2023年度（令和5年）版

Ver. 2023-11-07d

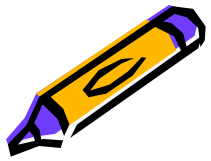
Course number: CSC.T363

コンピュータアーキテクチャ 演習(4)
Computer Architecture Exercise(4)

情報工学系　吉瀬謙二, Berjab Nesrine

Kenji KISE, Department of Computer Science

kise_at_c.titech.ac.jp



コンピュータアーキテクチャ 演習(Exercise)の注意点

- 連絡手段は Slack を利用します.
- 演習は 15:25～17:05 です.
  - 20分以上(15:45までに)遅刻したら欠席扱いになります.
  - 前半は課題の説明(15分程度)で, 後半は課題の解決とチェックポイントの確認.

- 演習は手元の FPGA ボードと ACRi ルームを利用します.
- 3~4人のグループを作成します. そのグループ内で情報を共有しながら演習を進めください.
- 問題はグループ内で相談して解決する, あるいは, 担当 TA や教員に質問してください.
- 演習には出席点があります. 休まずにきちんと出席しましょう.
- 演習スライドにチェックポイントの図がある場所は, 作業を確認してもらう場所です. すべてのチェックポイントをクリアしましょう.

【重要】ACRiルームのサーバの予約

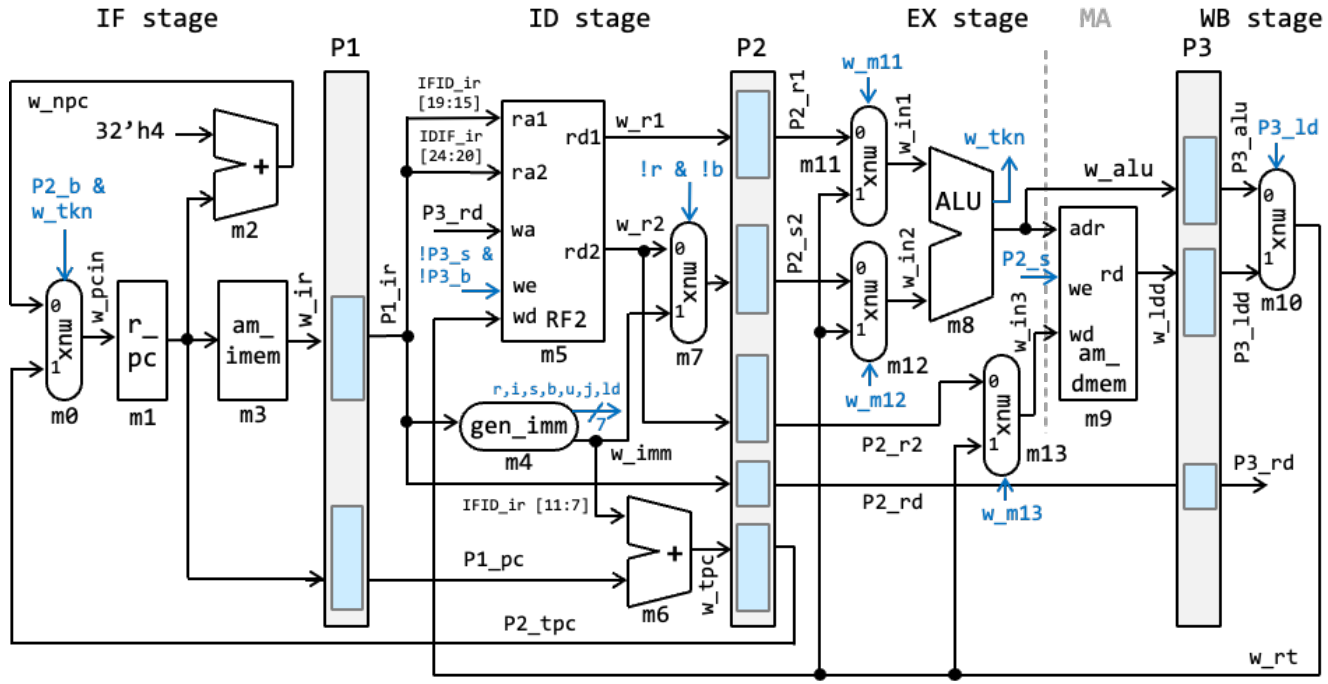
- ACRiルームのアカウントを使って，次のURLからログインする．
  - https://gw.acri.c.titech.ac.jp/wp
- 「予約ページトップ」から，vsで始まるサーバで演習の日の15:00～18:00の枠を予約すること．

Exercise(4)

