令のフェッチを再開する．1サイクルの無駄が発生する．
- 具体的には，w_tkn==1 の時に，フェッチすべき命令を 0 (NOP) で上書きする．
  - always @(posedge w_clk) IfEx_ir <= (w_tkn) ? 0 : w_ir;
- 命令に有効ビット (valid bit) を付けてもよい．

```
module m_proc13 (w_clk, w_ce, w_led);
  input  wire w_clk, w_ce;
  output wire [31:0] w_led;

  reg [31:0] r_pc = 0;
  reg [31:0] IfEx_pc=0;
  reg [31:0] IfEx_ir=0;

  wire [31:0] w_ir;
  wire [4:0]  w_op5 = IfEx_ir[6:2];
  wire [4:0]  w_rs1 = IfEx_ir[19:15];
  wire [4:0]  w_rs2 = IfEx_ir[24:20];
  wire [4:0]  w_rd  = IfEx_ir[11:7];
  wire [2:0]  w_f3  = IfEx_ir[14:12];
```

code182.v

```
  wire [31:0] w_imm, w_rrs1, w_rrs2, w_ain, w_rslt, w_ldd, w_rslt2;
  wire #4 w_we = w_ce & (w_op5==5'b01100 || w_op5==5'b00100 || w_op5==5'b00000);
  always @(posedge w_clk) #5 IfEx_pc <= r_pc;
  m_amemory m_imem (w_clk, r_pc[13:2], 1'd0, 32'd0, w_ir);
  always @(posedge w_clk) #5 IfEx_ir <= (w_tkn) ? 0 : w_ir;

  m_immgen m_immgen0 (IfEx_ir, w_imm);

  m_regfile m_regs (w_clk, w_rs1, w_rs2, w_rd, w_we, w_rslt2, w_rrs1, w_rrs2);
  assign w_ain = (w_op5==5'b01100) ? w_rrs2 : w_imm;
  assign w_rslt = (w_f3==3'b001) ? w_rrs1 << w_ain[4:0] :
                  (w_f3==3'b101) ? w_rrs1 >> w_ain[4:0] : w_rrs1 + w_ain;

  m_amemory m_dmem (w_clk, w_rslt[13:2], (w_op5==5'b01000), w_rrs2, w_ldd);
  assign w_rslt2 = (w_op5==5'b00000) ? w_ldd : w_rslt;

  wire w_tkn = ({IfEx_ir[12],w_op5}==6'b011000 & w_rrs1==w_rrs2) ||  // BEQ
               ({IfEx_ir[12],w_op5}==6'b111000 & w_rrs1!=w_rrs2);    // BNE
  always @(posedge w_clk) #5
    if(w_ce && IfEx_ir!=32'h000f0033) r_pc <= (w_tkn) ? IfEx_pc + w_imm : r_pc+4;

  reg [31:0] r_led = 0;
  always @(posedge w_clk) if(w_ce & w_we & w_rd==30) r_led <= w_rslt;
  assign w_led = r_led;
endmodule
```

m_proc13 2段のパイプライン版

/home/tu_kise/cld/2023/baseline/program5.txt

```
initial begin
  cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011};     // 00    add  x0, x0, x0  // NOP
  cm_ram[1]={12'd5,      5'd0, 3'b000, 5'd4, 7'b0010011};     // 04    addi x4, x0, 5   // x4 = 5
  cm_ram[2]={12'd1,      5'd0, 3'b000, 5'd5, 7'b0010011};     // 08    addi x5, x0, 1   // x5 = 1
  cm_ram[3]={12'd1,      5'd5, 3'b000, 5'd5, 7'b0010011};     // 0c L1:addi x5, x5, 1   // x5 = x5 + 1
  cm_ram[4]={7'h7f,5'd4, 5'd5, 3'b001, 5'b11101, 7'b1100011}; // 10    bne  x5, x4, L1  // goto L1 if x5!=x4
  cm_ram[5]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011};     // 14    add  x30,x5, x0  // led = x5
  cm_ram[6]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011};     // 18    add  x0, x0, x0  // NOP
  cm_ram[7]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011};     // 1c    add  x0, x0, x0  // NOP
end
```

```
clock: r_pc     w_ir     w_rrs1   w_ain    r_rslt2  r_led
    1: 00000000 00000033 00000000 00000000 00000000 00000000
    2: 00000004 00500213 00000000 00000005 00000005 00000000
    3: 00000008 00100293 00000000 00000001 00000001 00000000
    4: 0000000c 00128293 00000001 00000001 00000002 00000000
    5: 00000010 fe429ee3 00000002 fffffffc 20000000 00000000
    6: 0000000c 00128293 00000002 00000001 00000003 00000000
    7: 00000010 fe429ee3 00000003 fffffffc 30000000 00000000
    8: 0000000c 00128293 00000003 00000001 00000004 00000000
    9: 00000010 fe429ee3 00000004 fffffffc 40000000 00000000
   10: 0000000c 00128293 00000004 00000001 00000005 00000000
   11: 00000010 fe429ee3 00000005 fffffffc 50000000 00000000
   12: 00000014 00028f33 00000005 00000000 00000005 00000000
   13: 00000018 00000033 00000000 00000000 00000000 00000005
```

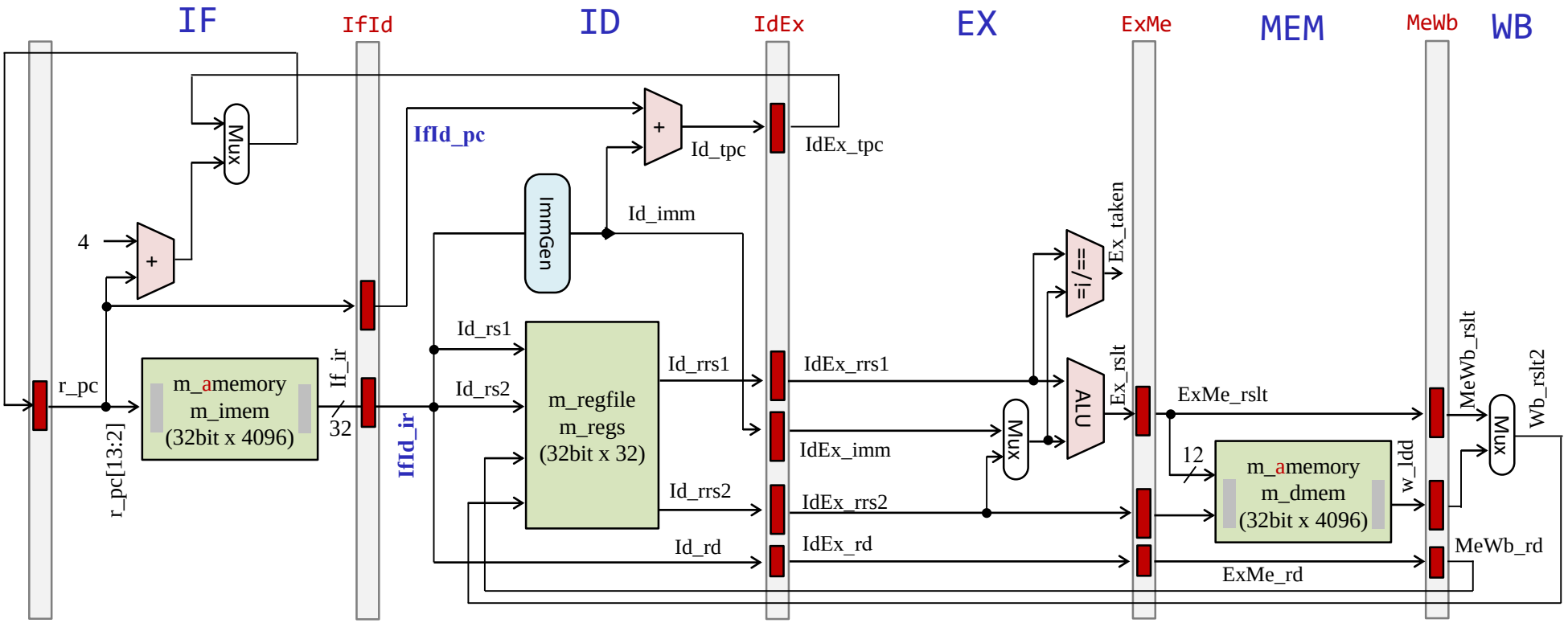
m_proc07 ベースラインのプロセッサの結果

```
clock: r_pc     IfEx_pc  IfEx_ir  w_rrs1   w_ain    r_rslt2  r_led
    1: 00000000 00000000 00000000 00000000 00000000 00000033 00000000
    2: 00000004 00000000 00000033 00000000 00000000 00000000 00000000
    3: 00000008 00000004 00500213 00000000 00000005 00000005 00000000
    4: 0000000c 00000008 00100293 00000000 00000001 00000001 00000000
    5: 00000010 0000000c 00128293 00000001 00000001 00000002 00000000
    6: 00000014 00000010 fe429ee3 00000002 fffffffc 20000000 00000000
    7: 0000000c 00000014 00000000 00000000 00000000 00000033 00000000
    8: 00000010 0000000c 00128293 00000002 00000001 00000003 00000000
    9: 00000014 00000010 fe429ee3 00000003 fffffffc 30000000 00000000
   10: 0000000c 00000014 00000000 00000000 00000000 00000033 00000000
   11: 00000010 0000000c 00128293 00000003 00000001 00000004 00000000
   12: 00000014 00000010 fe429ee3 00000004 fffffffc 40000000 00000000
   13: 0000000c 00000014 00000000 00000000 00000000 00000033 00000000
   14: 00000010 0000000c 00128293 00000004 00000001 00000005 00000000
   15: 00000014 00000010 fe429ee3 00000005 fffffffc 50000000 00000000
   16: 00000018 00000014 00028f33 00000005 00000000 00000005 00000000
   17: 0000001c 00000018 00000033 00000000 00000000 00000000 00000005
```

m_proc13 2段のパイプライン版の結果

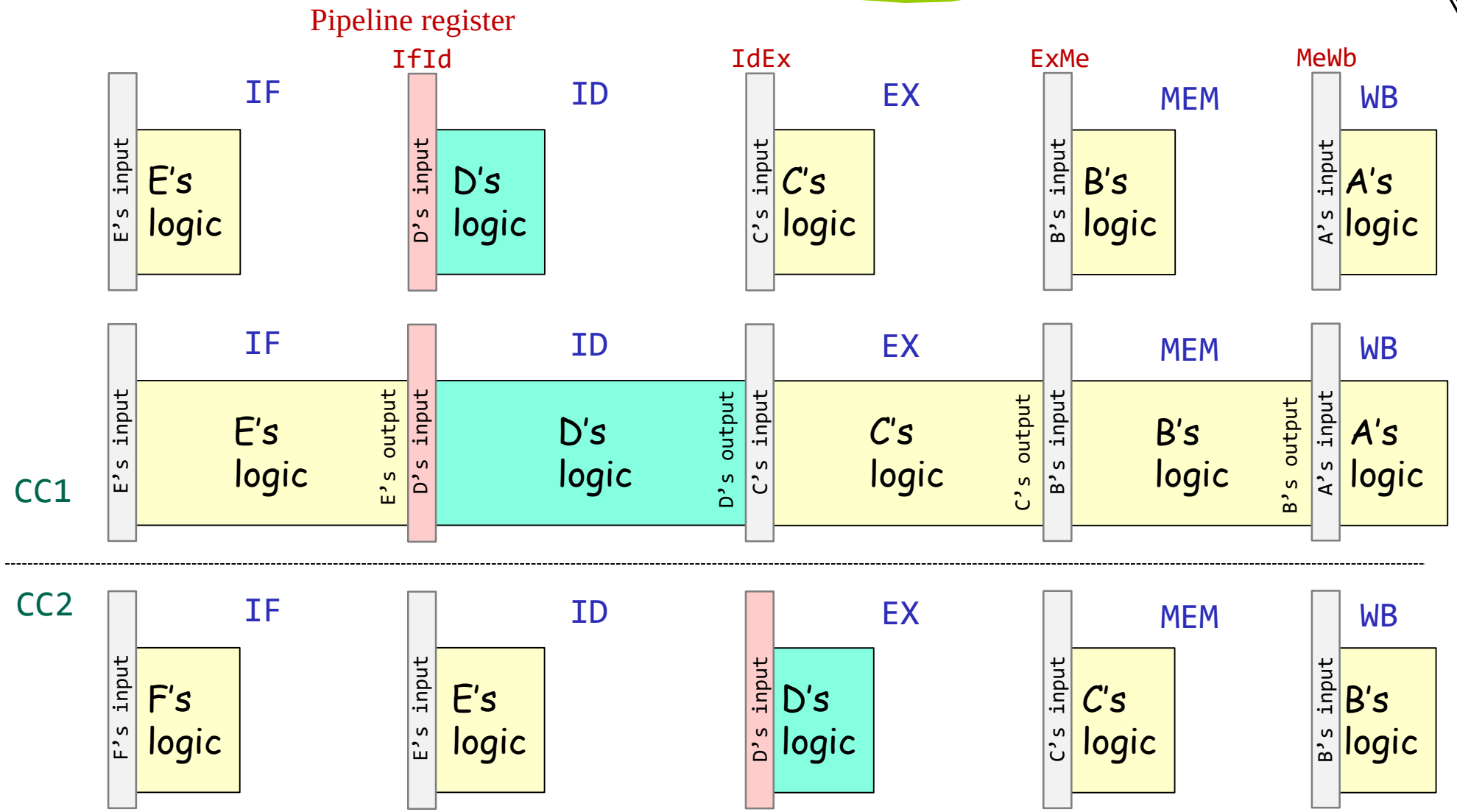
m_proc14 5段のパイプライン版

- add, addi, sll, srl, lw, sw, beq, bne命令に対応したプロセッサ（データフォワーディング無し）
- IF, ID, EX, MEM, WB のそれぞれをステージとする5段のパイプラインプロセッサ
- ステージ IF と ID の間のパイプラインレジスタには IfId_ から始まる名前を利用する.
- ステージ ID で生成される配線には Id_ から始まる名前を利用する.



code185.v

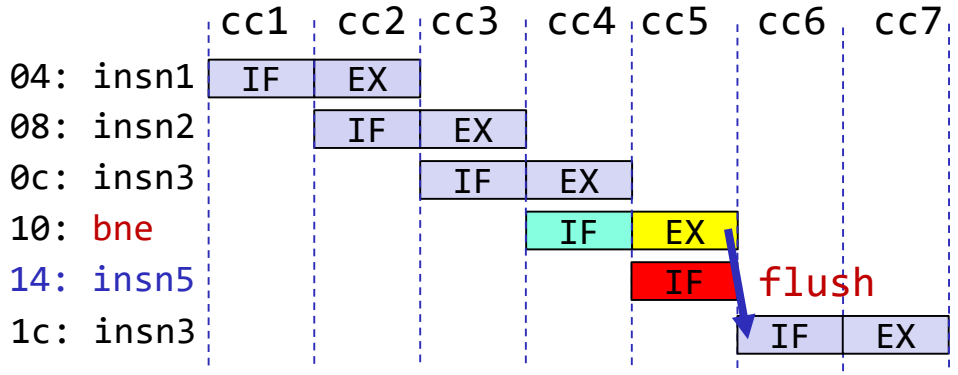
5段のパイプラインプロセッサとパイプラインレジスタ



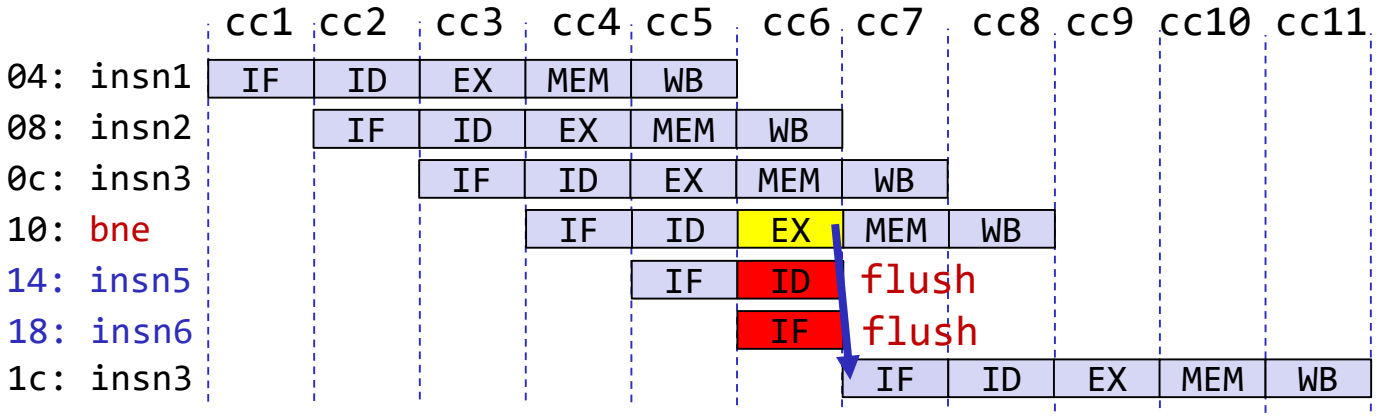
m_proc14 5段のパイプライン版の制御ハザードの対処

- 5段のパイプラインプロセッサでは，EXステージ（図ではcc6の黄色）で分岐の結果がわかる．
- この時，ID と IF で処理している命令を NOP に変更（flush，赤色）することで，無効な命令（正しくない制御の命令）を実行しない．

2段のパイプライン
分岐が成立の場合



5段のパイプライン
分岐が成立の場合



m_proc14 5段のパイプライン版のテストプログラム

- データ・ハザードを無くすために NOP を挿入する.