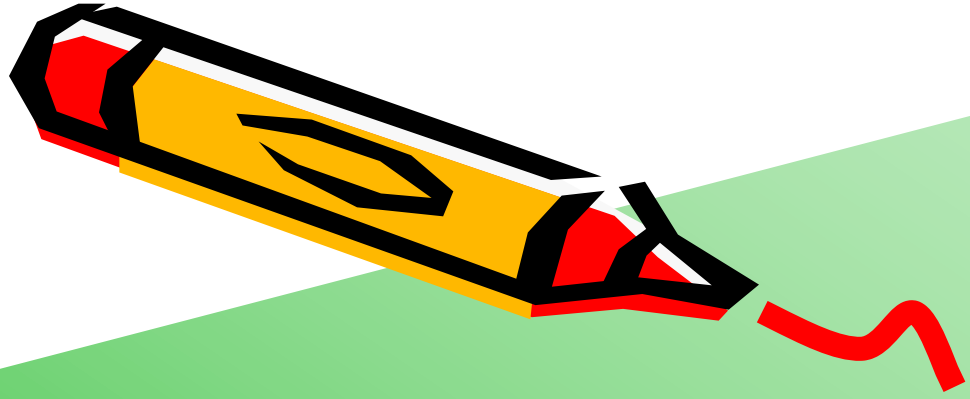
- Project_23
  - DRAMを利用して，DRAMのアクセスに必要となるサイクル数を見積もる.
    - MIGの環境でロード命令に必要となるサイクル数の最小値と最大値を求める
- Project_24
  - 命令キャッシュを実装してシミュレーションにより正しくキャッシュが実装されていることを示す。

Exercise(4) にも proc8s プロセッサを使用

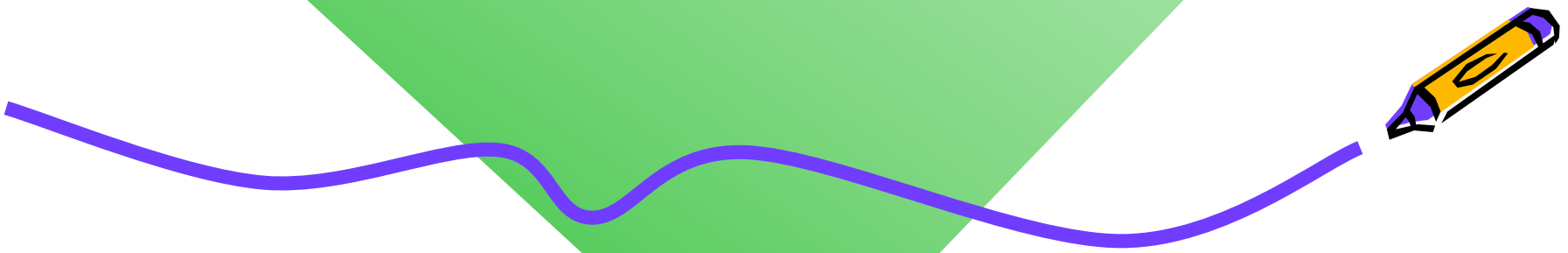
- 4段のパイプライン処理のプロセッサ



31–25	24–20	19–15	14–12	11–7	6–0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12] imm[10:5]	rs2	rs1	funct3	imm[4:1] imm[11]	opcode	B-type

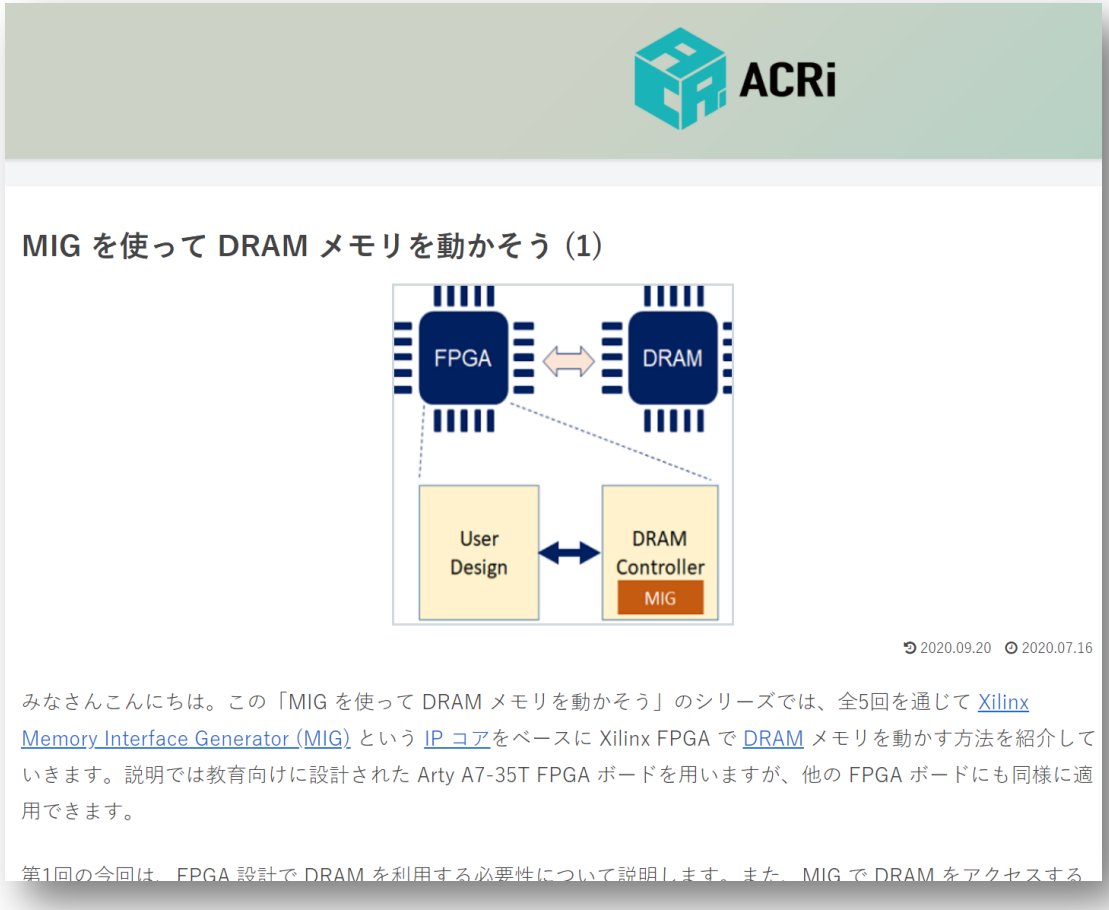


Project_23



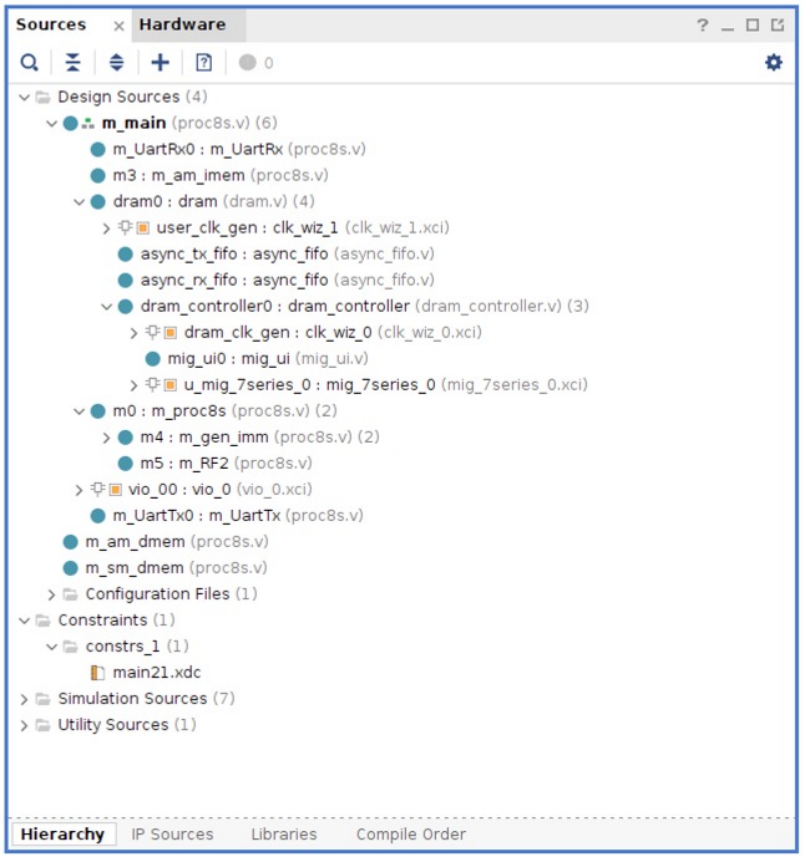
読んでおきたい記事

- MIG を使って DRAM メモリを動かそう (1)→ (5)
  - https://www.acri.c.titech.ac.jp/wordpress/archives/6048



DRAMを用いたプロセッサ

- プロセッサ設計コンテストのベースとなるプロジェクト
  - ハードウェアのモジュール構成（左）とソースコードとIP（右）



プロジェクトのコピー，論理合成，コンフィギュレーション (1/2)

```
$ cd ~/ca
$ cp -r /home/u_nesrine/ca/2023/src/project_23 ./
```

- 上のコマンドで，project_23 をコピーする．
- プロジェクトディレクトリの構成：

•constrs/
  - 制約ファイルが入っている．

•prog/
  - 1～1000 の合計値 500500 を求めるプログラムのアセンブリコード (program0.s) と program0.{dump,elf,bin,hex} ファイルが入っている．