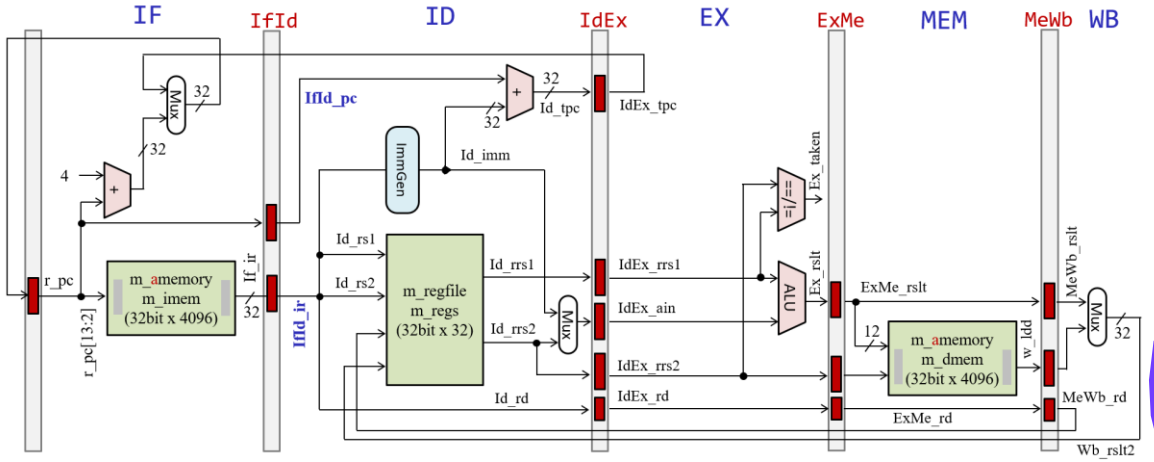
/home/tu_kise/cld/2023/baseline/program6.txt

```
initial begin
  cm_ram[ 0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011};     //    add  x0, x0, x0  // NOP
  cm_ram[ 1]={12'd5,      5'd0, 3'b000, 5'd4, 7'b0010011};     //    addi x4, x0, 5   // x4 = 5
  cm_ram[ 2]={12'd1,      5'd0, 3'b000, 5'd5, 7'b0010011};     //    addi x5, x0, 1   // x5 = 1
  cm_ram[ 3]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011};     //    add  x0, x0, x0  // NOP
  cm_ram[ 4]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011};     //    add  x0, x0, x0  // NOP
  cm_ram[ 5]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011};     //    add  x0, x0, x0  // NOP
  cm_ram[ 6]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011};     //    add  x0, x0, x0  // NOP
  cm_ram[ 7]={12'd1,      5'd5, 3'b000, 5'd5, 7'b0010011};     // L1:addi x5, x5, 1   // x5 = x5 + 1
  cm_ram[ 8]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011};     //    add  x0, x0, x0  // NOP
  cm_ram[ 9]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011};     //    add  x0, x0, x0  // NOP
  cm_ram[10]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011};     //    add  x0, x0, x0  // NOP
  cm_ram[11]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011};     //    add  x0, x0, x0  // NOP
  cm_ram[12]={7'h7f,5'd4, 5'd5, 3'b001, 5'b0110_1, 7'b1100011};//    bne  x5, x4, L1  // goto L1 if x5!=x4
  cm_ram[13]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011};     //    add  x0, x0, x0  // NOP
  cm_ram[14]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011};     //    add  x0, x0, x0  // NOP
  cm_ram[15]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011};     //    add  x0, x0, x0  // NOP
  cm_ram[16]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011};     //    add  x0, x0, x0  // NOP
  cm_ram[17]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011};     //    add  x30,x5, x0  // led = x5
  cm_ram[18]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011};     //    add  x0, x0, x0  // NOP
  cm_ram[19]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011};     //    add  x0, x0, x0  // NOP
end
```

m_proc14 5段のパイプライン版

- ステージ IF と ID の間のパイプラインレジスタには IfId_ から始まる名前を利用する.
- ステージ ID で生成される配線には Id_ から始まる名前を利用する.

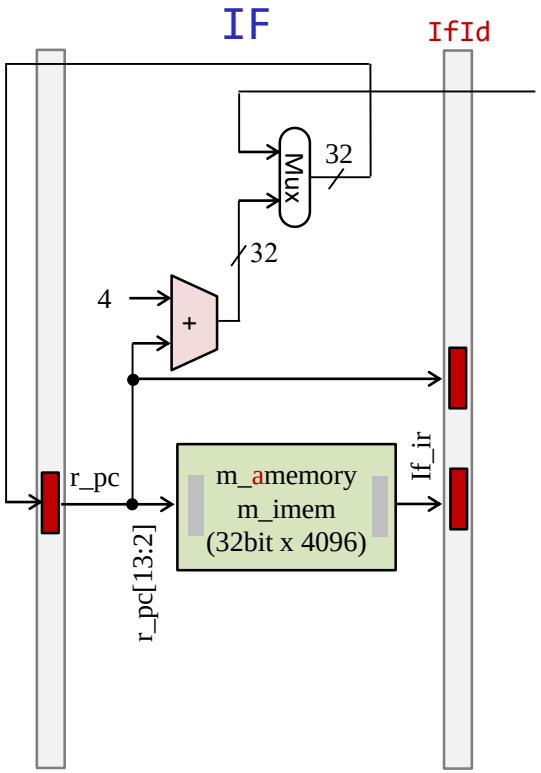
```
module m_proc14 (w_clk, w_ce, w_led);
  input  wire w_clk, w_ce;
  output wire [31:0] w_led;

  reg [31:0] r_pc      = 0; // program counter
  reg [31:0] IfId_pc   = 0; // IfId pipeline registers
  reg [31:0] IfId_ir   = 0;

  reg [31:0] IdEx_pc   = 0; // IdEx pipeline registers
  reg [31:0] IdEx_ir   = 0;
  reg [31:0] IdEx_tpc  = 0;
  reg [31:0] IdEx_rrs1 = 0;
  reg [31:0] IdEx_rrs2 = 0;
  reg [31:0] IdEx_imm  = 0;
  reg [4:0]  IdEx_rd   = 0;

  reg [31:0] ExMe_pc   = 0; // ExMe pipeline registers
  reg [31:0] ExMe_ir   = 0;
  reg [31:0] ExMe_rslt = 0;
  reg [31:0] ExMe_rrs2 = 0;
  reg [4:0]  ExMe_rd   = 0;

  reg [31:0] MeWb_pc   = 0; // MeWb pipeline registers
  reg [31:0] MeWb_ir   = 0;
  reg [31:0] MeWb_rslt = 0;
  reg [31:0] MeWb_ldd  = 0;
  reg [4:0]  MeWb_rd   = 0;
```



/home/tu_kise/cld/2023/code185.v

m_proc14 5段のパイプライン版

- ステージ IF と ID の間のパイプラインレジスタには IfId_ から始まる名前を利用する.
- ステージ ID で生成される配線には Id_ から始まる名前を利用する.

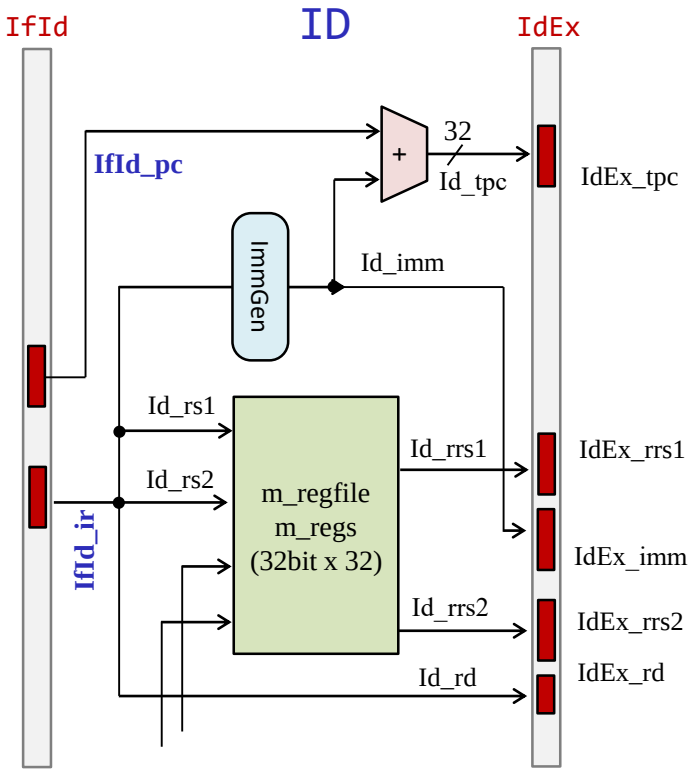



```
/********************** IF stage *******************************/
 wire [31:0] If_ir;
 m_amemory m_imem (w_clk, r_pc[13:2], 1'd0, 32'd0, If_ir);
 always @(posedge w_clk) #5 if(w_ce) begin
   IfId_pc <= (Ex_taken) ? 0 : r_pc;
   IfId_ir <= (Ex_taken) ? {25'd0, 7'b0010011} : If_ir;
 end
 /********************** ID stage *******************************/
 wire [4:0] Id_op5 = IfId_ir[6:2];
 wire [4:0] Id_rs1 = IfId_ir[19:15];
 wire [4:0] Id_rs2 = IfId_ir[24:20];
 wire [4:0] Id_rd  = IfId_ir[11:7];
 wire Id_we = (Id_op5==5'b01100 || Id_op5==5'b00100 || Id_op5==5'b00000);
 wire [31:0] Id_imm, Id_rrs1, Id_rrs2;
 m_immgen m_immgen0 (IfId_ir, Id_imm);
 m_regfile m_regs (w_clk, Id_rs1, Id_rs2, MeWb_rd, 1'b1, Wb_rslt2,
    Id_rrs1, Id_rrs2);
 always @(posedge w_clk) #5 if(w_ce) begin
   IdEx_pc   <= (Ex_taken) ? 0 : IfId_pc;
   IdEx_ir   <= (Ex_taken) ? {25'd0, 7'b0010011} : IfId_ir;
   IdEx_tpc  <= IfId_pc + Id_imm;
   IdEx_imm  <= Id_imm;
   IdEx_rrs1 <= Id_rrs1;
   IdEx_rrs2 <= Id_rrs2;
   IdEx_rd   <= (Id_we==0 || Ex_taken) ? 0 : Id_rd; // Note
 end
```

/home/tu_kise/cld/2023/code185.v

m_proc14 5段のパイプライン版

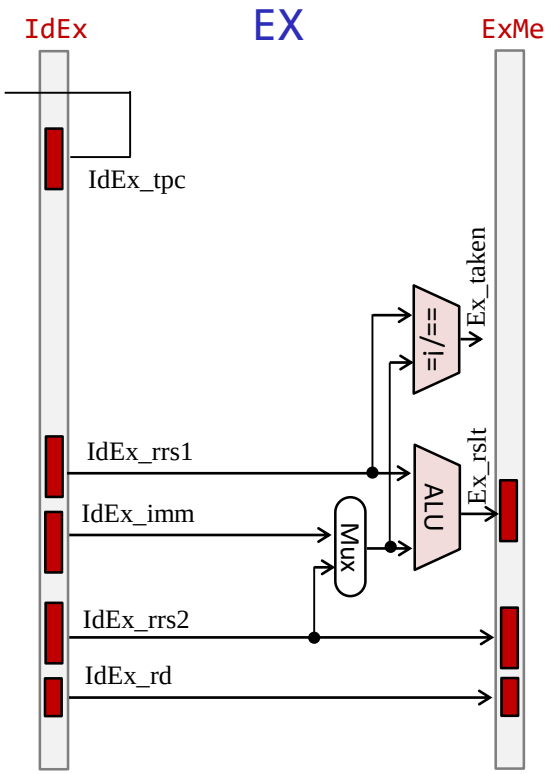
- ステージ IF と ID の間のパイプラインレジスタには IfId_ から始まる名前を利用する.
- ステージ ID で生成される配線には Id_ から始まる名前を利用する.




```
/********************** IF stage *******************************/
 wire [31:0] If_ir;
 m_amemory m_imem (w_clk, r_pc[13:2], 1'd0, 32'd0, If_ir);
 always @(posedge w_clk) #5 if(w_ce) begin
   IfId_pc <= (Ex_taken) ? 0 : r_pc;
   IfId_ir <= (Ex_taken) ? {25'd0, 7'b0010011} : If_ir;
 end
 /********************** ID stage *******************************/
 wire [4:0] Id_op5 = IfId_ir[6:2];
 wire [4:0] Id_rs1 = IfId_ir[19:15];
 wire [4:0] Id_rs2 = IfId_ir[24:20];
 wire [4:0] Id_rd  = IfId_ir[11:7];
 wire Id_we = (Id_op5==5'b01100 || Id_op5==5'b00100 || Id_op5==5'b00000);
 wire [31:0] Id_imm, Id_rrs1, Id_rrs2;
 m_immgen m_immgen0 (IfId_ir, Id_imm);
 m_regfile m_regs (w_clk, Id_rs1, Id_rs2, MeWb_rd, 1'b1, Wb_rslt2,
    Id_rrs1, Id_rrs2);
 always @(posedge w_clk) #5 if(w_ce) begin
   IdEx_pc   <= (Ex_taken) ? 0 : IfId_pc;
   IdEx_ir   <= (Ex_taken) ? {25'd0, 7'b0010011} : IfId_ir;
   IdEx_tpc  <= IfId_pc + Id_imm;
   IdEx_imm  <= Id_imm;
   IdEx_rrs1 <= Id_rrs1;
   IdEx_rrs2 <= Id_rrs2;
   IdEx_rd   <= (Id_we==0 || Ex_taken) ? 0 : Id_rd; // Note
 end
```

/home/tu_kise/cld/2023/code185.v

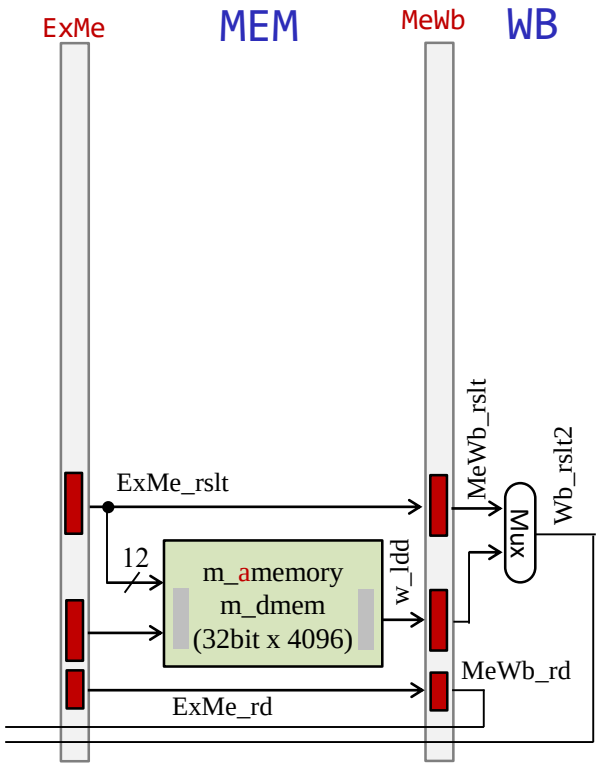
m_proc14 5段のパイプライン版




```
/********************** Ex stage ********************************/
 wire [4:0] Ex_op5 = IdEx_ir[6:2];
 wire Ex_SLL = (IdEx_ir[14:12]==3'b001);
 wire Ex_SRL = (IdEx_ir[14:12]==3'b101);
 wire Ex_BEQ = ({IdEx_ir[12], Ex_op5}==6'b011000);
 wire Ex_BNE = ({IdEx_ir[12], Ex_op5}==6'b111000);
 wire [31:0] Ex_ain = (Ex_op5==5'b01100 || Ex_op5==5'b11000) ? IdEx_rrs2 : IdEx_imm;
 wire [31:0] Ex_rslt = (Ex_SLL) ? IdEx_rrs1 << Ex_ain[4:0] :
                       (Ex_SRL) ? IdEx_rrs1 >> Ex_ain[4:0] : IdEx_rrs1 + Ex_ain;
 wire Ex_taken = (Ex_BEQ & IdEx_rrs1==Ex_ain) || (Ex_BNE & IdEx_rrs1!=Ex_ain);
 always @(posedge w_clk) #5 if(w_ce) begin
   ExMe_pc   <= IdEx_pc;
   ExMe_ir   <= IdEx_ir;
   ExMe_rslt <= Ex_rslt;
   ExMe_rrs2 <= IdEx_rrs2;
   ExMe_rd   <= IdEx_rd;
 end
```