  → UART で送信するのは mem_img.bin

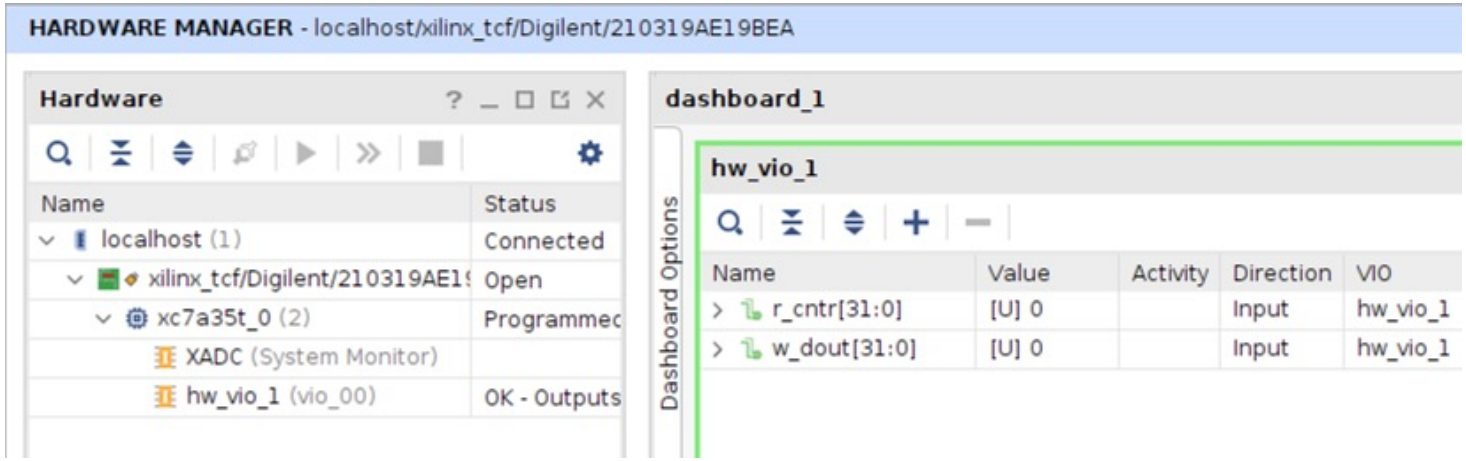
•src/
  - プロセッサ proc8s と DRAM コントローラのソースコード．

•vivado/
  - Vivado の Open Project で dram_proc.xpr ファイルを開くと Vivado のプロジェクトが開ける．

•Makefile
  - make と打つと prog/program0.hex ファイルを読み込んでシミュレーションが実行される．

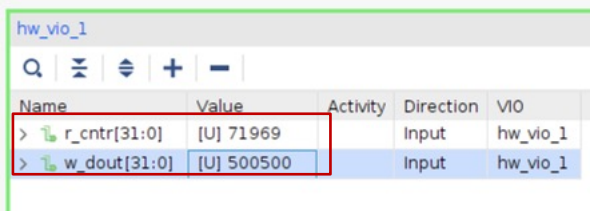
プロジェクトのコピー，論理合成，コンフィギュレーション (2/2)

- Vivado で，~/ca/src/project_23/vivado/dram_proc.xpr を開く．
- 論理合成，配置・配線，ビットストリームファイルを作成する．
- 生成したビットストリームファイルで FPGA をコンフィギュレーションする．
  - ~/ca/2023/src/project_23/vivado/dram_proc.runs/impl_1/m_main.bit



RISC-V プログラムをシリアル通信で送信して実行

- gtkterm を起動して 1Mbaud に設定する.
- UART で送信するのは ~/ca/2023/src/project_23/prog/program0.bin
- VIOの r_cntr は実行サイクル数、w_dout はプロセッサの出力(x30 レジスタ)である.



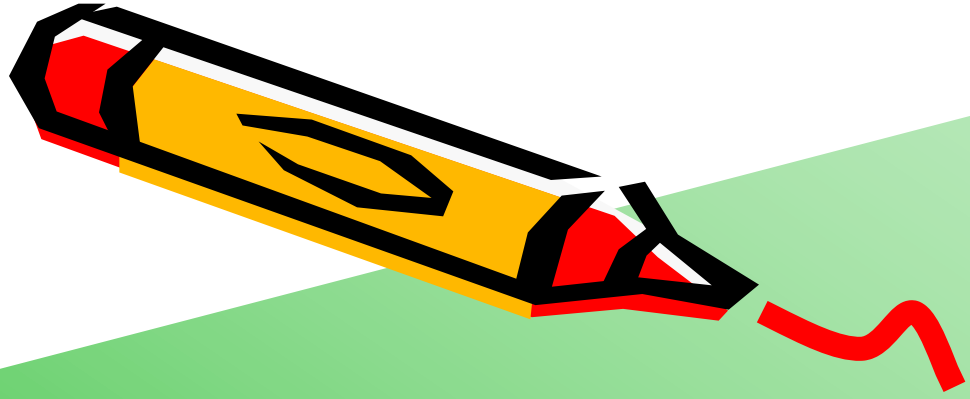
r_cntr は unsigned decimal で表示.

program.s

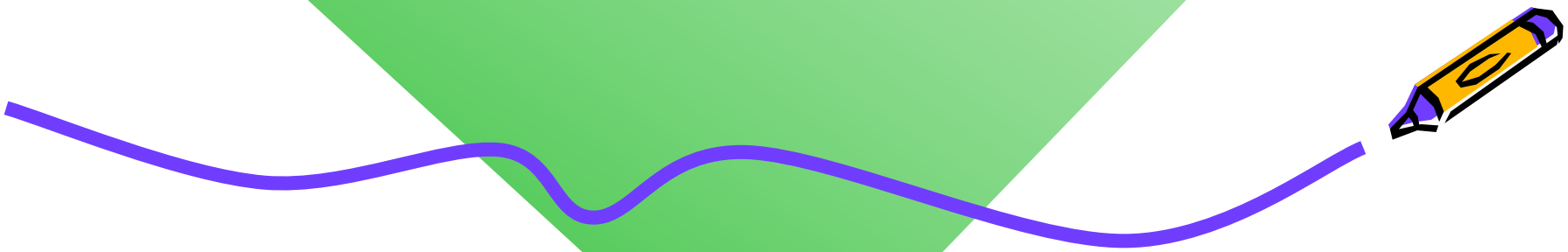
```
.section .text
    .globl _main
_main:
    addi s0, x0, 1000   #     max = 1000
    addi a0, x0, 0      #     sum = 0
    addi s1, x0, 0      #     i   = 0

1:  addi s1, s1, 1      # 1:  i   = i + 1
    add  s2, s1, s1     #     adr = i + i
    add  s2, s2, s2     #     adr = adr + adr
    sw   s1, 0(s2)      #     mem[adr] = i
    bne  s1, s0, 1b     #     if (i!=max) goto 1b

    addi s1, x0, 0      #     i   = 0
1:  addi s1, s1, 1      # 1:  i   = i + 1
    add  s2, s1, s1     #     adr = i + i
    add  s2, s2, s2     #     adr = adr + adr
    lw   s3, 0(s2)      #     d   = mem[adr]
    add  a0, a0, s3     #     sum = sum + d
    bne  s1, s0, 1b     #     if (i!=max) goto 1b

_exit:
    add  x30, a0, 0     #     print(sum)
    addi s0, x0, 1
1:  bne  s0, x0, 1b
```

DRAMからのリードのレイテンシを求める

- ロード命令に必要となるサイクル数の**最小値**と**最大値**を求める.
  - **program0.s を変更**し，様々なメモリアドレスに対するロード命令を追加する.
  - **proc8s.v を変更**し，それぞれのロード命令の実行に必要となったサイクル数を求める.
    - このサイクル数の測定には 160MHz のクロックを用いている点に注意する.
  - そのために用いた RISC-V アセンブリ言語のプログラムと **proc8s.v** の変更箇所を示すこと.

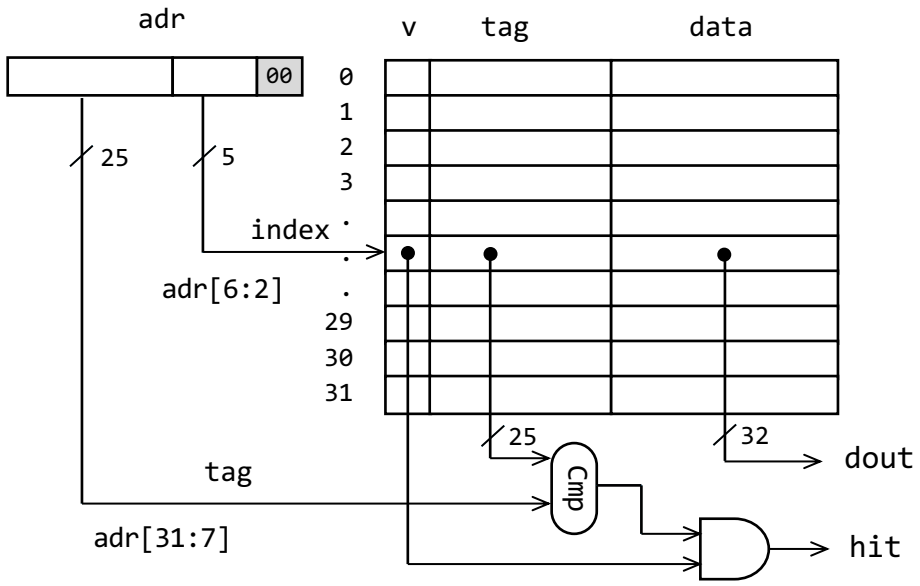


Project_24



実装する命令キャッシュの構成

- 命令キャッシュを実装して，シミュレーションで動作を確認する.
- ダイレクトマップ方式32エントリ のキャッシュを実装する.



RISC-V Program: 1～100の合計値

- 1～100 までの合計値 (5050) を出力するプログラム.

Program_24.txt

```
`MM[0]={12'd0,5'd0,3'h0,5'd10,7'h13}; // 00 addi x10,x0,0
`MM[1]={12'd0,5'd0,3'h0,5'd3,7'h13}; // 04 addi x3,x0,0
`MM[2]={12'd101,5'd0,3'h0,5'd1,7'h13}; // 08 addi x1,x0,101
`MM[3]={7'd0,5'd3,5'd10,3'h0,5'd10,7'h33}; // 0c L:add x10,x10,x3
`MM[4]={12'd1,5'd3,3'h0,5'd3,7'h13}; // 10 addi x3,x3,1
`MM[5]={~12'd0,5'd1,5'd3,3'h1,5'b11001,7'h63};// 14 bne x3,x1,L
`MM[6]=32'h00050f13; // 18 HALT
```

Program_24.c

```
#include <stdio.h>
  int main() {
    int sum = 0;
    for (int i = 1; i <= 100; i++) {
         sum += i;
    }

     printf("0000_%x\n", sum)

     return 0;
}
```