/home/tu_kise/cld/2023/code185.v

m_proc14 5段のパイプライン版




```
/********************** Me stage ********************************/
  wire [4:0] Me_op5 = ExMe_ir[6:2];
  wire       Me_we = (Me_op5==5'b01100 || Me_op5==5'b00100 || Me_op5==5'b00000);
  wire [31:0] Me_ldd;
  m_amemory m_dmem (w_clk, ExMe_rslt[13:2], (Me_op5==5'b01000), ExMe_rrs2, Me_ldd);
  always @(posedge w_clk) #5 if(w_ce) begin
     MeWb_pc   <= ExMe_pc;
     MeWb_ir   <= ExMe_ir;
     MeWb_rslt <= ExMe_rslt;
     MeWb_ldd  <= Me_ldd;
     MeWb_rd   <= ExMe_rd;
  end
  /********************** Wb stage ********************************/
  wire Wb_LW = (MeWb_ir[6:2]==5'b00000);
  wire [31:0] Wb_rslt2 = (Wb_LW) ? MeWb_ldd : MeWb_rslt;
  always @(posedge w_clk) #5
    if(w_ce && IfId_ir!=32'h000f0033) r_pc <= (Ex_taken) ? IdEx_tpc : r_pc+4;

  reg [31:0] r_led = 0;
  always @(posedge w_clk) if(w_ce & MeWb_rd==30) r_led <= Wb_rslt2;
  assign w_led = r_led;
endmodule
```

/home/tu_kise/cld/2023/code185.v

m_proc14 5段のパイプライン版の実行結果

t の列は Ex_taken を表示

```
clock: r_pc     IfId_pc  IdEx_pc  ExMe_pc  MeWb_pc : t Id_rrs1  Id_rrs2  Ex_ain   Ex_rslt  Wb_rslt2 w_led
    1: 00000000 00000000 00000000 00000000 00000000: 0 00000000 00000000 00000000 00000000 00000000 00000000
    2: 00000004 00000000 00000000 00000000 00000000: 0 00000000 00000000 00000000 00000000 00000033 00000000
    3: 00000008 00000004 00000000 00000000 00000000: 0 00000000 xxxxxxxx 00000000 00000000 00000033 00000000
    4: 0000000c 00000008 00000004 00000000 00000000: 0 00000000 xxxxxxxx 00000005 00000005 00000033 00000000
    5: 00000010 0000000c 00000008 00000004 00000000: 0 00000000 00000000 00000001 00000001 00000000 00000000
    6: 00000014 00000010 0000000c 00000008 00000004: 0 00000000 00000000 00000000 00000000 00000005 00000000
    7: 00000018 00000014 00000010 0000000c 00000008: 0 00000000 00000000 00000000 00000000 00000001 00000000
    8: 0000001c 00000018 00000014 00000010 0000000c: 0 00000000 00000000 00000000 00000000 00000000 00000000
    9: 00000020 0000001c 00000018 00000014 00000010: 0 00000001 xxxxxxxx 00000000 00000000 00000000 00000000
   10: 00000024 00000020 0000001c 00000018 00000014: 0 00000000 00000000 00000001 00000002 00000000 00000000
   11: 00000028 00000024 00000020 0000001c 00000018: 0 00000000 00000000 00000000 00000000 00000000 00000000
   12: 0000002c 00000028 00000024 00000020 0000001c: 0 00000000 00000000 00000000 00000000 00000002 00000000
   13: 00000030 0000002c 00000028 00000024 00000020: 0 00000000 00000000 00000000 00000000 00000000 00000000
   14: 00000034 00000030 0000002c 00000028 00000024: 0 00000002 00000005 00000000 00000000 00000000 00000000
   15: 00000038 00000034 00000030 0000002c 00000028: 1 00000000 00000000 00000005 00000040 00000000 00000000
   16: 0000001c 00000000 00000000 00000030 0000002c: 0 00000000 00000000 00000000 00000000 00000000 00000000
   17: 00000020 0000001c 00000000 00000000 00000030: 0 00000002 xxxxxxxx 00000000 00000000 00000040 00000000
   18: 00000024 00000020 0000001c 00000000 00000000: 0 00000000 00000000 00000001 00000003 00000000 00000000
   19: 00000028 00000024 00000020 0000001c 00000000: 0 00000000 00000000 00000000 00000000 00000000 00000000
   20: 0000002c 00000028 00000024 00000020 0000001c: 0 00000000 00000000 00000000 00000000 00000003 00000000
   21: 00000030 0000002c 00000028 00000024 00000020: 0 00000000 00000000 00000000 00000000 00000000 00000000
   22: 00000034 00000030 0000002c 00000028 00000024: 0 00000003 00000005 00000000 00000000 00000000 00000000
   23: 00000038 00000034 00000030 0000002c 00000028: 1 00000000 00000000 00000005 00000060 00000000 00000000
   24: 0000001c 00000000 00000000 00000030 0000002c: 0 00000000 00000000 00000000 00000000 00000000 00000000
   25: 00000020 0000001c 00000000 00000000 00000030: 0 00000003 xxxxxxxx 00000000 00000000 00000060 00000000
   26: 00000024 00000020 0000001c 00000000 00000000: 0 00000000 00000000 00000001 00000004 00000000 00000000
   27: 00000028 00000024 00000020 0000001c 00000000: 0 00000000 00000000 00000000 00000000 00000000 00000000
   28: 0000002c 00000028 00000024 00000020 0000001c: 0 00000000 00000000 00000000 00000000 00000004 00000000
   29: 00000030 0000002c 00000028 00000024 00000020: 0 00000000 00000000 00000000 00000000 00000000 00000000
   30: 00000034 00000030 0000002c 00000028 00000024: 0 00000004 00000005 00000000 00000000 00000000 00000000
   31: 00000038 00000034 00000030 0000002c 00000028: 1 00000000 00000000 00000005 00000080 00000000 00000000
   32: 0000001c 00000000 00000000 00000030 0000002c: 0 00000000 00000000 00000000 00000000 00000000 00000000
   33: 00000020 0000001c 00000000 00000000 00000030: 0 00000004 xxxxxxxx 00000000 00000000 00000080 00000000
   34: 00000024 00000020 0000001c 00000000 00000000: 0 00000000 00000000 00000001 00000005 00000000 00000000
   35: 00000028 00000024 00000020 0000001c 00000000: 0 00000000 00000000 00000000 00000000 00000000 00000000
   36: 0000002c 00000028 00000024 00000020 0000001c: 0 00000000 00000000 00000000 00000000 00000005 00000000
   37: 00000030 0000002c 00000028 00000024 00000020: 0 00000000 00000000 00000000 00000000 00000000 00000000
   38: 00000034 00000030 0000002c 00000028 00000024: 0 00000005 00000005 00000000 00000000 00000000 00000000
   39: 00000038 00000034 00000030 0000002c 00000028: 0 00000000 00000000 00000005 000000a0 00000000 00000000
   40: 0000003c 00000038 00000034 00000030 0000002c: 0 00000000 00000000 00000000 00000000 00000000 00000000
   41: 00000040 0000003c 00000038 00000034 00000030: 0 00000000 00000000 00000000 00000000 000000a0 00000000
   42: 00000044 00000040 0000003c 00000038 00000034: 0 00000000 00000000 00000000 00000000 00000000 00000000
   43: 00000048 00000044 00000040 0000003c 00000038: 0 00000005 00000000 00000000 00000000 00000000 00000000
   44: 0000004c 00000048 00000044 00000040 0000003c: 0 00000000 00000000 00000000 00000005 00000000 00000000
   45: 00000050 0000004c 00000048 00000044 00000040: 0 00000000 00000000 00000000 00000000 00000000 00000000
   46: 00000054 00000050 0000004c 00000048 00000044: 0 xxxxxxxx xxxxxxxx 00000000 00000000 00000005 00000000
   47: 00000054 00000054 00000050 0000004c 00000048: x xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx 00000000 00000005
```

/home/tu_kise/
cld/2023/baseline/
program6.txt
の実行結果

Hazards make pipelining hard (topic of the next lecture)

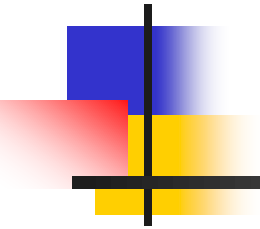
- 命令を適切なサイクルで実行できないような状況が存在する．これをハザード(hazard)と呼ぶ．
  - 構造ハザード (structural hazard)
    - オーバラップ実行する命令の組み合わせをハードウェアがサポートしていない場合．資源不足により生じる．
  - データ・ハザード(data hazard)
    - データの受け渡しの制約によって生じるハザード

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //    add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd4, 7'b0010011}; //    addi x4, x0, 3   // x4 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //    addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd5, 5'd4, 3'b000, 5'd30,7'b0110011}; //    add  x30,x4, x5  // led = x4 + x5
```

  - 制御ハザード(control hazard)
    - 分岐命令，ジャンプ命令によって生じるハザード

```
cm_ram[1]={12'd5,      5'd0, 3'b000, 5'd4, 7'b0010011};     //    addi x4, x0, 5   // x4 = 5
cm_ram[2]={12'd1,      5'd0, 3'b000, 5'd5, 7'b0010011};     //    addi x5, x0, 1   // x5 = 1
cm_ram[3]={12'd1,      5'd5, 3'b000, 5'd5, 7'b0010011};     // L1:addi x5, x5, 1   // x5 = x5 + 1
cm_ram[4]={7'h7f,5'd4, 5'd5, 3'b001, 5'b11101, 7'b1100011}; //    bne  x5, x4, L1  // goto L1 if x5!=x4
cm_ram[5]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011};     //    add  x30,x5, x0  // led = x5
```

Department of Computer Science
Course number: CSC.T341



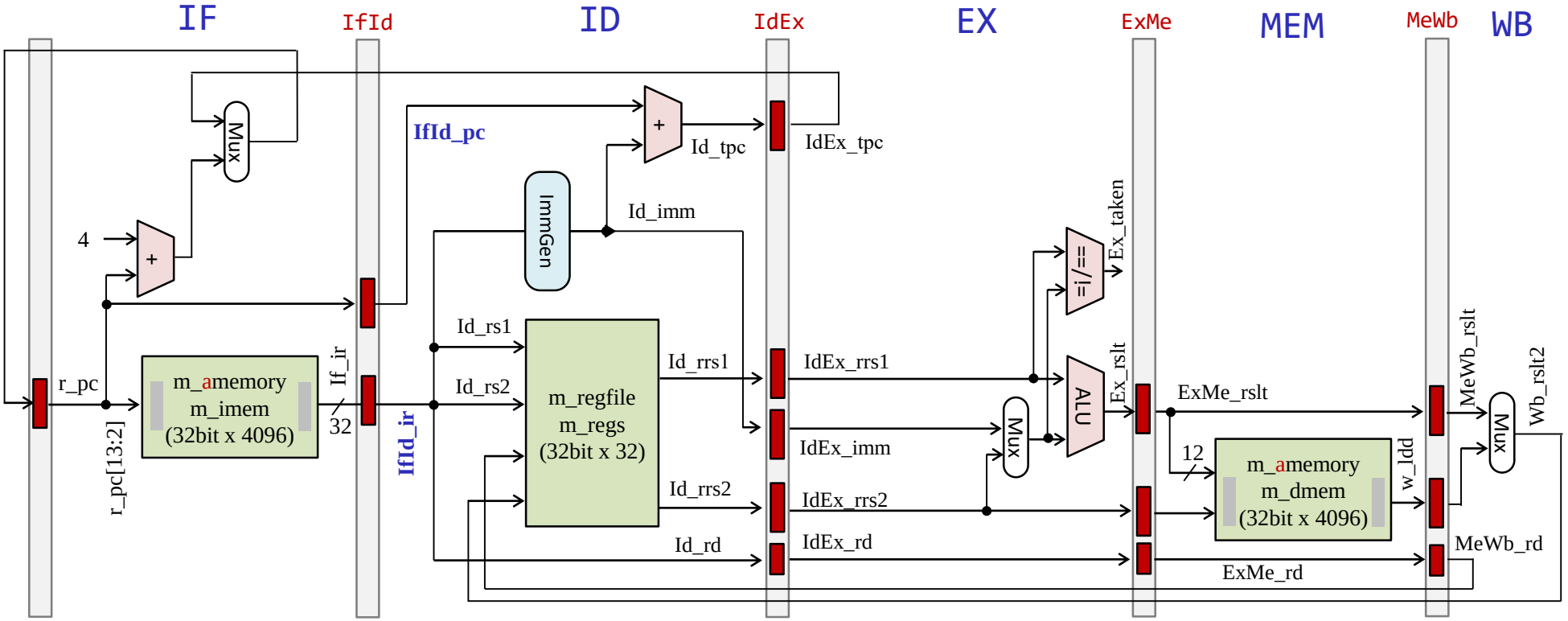
コンピュータ論理設計
Computer Logic Design

12. パイプラインプロセッサとハザード処理 (2)
Pipelining Processor and Hazards (2)

吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
講義：月曜日 10:45-12:25, 木曜日 10:45-12:25

m_proc14 5段のパイプライン版

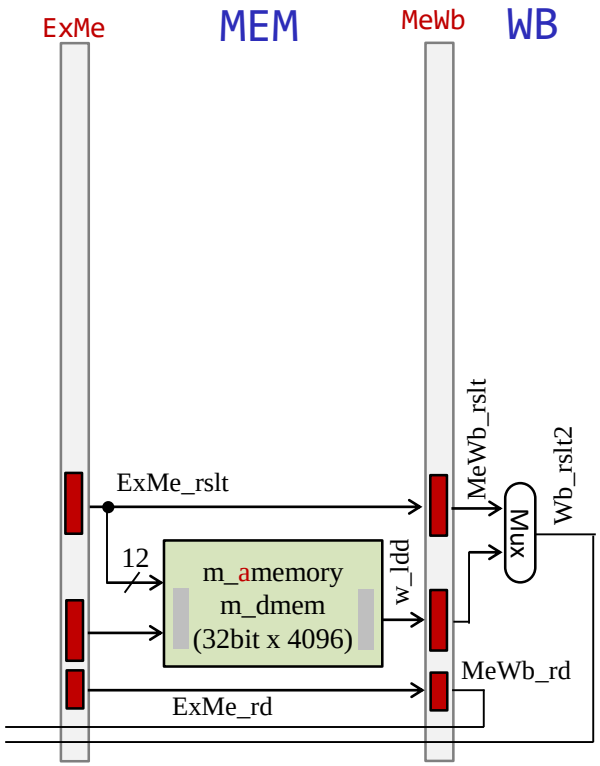
- add, addi, sll, srl, lw, sw, beq, bne命令に対応したプロセッサ（データフォワーディング無し）
- IF, ID, EX, MEM, WB のそれぞれをステージとする5段のパイプラインプロセッサ.
- m_amemory を利用しているが，その出力がレジスタに接続される dmem は同期メモリとして動作する. このため，BRAMが利用される.



code185.v

m_proc14 5段のパイプライン版

- m_amemory を利用しているが，その出力がレジスタに接続される dmem は同期メモリとして動作する．このため，BRAMが利用される．



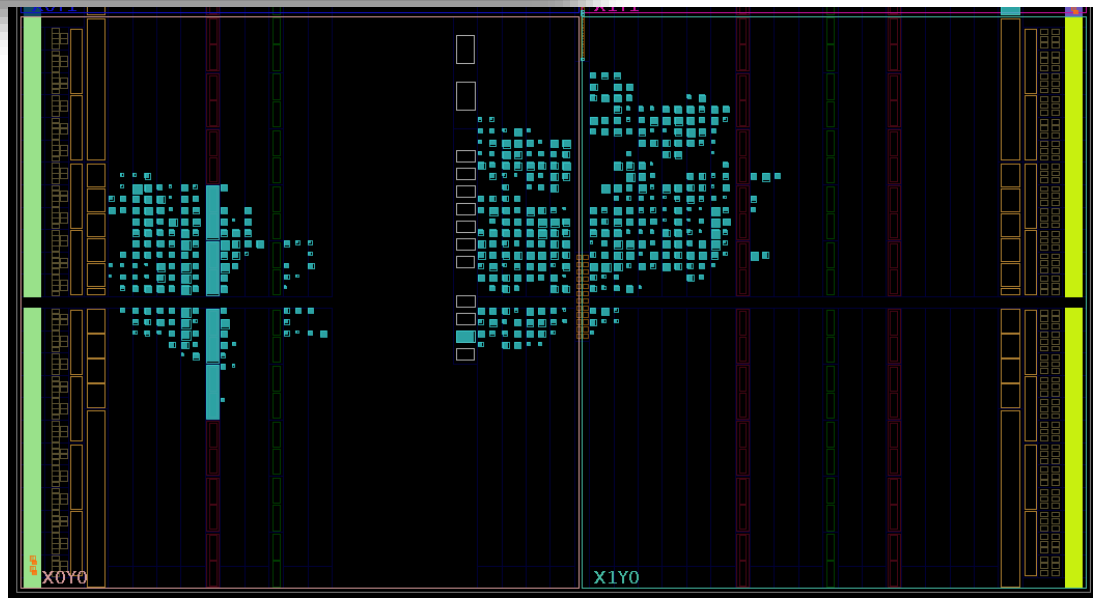