Result: 0000_13BA
(10進数で 5050)

遅いデータメモリとサンプルプログラム (1/2)

- 同期メモリに変更しながら、メモリの読み出しの要求を受けてから、一定のクロックサイクルが経過してからデータを出力するように修正する。
  - 遅いデータメモリのモジュール m_imem を利用するように修正している.

proc8s_c.v の一部

```
`define D_DELAY 5
module m_imem(w_clk, w_pc, w_re, r_insn, r_oe);
  input wire w_clk, w_re;
  input wire [31:0] w_pc;
  output reg [31:0] r_insn = 0;
  output reg r_oe = 0;
  reg [31:0] mem [0:2047];
  reg [31:0] r_c=1, r_pc=0;
  always@(posedge w_clk) begin
    r_pc <= (r_c==1 & w_re) ? w_pc : r_pc;
    r_c <= (r_c==1 & w_re) ? 2 : (r_c==1 | r_c==`D_DELAY) ? 1 : r_c+1;
    r_oe <= (r_c==`D_DELAY-1);
    r_insn <= (r_c==`D_DELAY-1) ? mem[r_pc[12:2]] : 0;
  end
  integer i; initial for (i=0; i<2048; i=i+1) mem[i] = 32'd0;
endmodule
```

- 1行目で、メモリを参照するための遅延のD_DELAYを定義する.
  - ここは、5サイクルを指定するが、この値は変更して使うことがある。

遅いデータメモリとサンプルプログラム (2/2)

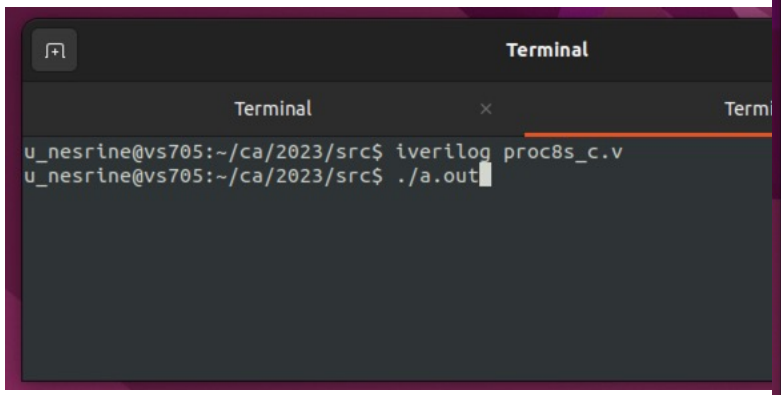
proc8s_c.v の一部

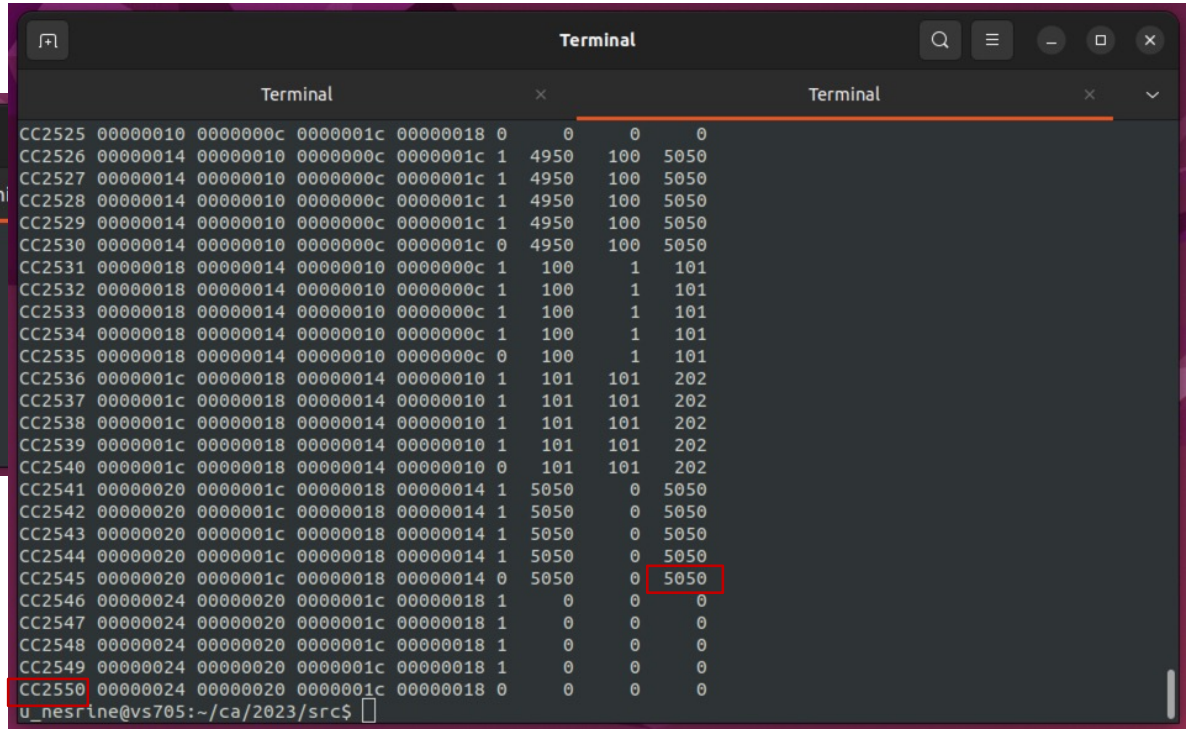
```
module m_sim(w_clk, w_cc);
  input wire w_clk; input wire [31:0] w_cc;
  wire w_oe;
  wire w_stall = !w_oe;
  wire [31:0] w_pc, w_ir;

  m_proc8_c1 u1 (w_clk, w_pc, w_ir, w_stall);
  m_imem u2 (w_clk, w_pc, 1'b1, w_ir, w_oe);
  initial begin
    `define MM u2.mem
    `include "program_24.txt"
  end
  initial #99 forever #100

  $display("CC%02d %h %h %h %h %b %5d %5d %5d",
  w_cc, u1.r_pc, u1.P1_pc, u1.P2_pc, u1.P3_pc, w_stall,
  u1.w_in1, u1.w_in2, u1.w_alu);
endmodule
```

- 次は、m_simのコードの修正
  - D_DELAYで設定する遅延だけ遅らせる前のスライドの命令メモリに置き換えましょう。

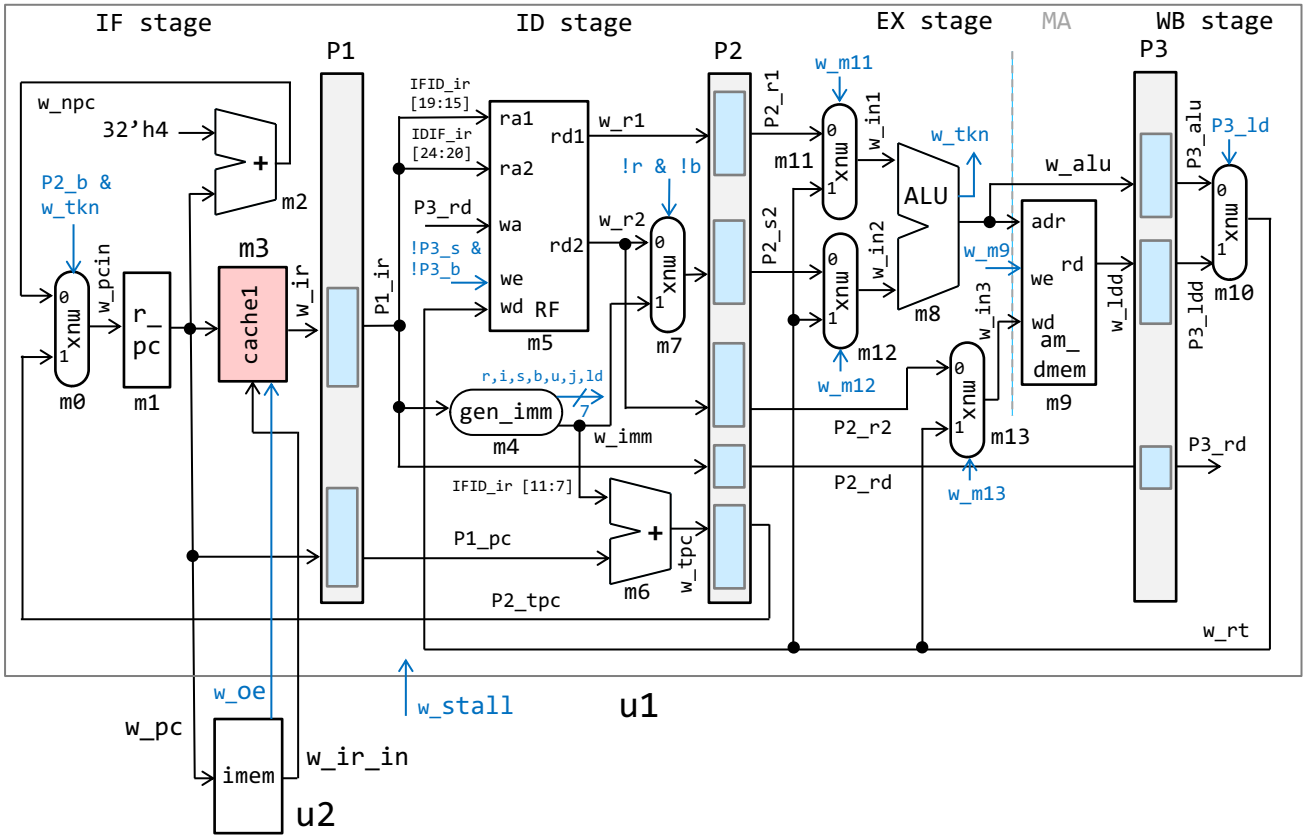
動作確認

- proc8s_c.v, program_24.txtを同じディレクトリーにコピーする.
- シミュレーションにより，動作確認を行う.
  - この実行では、正しい結果の5050が出力されて、その実行には2550サイクルが必要になる.