```
/********************** Me stage ********************************/
  wire [4:0] Me_op5 = ExMe_ir[6:2];
  wire       Me_we = (Me_op5==5'b01100 || Me_op5==5'b00100 || Me_op5==5'b00000);
  wire [31:0] Me_ldd;
  m_amemory m_dmem (w_clk, ExMe_rslt[13:2], (Me_op5==5'b01000), ExMe_rrs2, Me_ldd);
  always @(posedge w_clk) #5 if(w_ce) begin
     MeWb_pc   <= ExMe_pc;
     MeWb_ir   <= ExMe_ir;
     MeWb_rslt <= ExMe_rslt;
     MeWb_ldd  <= Me_ldd;
     MeWb_rd   <= ExMe_rd;
  end
  /********************** Wb stage ********************************/
  wire Wb_LW = (MeWb_ir[6:2]==5'b00000);
  wire [31:0] Wb_rslt2 = (Wb_LW) ? MeWb_ldd : MeWb_rslt;
  always @(posedge w_clk) #5
    if(w_ce && IfId_ir!=32'h000f0033) r_pc <= (Ex_taken) ? IdEx_tpc : r_pc+4;

  reg [31:0] r_led = 0;
  always @(posedge w_clk) if(w_ce & MeWb_rd==30) r_led <= Wb_rslt2;
  assign w_led = r_led;
endmodule
```

/home/tu_kise/cld/2023/code185.v

m_proc14 5段のパイプライン版

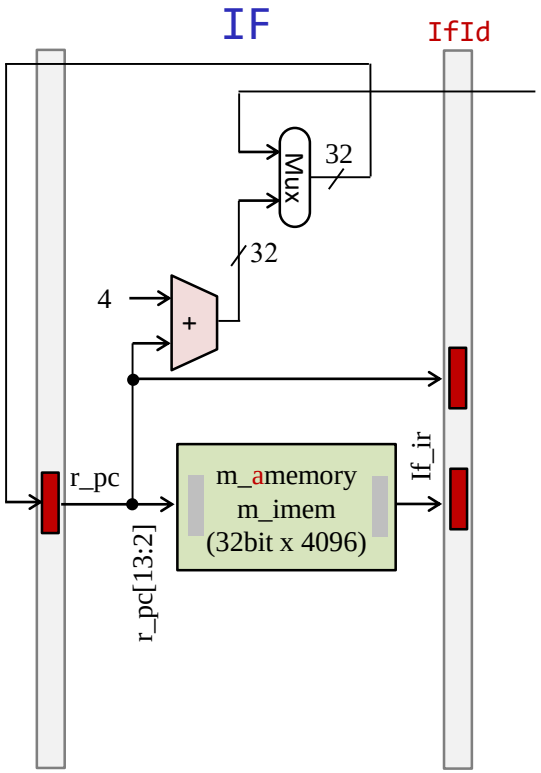
- add, addi, sll, srl, lw, sw, beq, bne命令に対応したプロセッサ（データフォワーディング無し）
- IF, ID, EX, MEM, WB のそれぞれをステージとする5段のパイプラインプロセッサ.
- m_amemory を利用しているが，その出力がレジスタに接続される dmem は同期メモリとして動作する．このため，BRAMが利用される.

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	Block RAM Tile (50)
N m_main	1042	1445	40	20	462	894	148	4
clk_w0 (clk_wiz_0)	0	0	0	0	0	0	0	0
dbg_hub (dbg_hub)	448	741	0	0	228	424	24	0
p (m_proc14)	381	292	40	20	138	257	124	4
m_dmem (m_amem	33	0	0	0	14	33	0	4
m_imem (m_amemc	100	0	40	20	29	20	80	0
m_regs (m_regfile)	44	0	0	0	11	0	44	0
vio_00 (vio_0)	205	408	0	0	108	205	0	0



m_proc14 5段のパイプライン版

- ステージ IF と ID の間のパイプラインレジスタには IfId_ から始まる名前を利用する.
- ステージ ID で生成される配線には Id_ から始まる名前を利用する.




```
/********************** IF stage *******************************/
 wire [31:0] If_ir;
 m_amemory m_imem (w_clk, r_pc[13:2], 1'd0, 32'd0, If_ir);
 always @(posedge w_clk) #5 if(w_ce) begin
   IfId_pc <= (Ex_taken) ? 0 : r_pc;
   IfId_ir <= (Ex_taken) ? {25'd0, 7'b0010011} : If_ir;
 end
 /********************** ID stage *******************************/
 wire [4:0] Id_op5 = IfId_ir[6:2];
 wire [4:0] Id_rs1 = IfId_ir[19:15];
 wire [4:0] Id_rs2 = IfId_ir[24:20];
 wire [4:0] Id_rd  = IfId_ir[11:7];
 wire Id_we = (Id_op5==5'b01100 || Id_op5==5'b00100 || Id_op5==5'b00000);
 wire [31:0] Id_imm, Id_rrs1, Id_rrs2;
 m_immgen m_immgen0 (IfId_ir, Id_imm);
 m_regfile m_regs (w_clk, Id_rs1, Id_rs2, MeWb_rd, 1'b1, Wb_rslt2,
    Id_rrs1, Id_rrs2);
 always @(posedge w_clk) #5 if(w_ce) begin
   IdEx_pc   <= (Ex_taken) ? 0 : IfId_pc;
   IdEx_ir   <= (Ex_taken) ? {25'd0, 7'b0010011} : IfId_ir;
   IdEx_tpc  <= IfId_pc + Id_imm;
   IdEx_imm  <= Id_imm;
   IdEx_rrs1 <= Id_rrs1;
   IdEx_rrs2 <= Id_rrs2;
   IdEx_rd   <= (Id_we==0 || Ex_taken) ? 0 : Id_rd; // Note
 end
```

/home/tu_kise/cld/2023/code185.v

Hazards make pipelining hard

- 命令を適切なサイクルで実行できないような状況が存在する．これをハザード(hazard)と呼ぶ．
  - 構造ハザード (structural hazard)
    - オーバラップ実行する命令の組み合わせをハードウェアがサポートしていない場合．資源不足により生じる．
  - データ・ハザード(data hazard)
    - データの受け渡しの制約によって生じるハザード

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //    add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd4, 7'b0010011}; //    addi x4, x0, 3   // x4 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //    addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd5, 5'd4, 3'b000, 5'd30,7'b0110011}; //    add  x30,x4, x5  // led = x4 + x5
```

  - 制御ハザード(control hazard)
    - 分岐命令，ジャンプ命令によって生じるハザード

```
cm_ram[1]={12'd5,      5'd0, 3'b000, 5'd4, 7'b0010011};     //    addi x4, x0, 5   // x4 = 5
cm_ram[2]={12'd1,      5'd0, 3'b000, 5'd5, 7'b0010011};     //    addi x5, x0, 1   // x5 = 1
cm_ram[3]={12'd1,      5'd5, 3'b000, 5'd5, 7'b0010011};     // L1:addi x5, x5, 1   // x5 = x5 + 1
cm_ram[4]={7'h7f,5'd4, 5'd5, 3'b001, 5'b11101, 7'b1100011}; //    bne  x5, x4, L1  // goto L1 if x5!=x4
cm_ram[5]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011};     //    add  x30,x5, x0  // led = x5
```

プロセッサが命令を処理するための基本的な5つのステップ

- IF (Instruction Fetch)
  メモリから命令をフェッチする.
- ID (Instruction Decode)
  命令をデコード(解読)しながら, レジスタファイルの値を読み出す (Operand Fetch)
- EX (Execution)
  命令操作の実行またはアドレスの生成を行う.
- MEM (Memory Access)
  必要であれば, メモリ(データ・メモリ)のオペランドにアクセスする.
- WB (Write Back)
  必要であれば, 結果をレジスタファイルに書き込む.

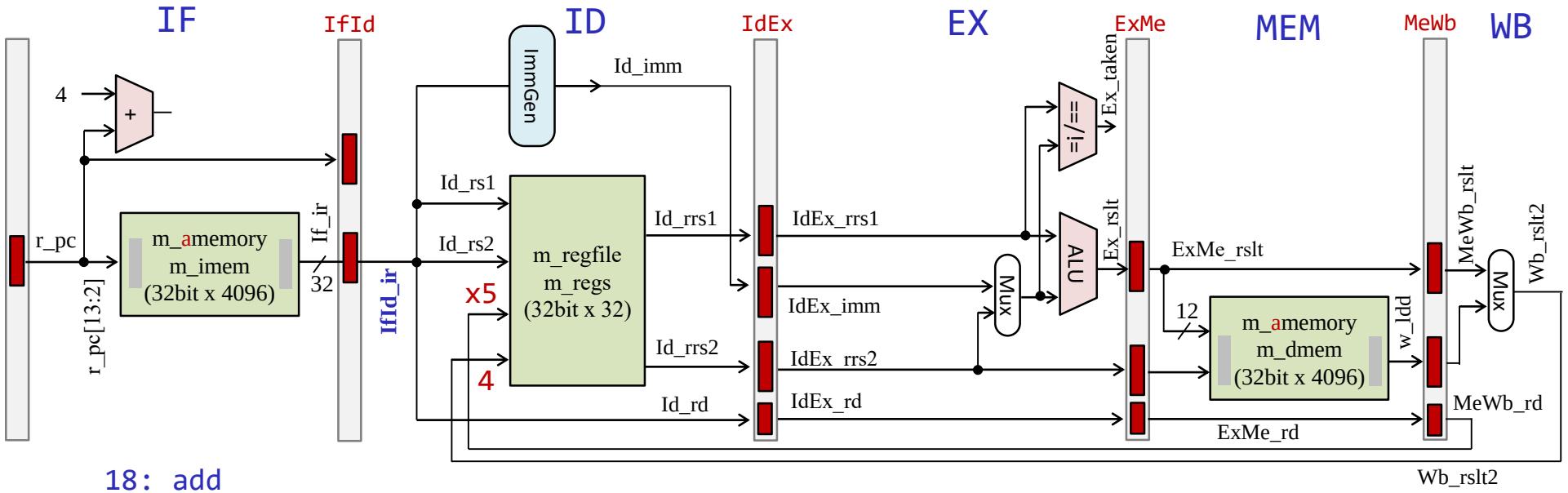
(1) m_proc14 5段のパイプライン版：データ依存の無い例

- 命令の間に3個の NOP 命令を挿入したプログラム.

program7.txt

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  00:   add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  04:   addi x5, x0, 3   // x5 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  08:   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  0c:   add  x0, x0, x0  // NOP
cm_ram[4]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  10:   add  x0, x0, x0  // NOP
cm_ram[5]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  14:   add  x0, x0, x0  // NOP
cm_ram[6]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011}; //  18:   add  x30,x5, x0  // led = x5
cm_ram[7]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  1c:   add  x0, x0, x0  // NOP
```

CC7



18: add x30,x5,x0 | 14: NOP | 10: NOP | 0c: NOP | 08: addi x5,x0,4

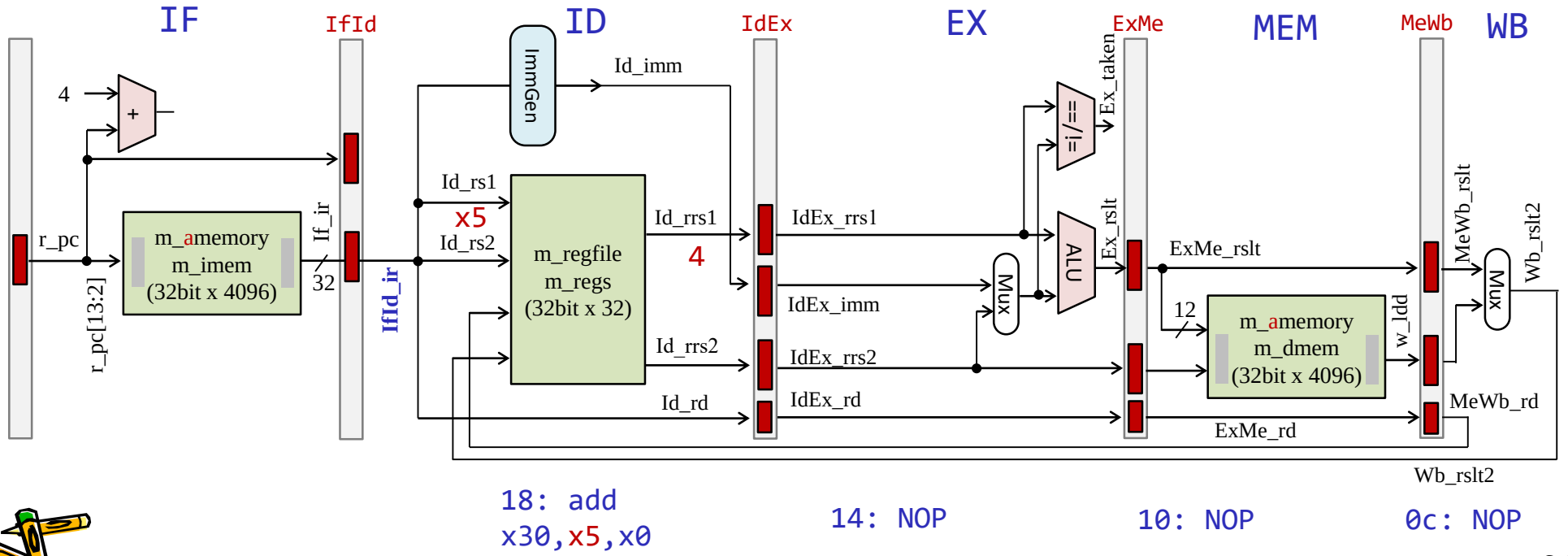
(1) m_proc14 5段のパイプライン版：データ依存の無い例

- IDステージの命令 18: add x30,x5,x0 は正しいレジスタの値4を読み出すことができる.

program7.txt

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  00:   add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  04:   addi x5, x0, 3   // x5 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  08:   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  0c:   add  x0, x0, x0  // NOP
cm_ram[4]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  10:   add  x0, x0, x0  // NOP
cm_ram[5]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  14:   add  x0, x0, x0  // NOP
cm_ram[6]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011}; //  18:   add  x30,x5, x0  // led = x5
cm_ram[7]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  1c:   add  x0, x0, x0  // NOP
```

CC8



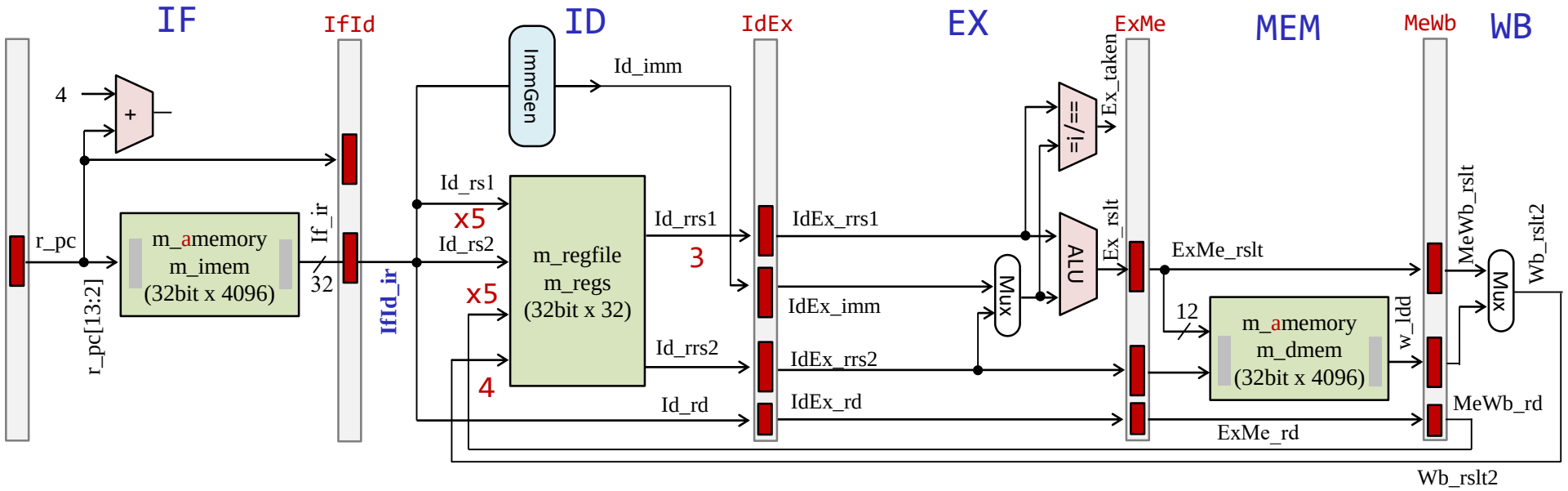
(2) m_proc14 5段のパイプライン版：レジスタバイパス

- 命令の間に2個の NOP 命令を挿入したプログラム.
- IDステージの命令 14: add x30,x5,x0 は正しくないレジスタの値3を読み出してしまう.