proc8s 4-stage pipelining for cache



命令キャッシュの実装 (1/2)

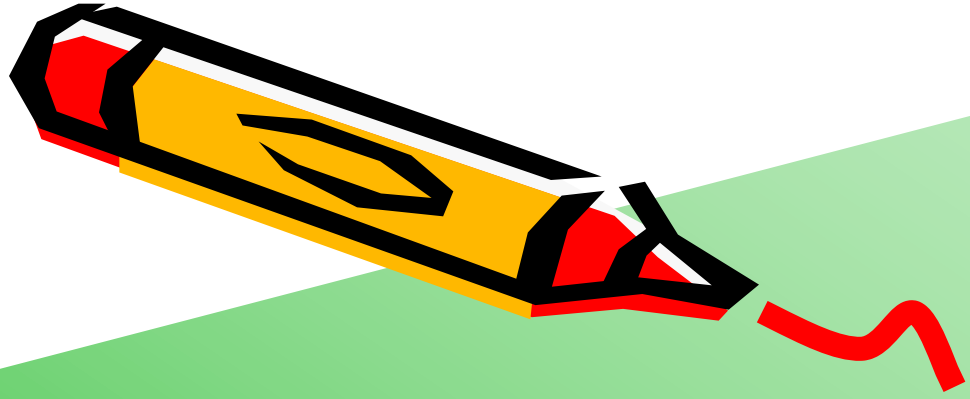
- proc8s_icache.v を修正して，命令キャッシュを実装する.
- シミュレーションにより，正しくキャッシュが実装されていることを示す． 以下に注意すること.
  - 実行結果が正しいこと: 5050
  - 実行サイクル数が，キャッシュ実装前の2550サイクルよりも少なくなっていること
- キャッシュヒットの回数とミスの回数を計測する.

Check Point 6

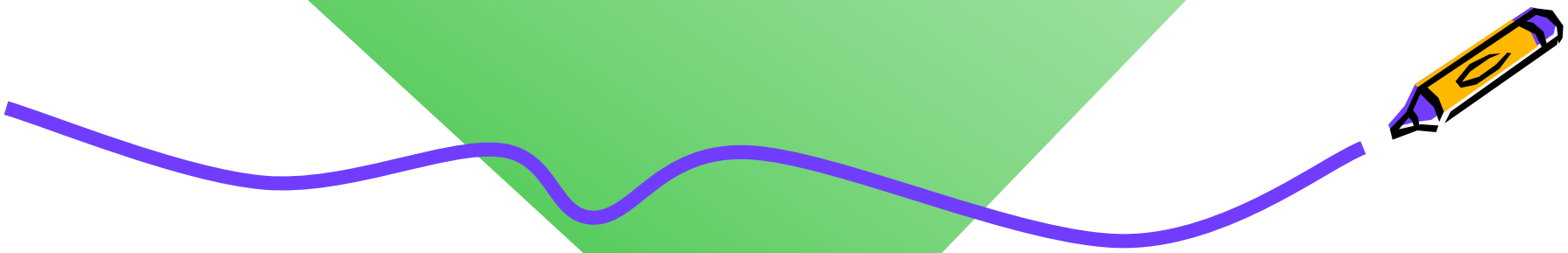
命令キャッシュの実装 (2/2)

proc8s_icache.v の一部

```
module m_cache1(w_clk, w_adr, w_hit, w_dout, w_wadr, w_we, w_wd);
          /* Please describe here by yourself */
endmodule

module m_sim(w_clk, w_cc);
  // Some register and wire definitions are omitted
  reg [31:0] r_miss = 0, r_hit = 0;

  m_proc8_c2 u1 (w_clk, w_pc, w_ir, w_oe, w_re);
  m_imem u2 (w_clk, w_pc, w_re, w_ir, w_oe);
  initial begin
    `define MM u2.mem
    `include "program_24.txt"
  end
  initial #99 forever #100
    $display("CC%02d %h %h %h %h %5d %5d %5d",
    w_cc, u1.r_pc, u1.P1_pc, u1.P2_pc, u1.P3_pc, u1.w_stall,
    u1.w_in1, u1.w_in2, u1.w_alu);

  always @(posedge w_clk)
     /* Please describe here by yourself */
     $display("hit = %4d, miss = %4d", r_hit, r_miss);
  end
end
endmodule
```



References



References (1/2)

- Computer Architecture support page
  - http://www.arch.cs.titech.ac.jp/lecture/CA/
- Computer Logic Design support page
  - http://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/

References (2/2)

- Xilinx Vivado Design Suite
  - https://japan.xilinx.com/products/design-tools/vivado.html
- Digilent Arty A7-35 A7: FPGA Trainer Board
  - https://reference.digilentinc.com/reference/programmable-logic/arty-a7/start
- Digilent Nexys 4 DDR Atrix-7 FPGA
  - https://store.digilentinc.com/nexys-4-ddr-artix-7-fpga-trainer-board-recommended-for-ece-curriculum/
- Verilog HDL
  - https://ja.wikipedia.org/wiki/Verilog