program8.txt

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  00:   add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  04:   addi x5, x0, 3   // x5 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  08:   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  0c:   add  x0, x0, x0  // NOP
cm_ram[4]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  10:   add  x0, x0, x0  // NOP
cm_ram[5]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011}; //  14:   add  x30,x5, x0  // led = x5
cm_ram[6]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  18:   add  x0, x0, x0  // NOP
```

CC7



14: add x30,x5,x0

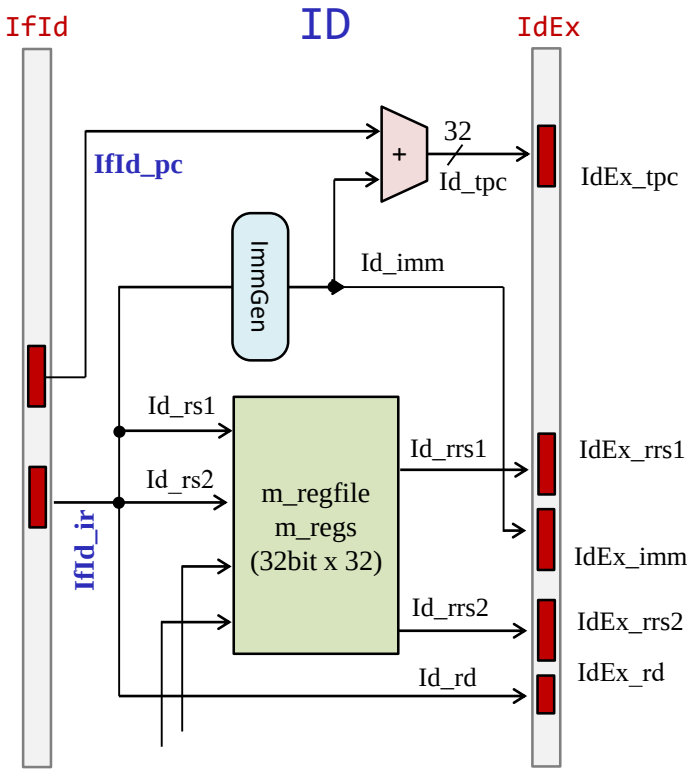
10: NOP

0c: NOP

08: addi x5,x0,4

m_proc14 5段のパイプライン版

- ステージ IF と ID の間のパイプラインレジスタには IfId_ から始まる名前を利用する.
- ステージ ID で生成される配線には Id_ から始まる名前を利用する.




```
/********************** IF stage *******************************/
 wire [31:0] If_ir;
 m_amemory m_imem (w_clk, r_pc[13:2], 1'd0, 32'd0, If_ir);
 always @(posedge w_clk) #5 if(w_ce) begin
   IfId_pc <= (Ex_taken) ? 0 : r_pc;
   IfId_ir <= (Ex_taken) ? {25'd0, 7'b0010011} : If_ir;
 end
 /********************** ID stage *******************************/
 wire [4:0] Id_op5 = IfId_ir[6:2];
 wire [4:0] Id_rs1 = IfId_ir[19:15];
 wire [4:0] Id_rs2 = IfId_ir[24:20];
 wire [4:0] Id_rd  = IfId_ir[11:7];
 wire Id_we = (Id_op5==5'b01100 || Id_op5==5'b00100 || Id_op5==5'b00000);
 wire [31:0] Id_imm, Id_rrs1, Id_rrs2;
 m_immgen m_immgen0 (IfId_ir, Id_imm);
 m_regfile m_regs (w_clk, Id_rs1, Id_rs2, MeWb_rd, 1'b1, Wb_rslt2,
                   Id_rrs1, Id_rrs2);
 always @(posedge w_clk) #5 if(w_ce) begin
   IdEx_pc   <= (Ex_taken) ? 0 : IfId_pc;
   IdEx_ir   <= (Ex_taken) ? {25'd0, 7'b0010011} : IfId_ir;
   IdEx_tpc  <= IfId_pc + Id_imm;
   IdEx_imm  <= Id_imm;
   IdEx_rrs1 <= Id_rrs1;
   IdEx_rrs2 <= Id_rrs2;
   IdEx_rd   <= (Id_we==0 || Ex_taken) ? 0 : Id_rd; // Note
 end
```

/home/tu_kise/cld/2023/code185.v

(2) レジスタファイルにバイパスの経路を追加

- 書き込みレジスタ番号と読み出しレジスタ番号が一致するときに，書き込む値を出力する．
  すなわち，書き込む値をバイパスするように修正したモジュール m_rf_bypass を用いる．

```
module m_regfile (w_clk, w_rr1, w_rr2, w_wr, w_we, w_wdata, w_rdata1, w_rdata2);
   input  wire        w_clk;
   input  wire [4:0]  w_rr1, w_rr2, w_wr;
   input  wire [31:0] w_wdata;
   input  wire        w_we;
   output wire [31:0] w_rdata1, w_rdata2;

   reg [31:0] r[0:31];
   assign #8 w_rdata1 = (w_rr1==0) ? 0 : r[w_rr1];
   assign #8 w_rdata2 = (w_rr2==0) ? 0 : r[w_rr2];
   always @(posedge w_clk) if(w_we) r[w_wr] <= w_wdata;
endmodule
```

```
module m_rf_byass (w_clk, w_rr1, w_rr2, w_wr, w_we, w_wdata, w_rdata1, w_rdata2);
   input  wire        w_clk;
   input  wire [4:0]  w_rr1, w_rr2, w_wr;
   input  wire [31:0] w_wdata;
   input  wire        w_we;
   output wire [31:0] w_rdata1, w_rdata2;

   reg [31:0] r[0:31];
   assign #8 w_rdata1 = (w_rr1==0) ? 0 : (w_rr1==w_wr & w_we) ? w_wdata : r[w_rr1];
   assign #8 w_rdata2 = (w_rr2==0) ? 0 : (w_rr2==w_wr & w_we) ? w_wdata : r[w_rr2];
   always @(posedge w_clk) if(w_we) r[w_wr] <= w_wdata;
endmodule
```

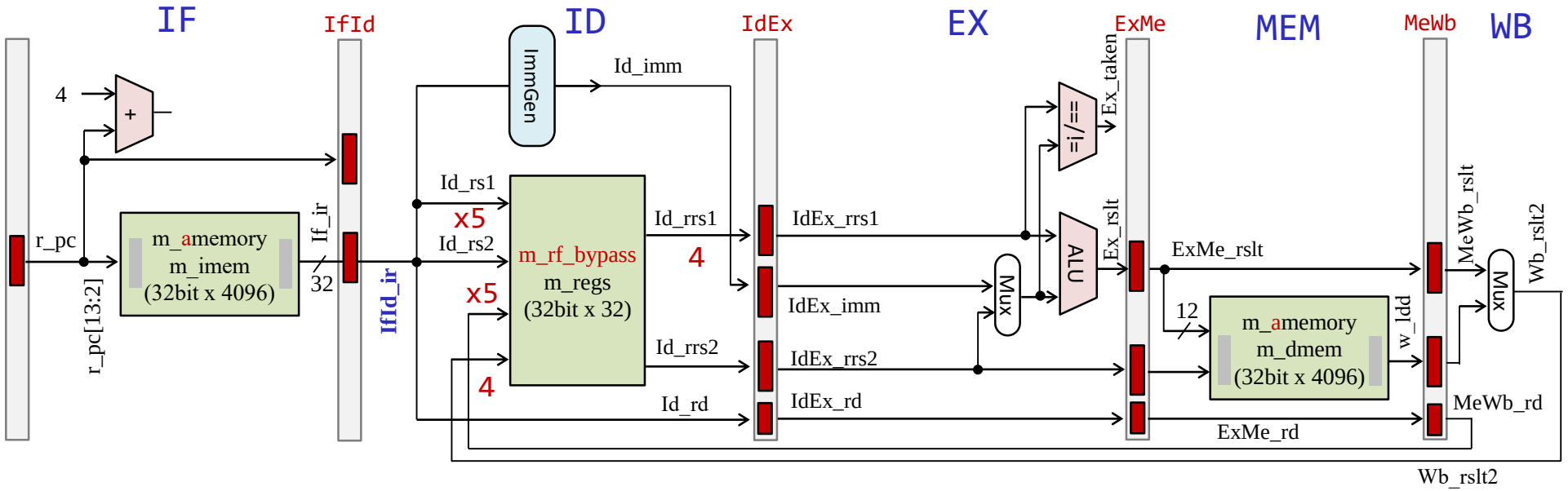
(2) m_proc14 5段のパイプライン版：レジスタバイパス

- 命令の間に2個の NOP 命令を挿入したプログラム．m_rf_bypass を用いる．
- IDステージの命令 14: add x30,x5,x0 は正しいレジスタの値4を読み出せる．

program8.txt

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  00:   add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  04:   addi x5, x0, 3   // x5 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  08:   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  0c:   add  x0, x0, x0  // NOP
cm_ram[4]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  10:   add  x0, x0, x0  // NOP
cm_ram[5]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011}; //  14:   add  x30,x5, x0  // led = x5
cm_ram[6]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  18:   add  x0, x0, x0  // NOP
```

CC7



14: add x30,x5,x0

10: NOP

0c: NOP

08: addi x5,x0,4

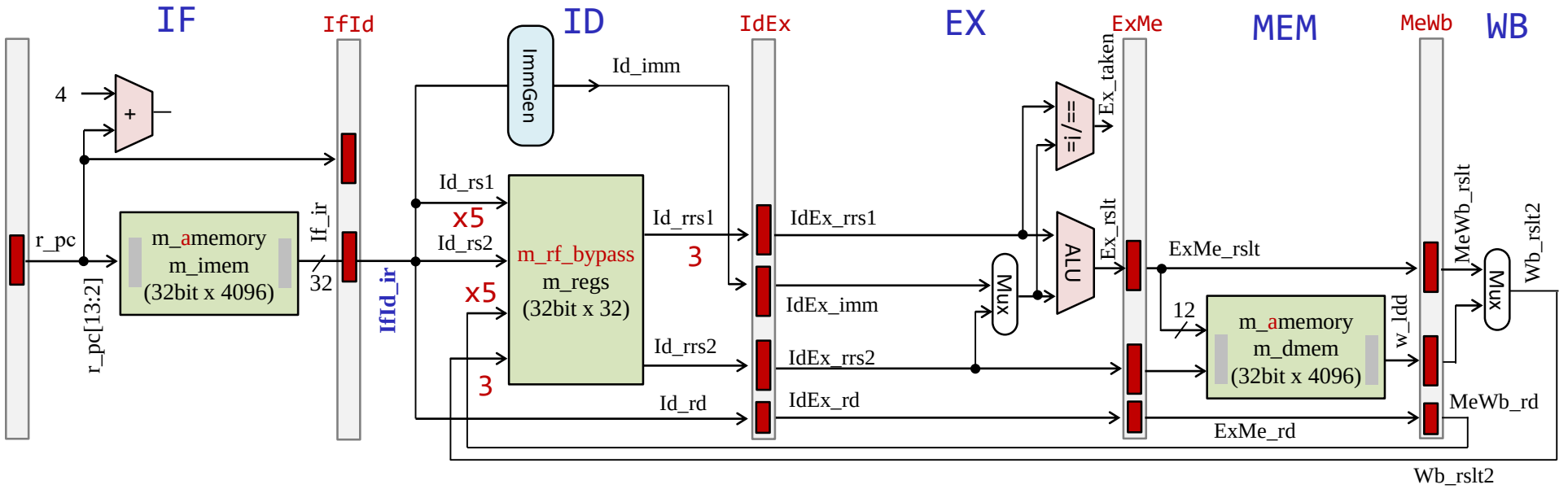
(3) m_proc14 5段のパイプライン版: WBフォワーディング

- 命令の間に1個の NOP 命令を挿入したプログラム.
- IDステージの命令 10: add x30,x5,x0 は正しくないレジスタの値3を読み出してしまう.

program9.txt

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  00:   add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  04:   addi x5, x0, 3   // x5 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  08:   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  0c:   add  x0, x0, x0  // NOP
cm_ram[4]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011}; //  10:   add  x30,x5, x0  // led = x5
cm_ram[5]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  14:   add  x0, x0, x0  // NOP
```

CC6



10: add x30,x5,x0

0c: NOP

08: addi x5,x0,4

04: addi x5,x0,3

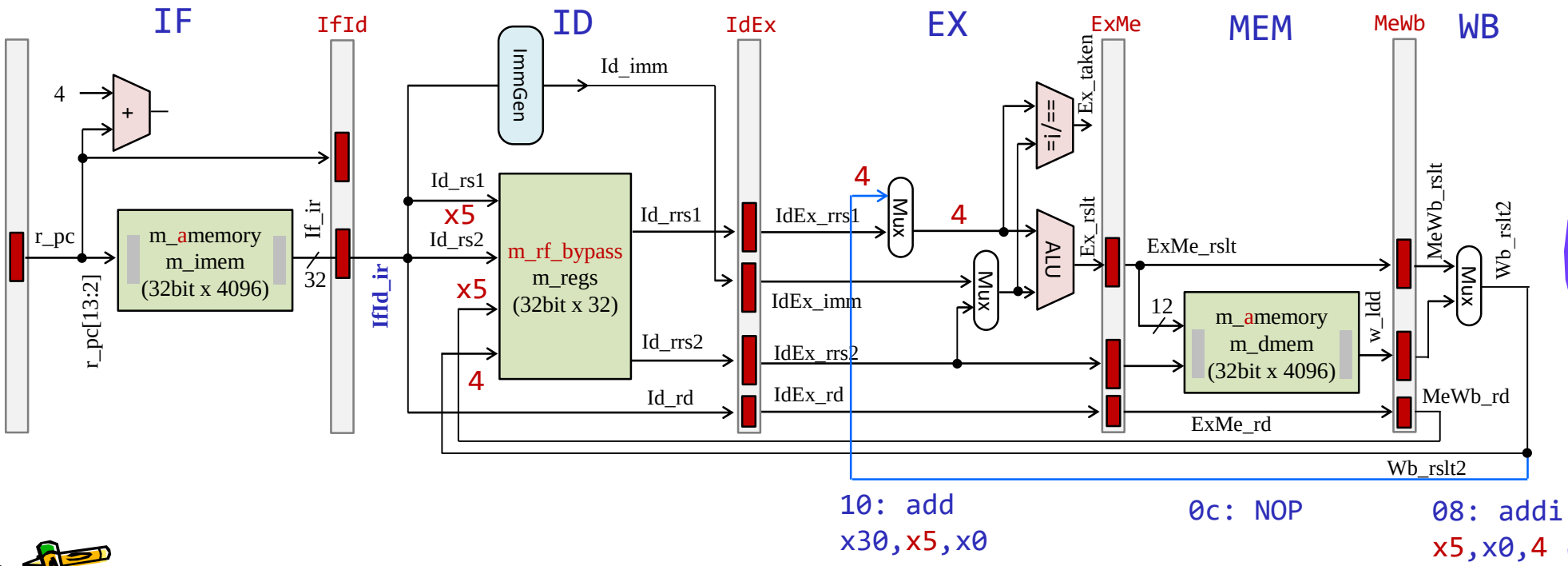
(3) m_proc14 5段のパイプライン版: WBフォワーディング

- 命令の間に1個の NOP 命令を挿入したプログラム.
- WBステージからオペランドをALUにフォワーディングする. 正しい値4を利用できる

program9.txt

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  00:   add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  04:   addi x5, x0, 3   // x5 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  08:   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  0c:   add  x0, x0, x0  // NOP
cm_ram[4]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011}; //  10:   add  x30,x5, x0  // led = x5
cm_ram[5]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  14:   add  x0, x0, x0  // NOP
```

CC7



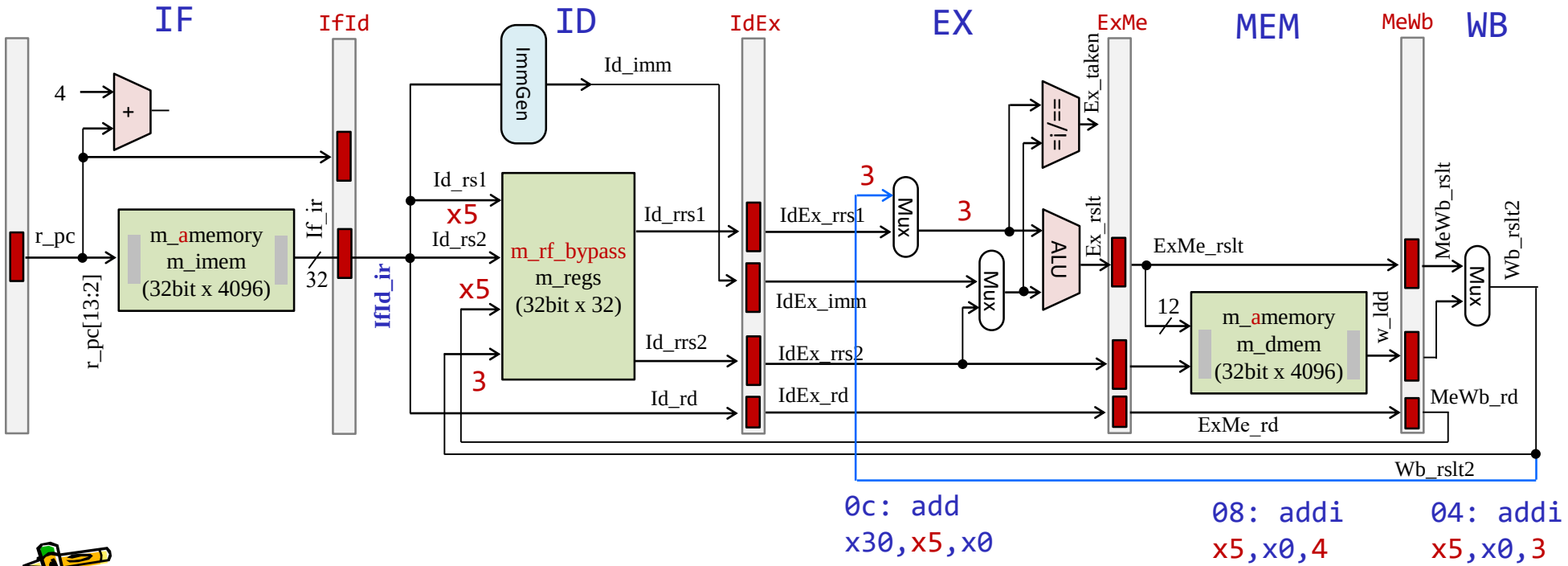
(4) m_proc14 5段のパイプライン版: MEMフォワーディング

- 命令の間に0個の NOP 命令を挿入したプログラム.
- Exステージの命令 0c: add x30,x5,x0 は正しくないレジスタの値3を読み出してしまう.

program10.txt

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  00:   add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  04:   addi x5, x0, 3   // x5 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  08:   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011}; //  0c:   add  x30,x5, x0  // led = x5
cm_ram[4]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  10:   add  x0, x0, x0  // NOP
```

CC6



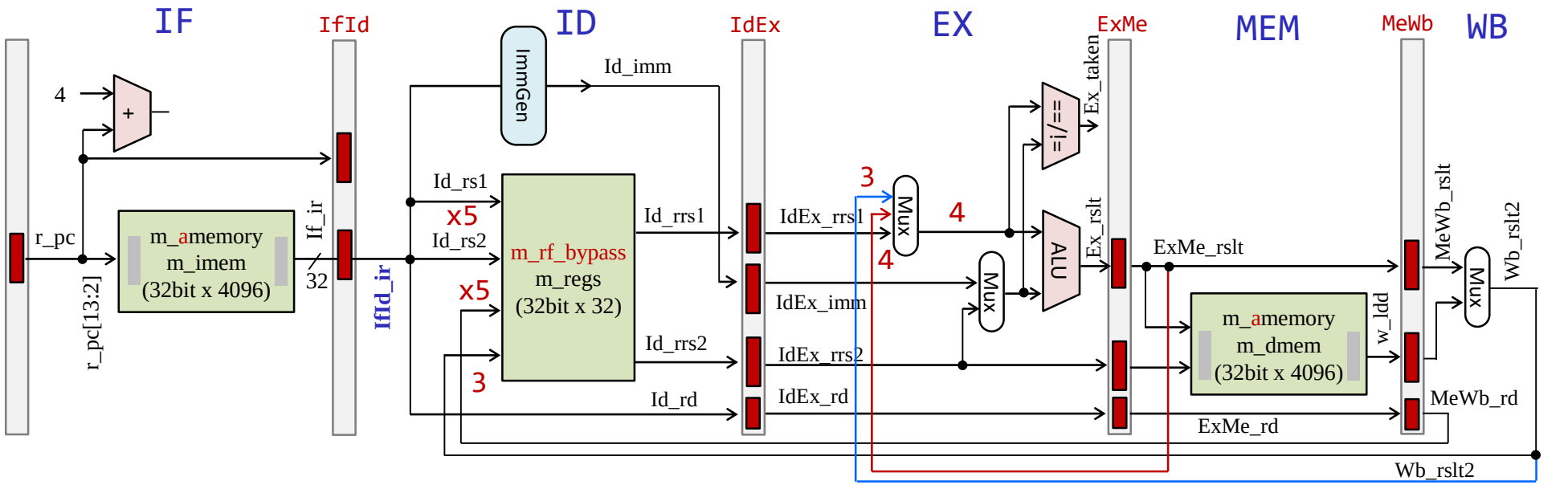
(4) m_proc14 5段のパイプライン版: MEMフォワーディング

- 命令の間に0個の NOP 命令を挿入したプログラム.
- MEMステージからオペランドをALUにフォワーディングする. 正しい値4を利用できる.

program10.txt

```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  00:   add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  04:   addi x5, x0, 3   // x5 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  08:   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd0, 5'd5, 3'b000, 5'd30,7'b0110011}; //  0c:   add  x30,x5, x0  // led = x5
cm_ram[4]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  10:   add  x0, x0, x0  // NOP
```

CC6



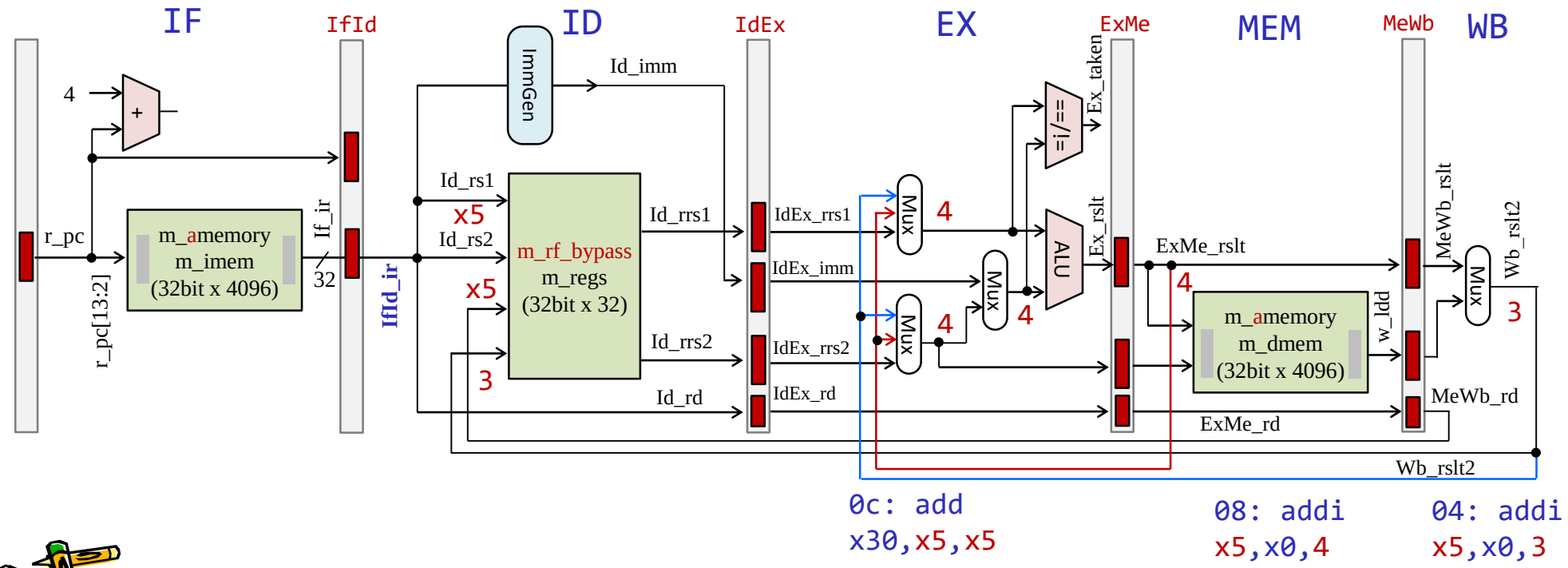
m_proc14 5段のパイプライン版

- rrs1 と同様に rrs2 のためにもデータをフォワーディングする.

program10.txt

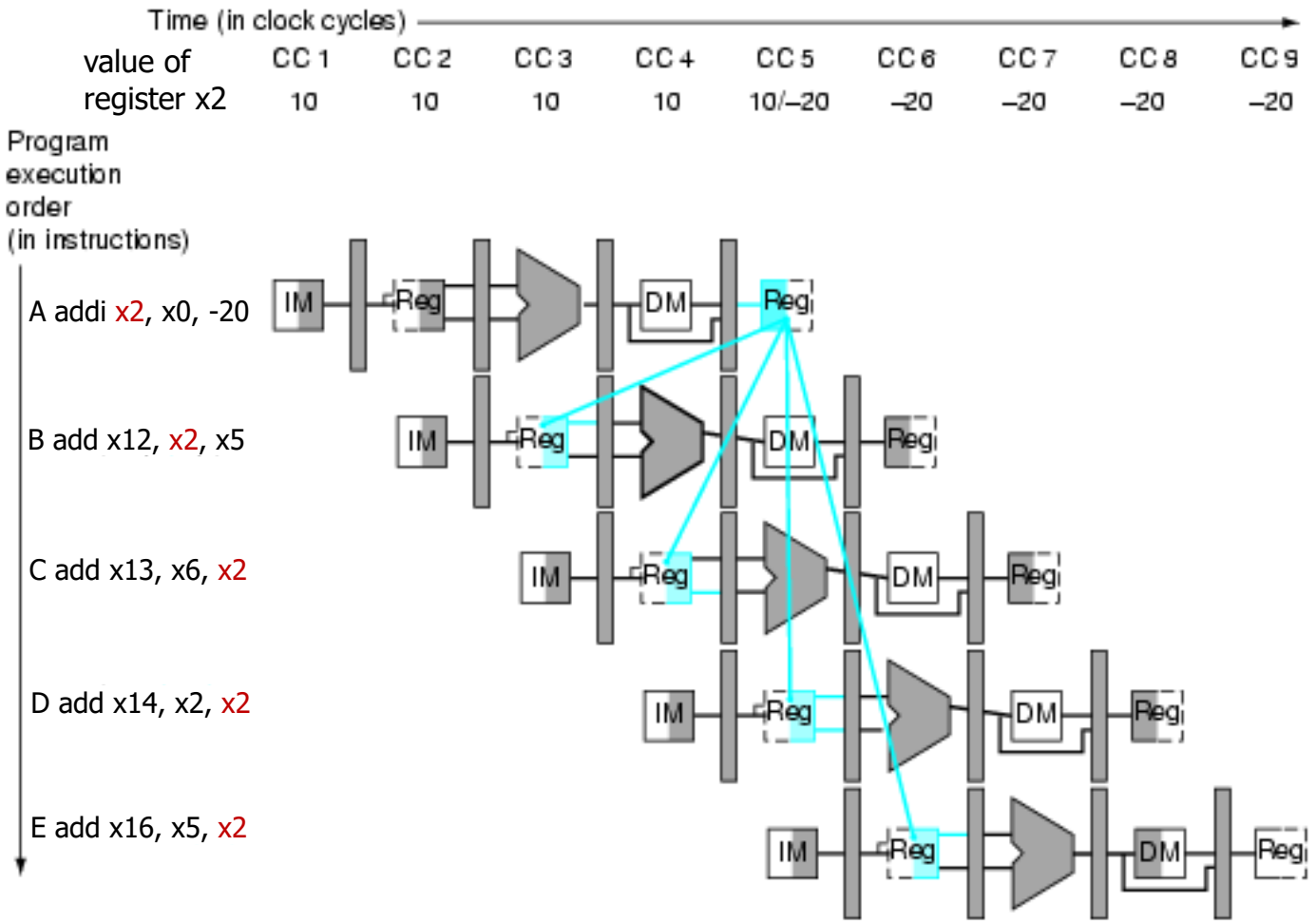
```
cm_ram[0]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  00:   add  x0, x0, x0  // NOP
cm_ram[1]={12'd3,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  04:   addi x5, x0, 3   // x5 = 3
cm_ram[2]={12'd4,      5'd0, 3'b000, 5'd5, 7'b0010011}; //  08:   addi x5, x0, 4   // x5 = 4
cm_ram[3]={7'd0, 5'd5, 5'd5, 3'b000, 5'd30,7'b0110011}; //  0c:   add  x30,x5, X5  // led = x5 + x5
cm_ram[4]={7'd0, 5'd0, 5'd0, 3'b000, 5'd0, 7'b0110011}; //  10:   add  x0, x0, x0  // NOP
```

CC6

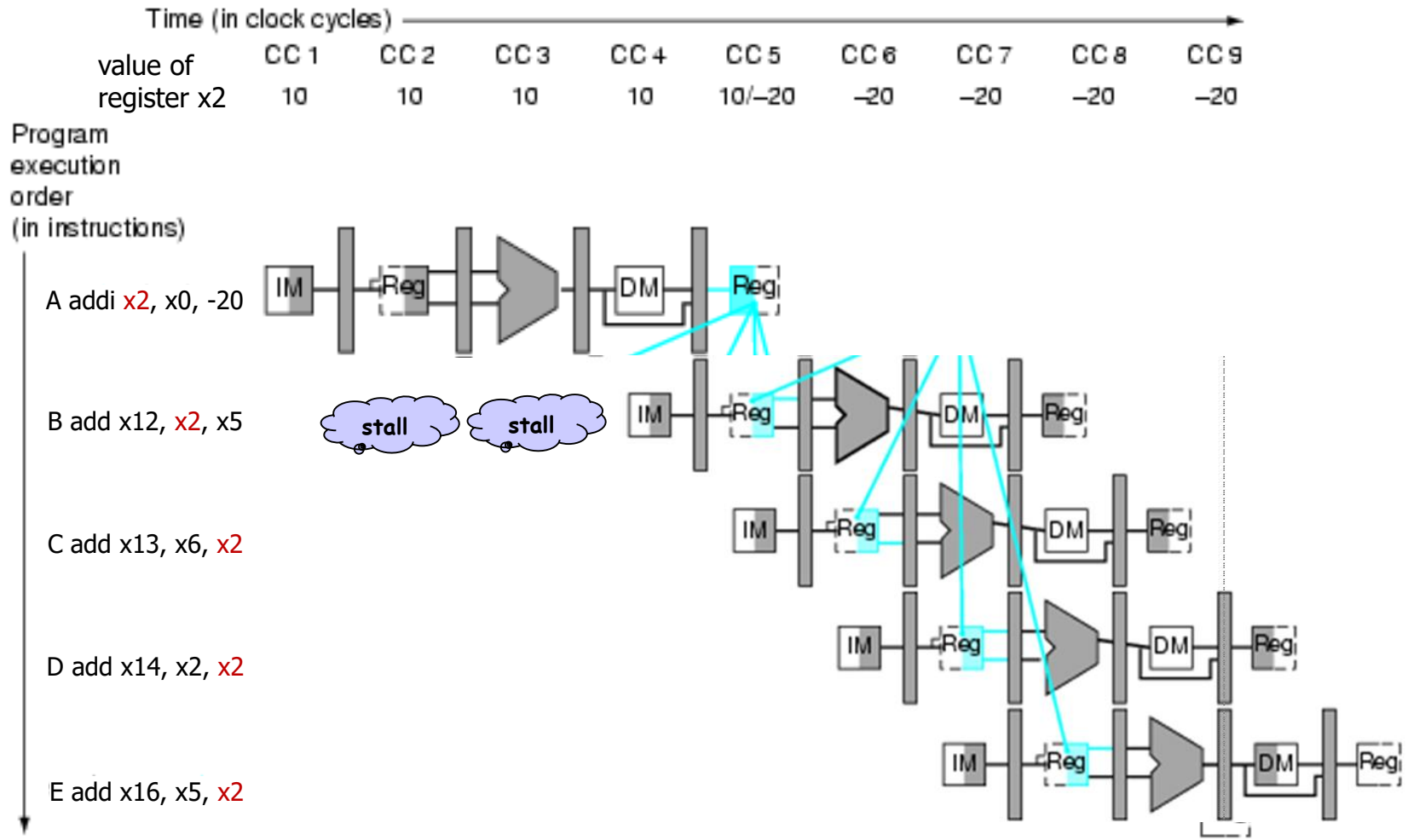


データ・ハザードとパイプラインチャート

- 命令Aが生成した x2 を後続の命令が利用する場合に，データの受け渡しの制約が生じる．



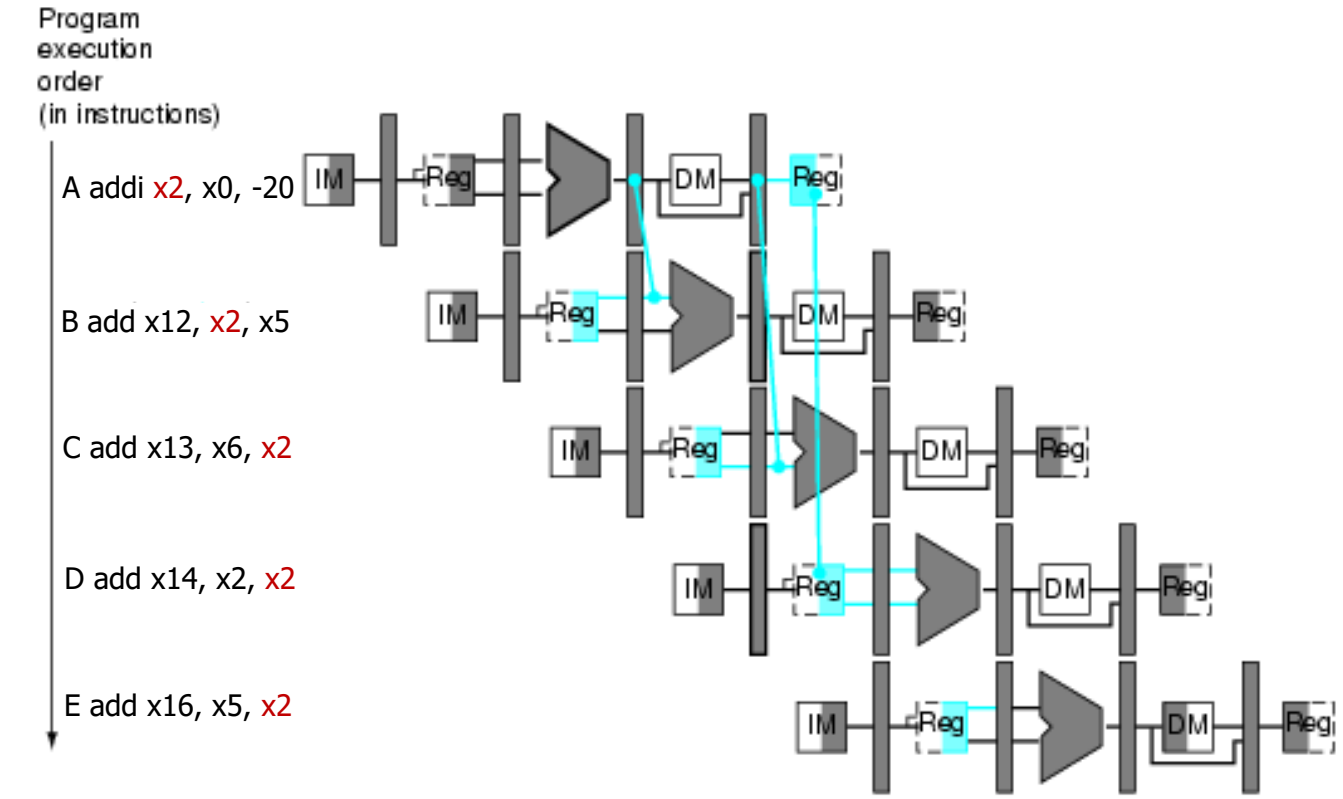
Data Hazard and Stall



フォワーディングによるデータハザードの回避

Time (in clock cycles)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
value of register x2	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X



フォワーディングによるデータハザードの回避

Time (in clock cycles)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
value of register x2	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X

Program execution order (in instructions)

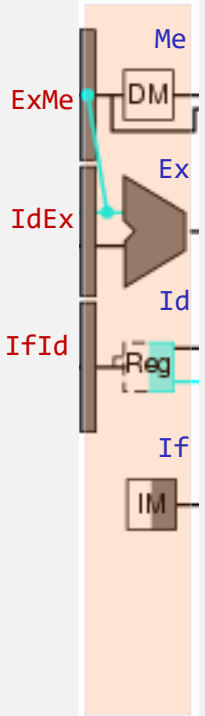
A addi x2, x0, -20

B add x12, x2, x5

C add x13, x6, x2

D add x14, x2, x2

E add x16, x5, x2



フォワーディングによるデータハザードの回避

Time (in clock cycles)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
value of register x2	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X

Program execution order (in instructions)

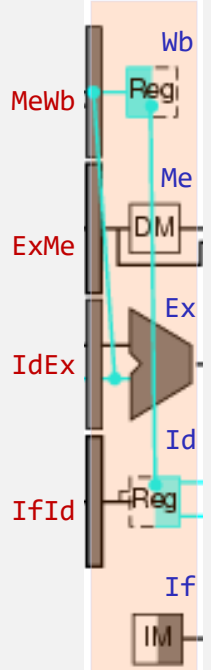
A addi x2, x0, -20

B add x12, x2, x5

C add x13, x6, x2

D add x14, x2, x2

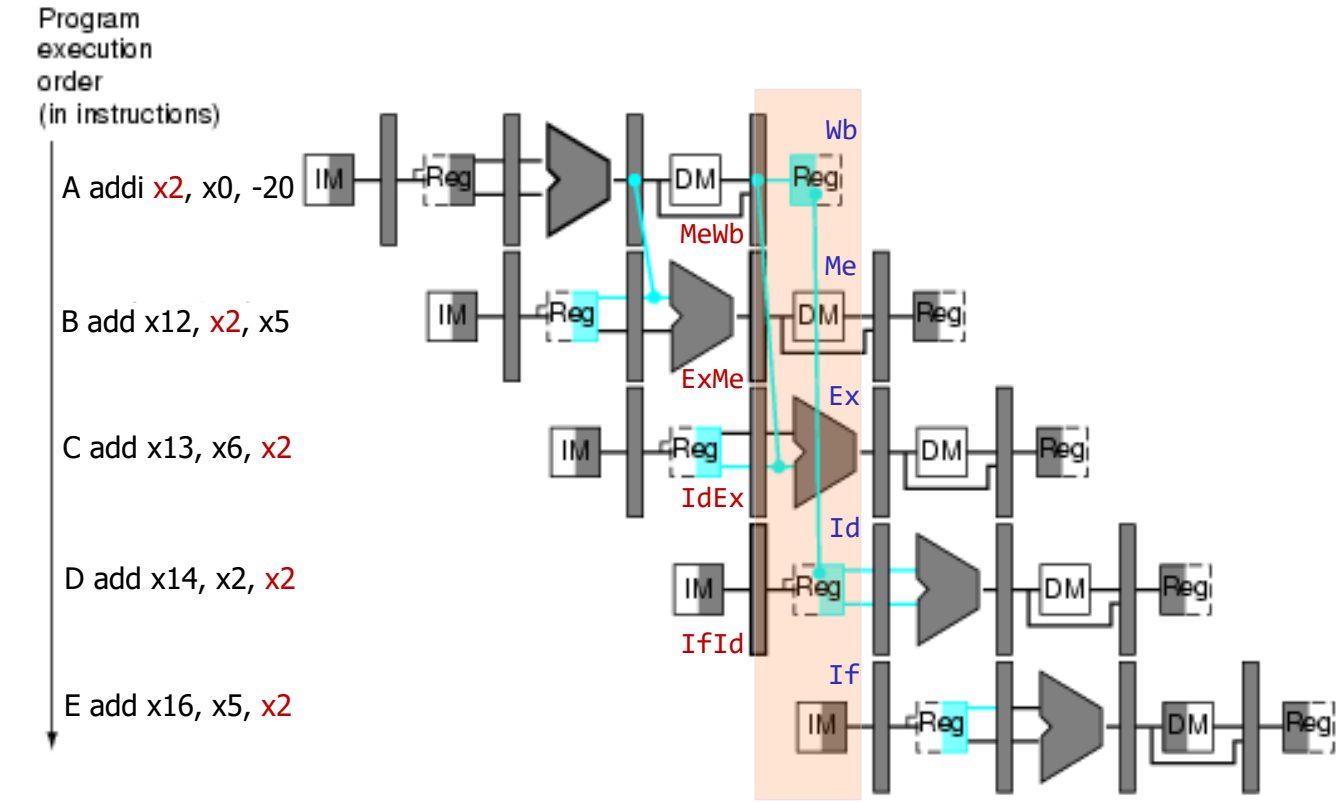
E add x16, x5, x2



フォワーディングによるデータハザードの回避

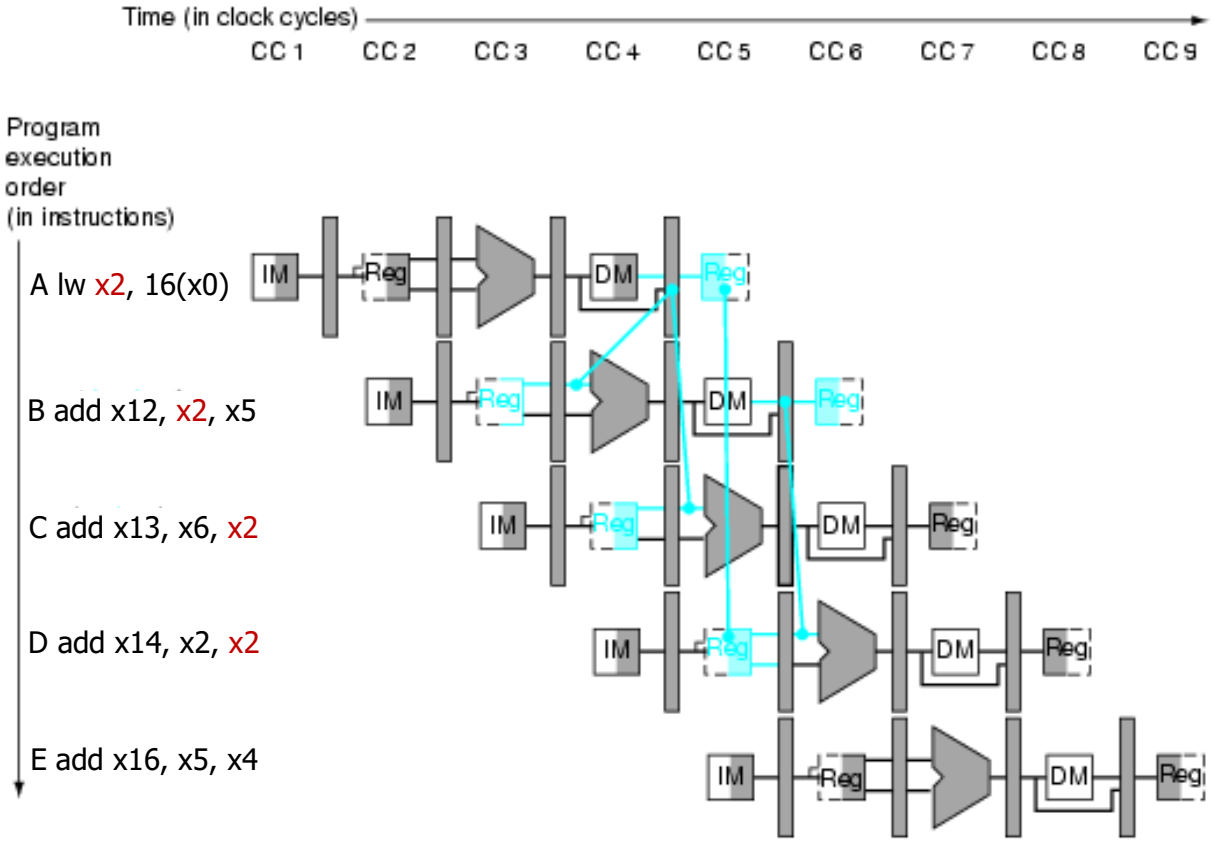
Time (in clock cycles)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
value of register x2	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X



フォワーディングで解決できないデータハザード

- ロード命令がロードしたデータを次の命令が使う場合には，フォワーディングでもデータを供給できない．
- このような命令列が生成されないように，ソフトウェア（コンパイラ，アセンブラ）が工夫する．または，このような命令列が実行される時に，ハードウェアをストールさせる．
- 今回のコンテストでは，lw命令でロードした値を次の命令が使うことは無いので、このストールを実装する必要はない．



RISC-V Program for the contest

- Result : 0x017fd000

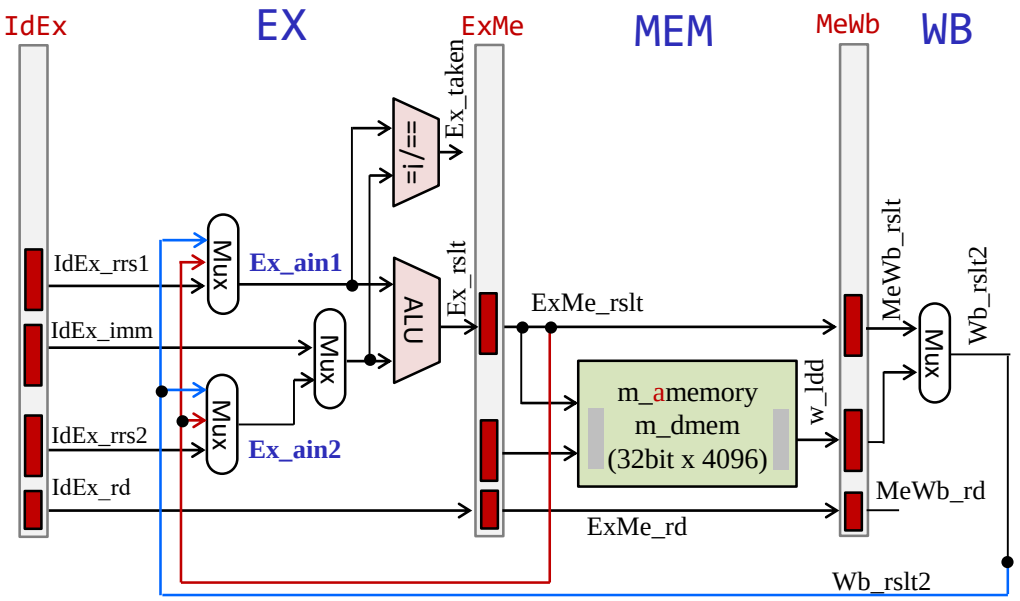
```c
#include <stdio.h>

main()
{
    int mem[2048];
    int i=0, j=0, sum = 0;

    for(j=0; j<3; j++){
        for(i=0; i<2048; i++) {
            mem[i] = i*4;
        }
        for(i=0; i<2048; i++) {
            sum += mem[i];
        }
    }
    printf("%d %x¥n", sum, sum);
}
```

/home/tu_kise/cld/2023/baseline/program.txt

```
/*************************************************************/
/** program for CLD design contest 2023 (Version 2023-05-14a) **/
/** do not modify this code                                  **/
/*************************************************************/
initial begin
cm_ram[ 0] = 32'h00000033; //     add  x0,  x0,  x0
cm_ram[ 1] = 32'h00000033; //     add  x0,  x0,  x0
cm_ram[ 2] = 32'h00000a13; //     addi x20, x0,  0
cm_ram[ 3] = 32'h00300a93; //     addi x21, x0,  3
cm_ram[ 4] = 32'h00000633; //     add  x12, x0,  x0
cm_ram[ 5] = 32'h00000593; // L03:addi x11, x0,  0
cm_ram[ 6] = 32'h40000413; //     addi x8,  x0,  1024
cm_ram[ 7] = 32'h40040413; //     addi x8,  x8,  1024
cm_ram[ 8] = 32'h00000493; //     addi x9,  x0,  0
cm_ram[ 9] = 32'h00000513; //     addi x10, x0,  0
cm_ram[10] = 32'h00b52023; // L01:sw   x11, 0(x10)
cm_ram[11] = 32'h00148493; //     addi x9,  x9,  1
cm_ram[12] = 32'h00458593; //     addi x11, x11, 4
cm_ram[13] = 32'h00959cb3; //     sll  x25, x11, x9
cm_ram[14] = 32'h009cdd33; //     srl  x26, x25, x9
cm_ram[15] = 32'h00058593; //     addi x11, x11, 0
cm_ram[16] = 32'h00450513; //     addi x10, x10, 4
cm_ram[17] = 32'h00940463; //     beq  x8,  x9,  L04
cm_ram[18] = 32'hfe0410e3; //     bne  x8,  x0,  L01
cm_ram[19] = 32'h40000413; // L04:addi x8,  x0,  1024
cm_ram[20] = 32'h40040413; //     addi x8,  x8,  1024
cm_ram[21] = 32'h00000493; //     addi x9,  x0,  0
cm_ram[22] = 32'h00000513; //     addi x10, x0,  0
cm_ram[23] = 32'h00052583; // L02:lw   x11, 0(x10)
cm_ram[24] = 32'h00148493; //     addi x9,  x9,  1
cm_ram[25] = 32'h00450513; //     addi x10, x10, 4
cm_ram[26] = 32'h00b60633; //     add  x12, x12, x11
cm_ram[27] = 32'h00160613; //     addi x12, x12, 1
cm_ram[28] = 32'hfff60613; //     addi x12, x12, -1
cm_ram[29] = 32'h00160613; //     addi x12, x12, 1
cm_ram[30] = 32'h00160613; //     addi x12, x12, 1
cm_ram[31] = 32'hfff60613; //     addi x12, x12, -1
cm_ram[32] = 32'h00160613; //     addi x12, x12, 1
cm_ram[33] = 32'hffe60613; //     addi x12, x12, -2
cm_ram[34] = 32'hfc941ae3; //     bne  x8,  x9,  L02
cm_ram[35] = 32'h015d5d33; //     srl  x26, x26, x21
cm_ram[36] = 32'h001a0a13; //     addi x20, x20, 1
cm_ram[37] = 32'h01140413; //     addi x8,  x8,  0x11
cm_ram[38] = 32'h01240413; //     addi x8,  x8,  0x12
cm_ram[39] = 32'h01340413; //     addi x8,  x8,  0x13
cm_ram[40] = 32'h01440413; //     addi x8,  x8,  0x14
cm_ram[41] = 32'hf75a18e3; //     bne  x20, x21, L03
cm_ram[42] = 32'h00000033; //     add  x0,  x0,  x0
cm_ram[43] = 32'h00060f33; //     add  x30, x12, x0
cm_ram[44] = 32'h000f0033; //     add  x0,  x30, x0
cm_ram[45] = 32'h00000033; //     add  x0,  x0,  x0
cm_ram[46] = 32'h00000033; //     add  x0,  x0,  x0
cm_ram[47] = 32'h00000033; //     add  x0,  x0,  x0
cm_ram[48] = 32'h00000033; //     add  x0,  x0,  x0
cm_ram[49] = 32'h00000033; //     add  x0,  x0,  x0
end
```

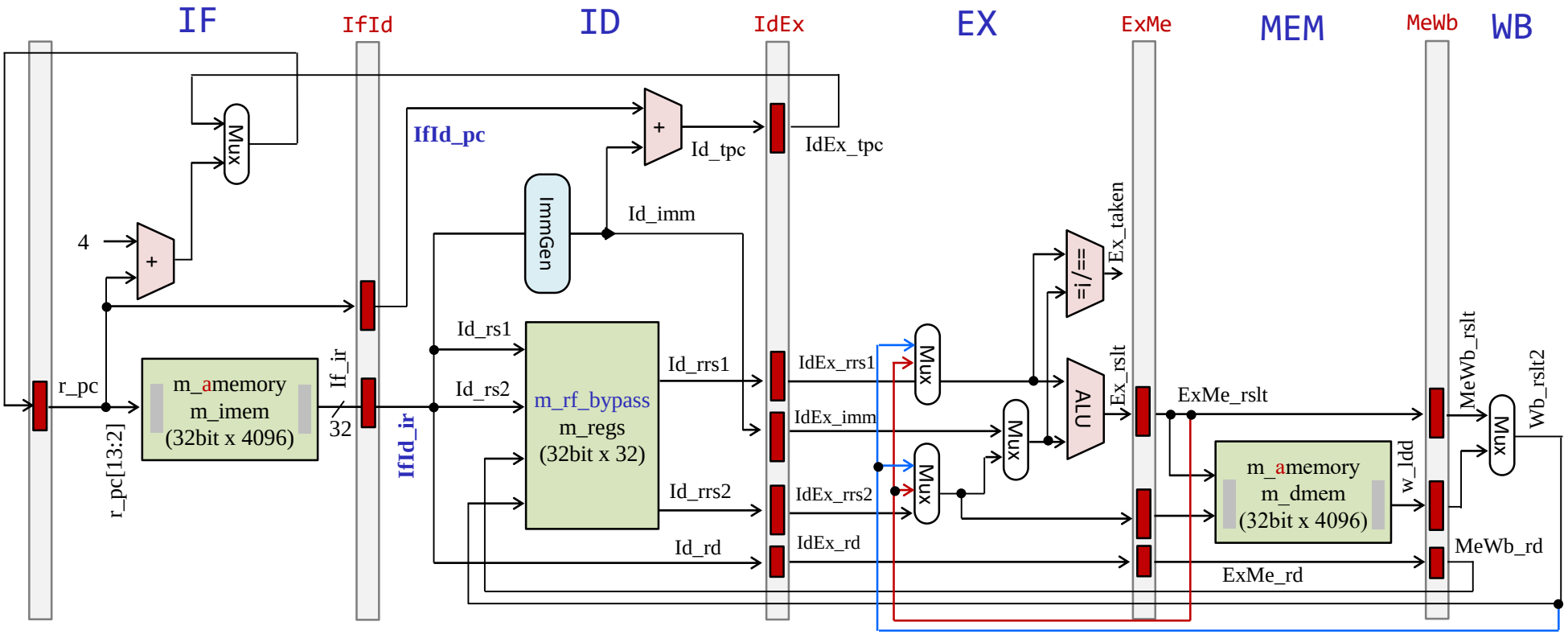
フォワーディングのための変更点

```
wire [31:0] Ex_ain1 = (Ex_rs1==0) ? 0 : (Ex_rs1==Me_rd & Me_we) ? ExMe_rslt : (Ex_rs1==Wb_rd & Wb_we) ? Wb_rslt2 : IdEx_rrs1;

wire [31:0] Ex_ain2 = (Ex_rs2==0) ? 0 : (Ex_rs2==Me_rd & Me_we) ? ExMe_rslt : (Ex_rs2==Wb_rd & Wb_we) ? Wb_rslt2 : IdEx_rrs2;
```



m_proc20 5段のパイプライン版

- add, addi, sll, srl, lw, sw, beq, bne命令に対応したプロセッサ（データフォワーディング有り）
- IF, ID, EX, MEM, WB のそれぞれをステージとする5段のパイプラインプロセッサ
- ステージ IF と ID の間のパイプラインレジスタには IfId_ から始まる名前を利用する.
- ステージ ID で生成される配線には Id_ から始まる名前を利用する.



code190.v（code185.v を参考に自分で実装する）

Department of Computer Science
Course number: CSC.T341

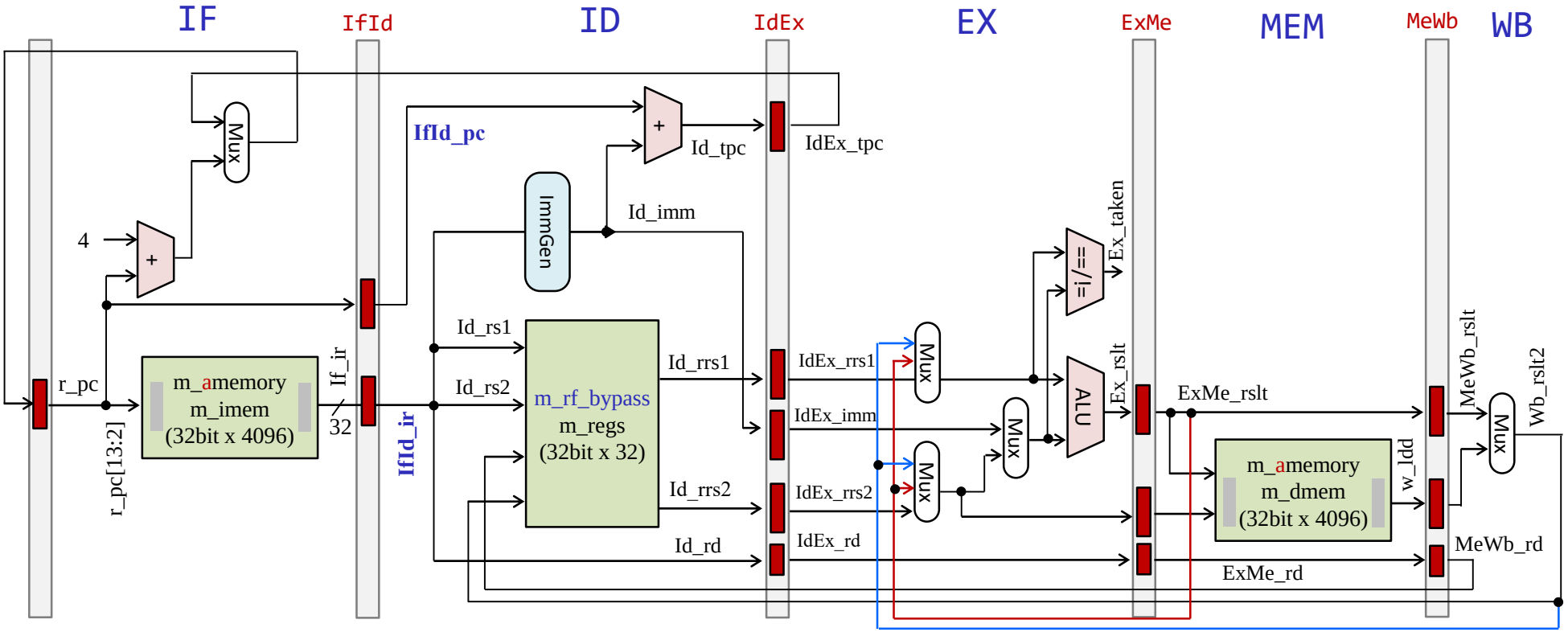
コンピュータ論理設計
Computer Logic Design

13. Supplemental explanation,
and Preparing for the design contest (group work)

吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
講義：月曜日 10:45-12:25, 木曜日 10:45-12:25

m_proc20 5段のパイプライン版

- add, addi, sll, srl, lw, sw, beq, bne命令に対応したプロセッサ（データフォワーディング有り）
- IF, ID, EX, MEM, WB のそれぞれをステージとする5段のパイプラインプロセッサ
- ステージ IF と ID の間のパイプラインレジスタには IfId_ から始まる名前を利用する.
- ステージ ID で生成される配線には Id_ から始まる名前を利用する.



code190.v（code185.v を参考に自分で実装する）

Recommended Reading

PAPER *Special Section on Parallel, Distributed, and Reconfigurable Computing, and Networking*

RVCoreP: An Optimized RISC-V Soft Processor of Five-Stage Pipelining

Hiromu MIYAZAKI[†a)], ***Student Member***, **Takuto KANAMORI**[†b)], **Md Ashraful ISLAM**[†c)], ***Nonmembers***, ***and*** **Kenji KISE**[†d)], ***Member***

SUMMARY RISC-V is a RISC based open and loyalty free instruction set architecture which has been developed since 2010, and can be used for cost-effective soft processors on FPGAs. The basic 32-bit integer instruction set in RISC-V is defined as RV32I, which is sufficient to support the operating system environment and suits for embedded systems. In this paper, we propose an optimized RV32I soft processor named RVCoreP adopting five-stage pipelining. Three effective methods are applied to the processor to improve the operating frequency. These methods are instruction fetch unit optimization, ALU optimization, and data memory optimization. We implement RVCoreP in Verilog HDL and verify the behavior using Verilog simulation and an actual Xilinx Atrix-7 FPGA board. We evaluate IPC (instructions per cycle), operating frequency, hardware resource utilization, and processor performance. From the evaluation results, we show that RVCoreP achieves 30.0% performance improvement compared with VexRiscv, which is a high-performance and open source RV32I processor selected from some related works.

key words: *soft processor, FPGA, RISC-V, RV32I, Verilog HDL, five-stage pipelining*

1. Introduction

version of a general-purpose instruction set.

Among these instruction sets, we focus on RV32I in this paper because it is sufficient to support the operating system environment and suits for embedded systems. RV32I can emulate other extensions of M, F, and D, and can be configured with fewer hardware resources than processors supporting RV32G. Although several soft processors that support RV32I have been released [5], they are not highly optimized for FPGAs.

In this paper, we propose an optimized RV32I soft processor named **RVCoreP** of five-stage pipelining which is highly optimized for FPGAs. The main contributions of this paper are as follows.

- We propose an optimized RV32I soft processor of five-stage pipelining highly optimized for FPGAs. To improve the operating frequency, three optimization methods are applied to the processor. They are instruction fetch unit optimization, ALU optimization, and

RV-PC: RISC-Vパーソナルコンピュータ $500

- Stand-alone RISC-V computer
  - Digilent Nexys A7 FPGA Board
  - Pmod ps2 keyboard / PS2 keyboard
  - USB wireless mouse
  - Mobile battery
  - VGA display
  - Some cables (microUSB, VGA)
  - Two acrylic plates, spacers
  - Option, ethernet connection

https://youtu.be/Kt_iXVAjXcQ

Syllabus (1/3)

講義の概要とねらい

本講義では、「論理回路理論」の講義で習得した知識をベースに、より実用的なディジタル回路について学ぶ。また、簡単なコンピュータを例題として、コンピュータの基本原理とその論理設計の方法を学習する。
演習では、学んだ組合せ回路と順序回路をVerilog HDL等のハードウェア記述言語で記述し、シミュレーションによる回路の動作検証、FPGAが搭載されたハードウェアボード等に実装して動作確認をおこなう。

到達目標

本講義を履修することによって以下を習得する。
・コンピュータシステムの基本構成
・シングルサイクルプロセッサの論理設計に関する知識
・パイプライン処理をおこなうプロセッサの論理設計に関する知識
・ハードウェア記述言語を用いたシンプルなコンピュータシステムの設計能力

キーワード

コンピュータ，命令セットアーキテクチャ，プロセッサ、パイプライン処理，ハードウェア記述言語，Verilog HDL、FPGA

学生が身につける力

国際的教養力	コミュニケーション力	専門力	課題設定力	実践力または解決力
-	-	✓	-	✓

授業の進め方

原則として、90分×2コマの講義の後、90分×1コマのFPGAボードを用いた演習をおこないます。

Syllabus (2/3)

授業計画・課題		
	授業計画	課題
第1回	コンピュータシステムの基本構成	コンピュータシステムの基本構成について理解する。
第2回	論理設計演習(1)	論理設計演習(1)
第3回	ハードウェア記述言語：組合せ回路	組合せ回路の記述を理解する。
第4回	ハードウェア記述言語：順序回路	順序回路の記述を理解する。
第5回	論理設計演習(2)	論理設計演習(2)
第6回	ハードウェア記述言語：よく使われる回路	よく使われる回路の記述を理解する。
第7回	リコンフィギャラブルシステム	リコンフィギャラブルシステムとFPGAボードについて理解する。
第8回	論理設計演習(3)	論理設計演習(3)
第9回	命令セットアーキテクチャ：データ表現とアドレス指定形式	ISAにおけるデータ表現とアドレス指定形式について理解する。
第10回	命令セットアーキテクチャ：算術論理演算命令	ISAにおける算術論理演算命令について理解する。
第11回	論理設計演習(4)	論理設計演習(4)
第12回	命令セットアーキテクチャ：ロードストア命令と分岐命令	ISAにおけるロードストア命令と分岐命令について理解する。
第13回	プロセッサの基本構成要素：算術論理演算ユニット	算術論理演算ユニットについて理解する。
第14回	論理設計演習(5)	論理設計演習(5)
第15回	プロセッサの基本構成要素：レジスタファイルとメモリ	レジスタファイルとメモリについて理解する。
第16回	シングルサイクルプロセッサのデータパス	シングルサイクルプロセッサのデータパスについて理解する。
第17回	論理設計演習(6)	論理設計演習(6)
第18回	シングルサイクルプロセッサの制御	シングルサイクルプロセッサの制御について理解する。
第19回	パイプライン処理	パイプライン処理について理解する。
第20回	論理設計演習(7)	論理設計演習(7)
第21回	パイプラインハザードとデータフォワーディング	パイプラインハザードとデータフォワーディングについて理解する。
第22回	論理設計演習(8)	論理設計演習(8)

Syllabus (3/3)

教科書

デイビッド・A. パターソン、ジョン・L. ヘネシー (著)、成田光彰 (翻訳)『コンピュータの構成と設計 第5版 上/下』日経BP社

参考書、講義資料等

無し。

成績評価の基準及び方法

講義で扱うコンピュータ論理設計に関する理解、ハードウェア記述言語を用いたコンピュータシステム実装への応用力を評価する。演習（30%）と期末

演習 30%, 設計コンテスト 20%, 期末試験 50%

関連する科目

CSC.T252 ： 論理回路理論
CSC.T262 ： アセンブリ言語
CSC.T372 ： コンパイラ構成
CSC.T363 ： コンピュータアーキテクチャ
CSC.T433 ： 先端コンピュータアーキテクチャ

履修の条件(知識・技能・履修済科目等)

履修条件は特に設けないが、関連する科目の論理回路理論を履修していることが望ましい。

連絡先（メール、電話番号）　※"[at]"を"@"(半角)に変換してください。

吉瀬謙二: kise[at]c.titech.ac.jp

オフィスアワー

メールで事前予約すること。

References

- Computer Logic Design support page
  - https://www.arch.cs.titech.ac.jp/lecture/CLD/
- ACRi Room
  - https://gw.acri.c.titech.ac.jp
- ACRi Blog
  - https://www.acri.c.titech.ac.jp/wordpress/
- 情報工学系計算機室
  - http://www.csc.titech.ac.jp/
- Xilinx Vivado Design Suite
  - https://japan.xilinx.com/products/design-tools/vivado.html
- Digilent Arty A7-35T
  - https://reference.digilentinc.com/reference/programmable-logic/arty-a7/start
- Verilog HDL
  - https://ja.wikipedia.org/wiki/Verilog