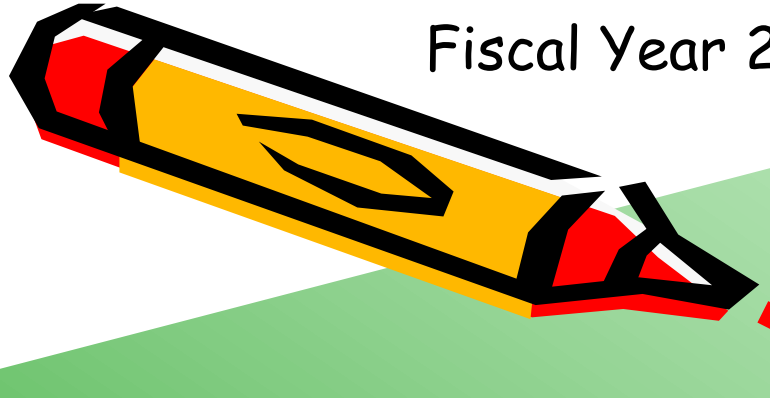


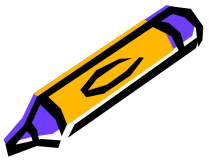
Fiscal Year 2022

Ver. 2022-12-19a

Course number: CSC.T433
School of Computing,
Graduate major in Computer Science

Advanced Computer Architecture

3. HDL, Single-cycle processor, and Memory Hierarchy Design

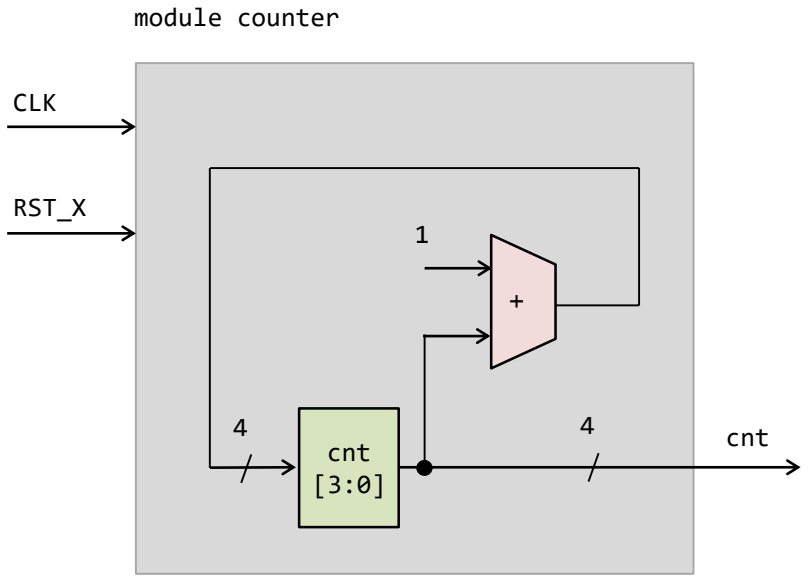


www.arch.cs.titech.ac.jp/lecture/ACA/
Room No.W831, HyFlex
Mon 13:45-15:25, Thr 13:45-15:25

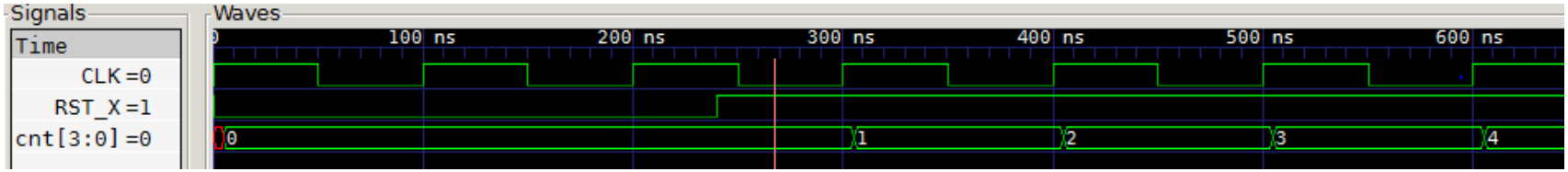
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp

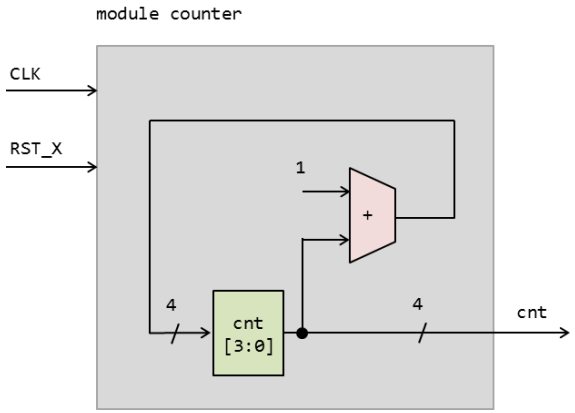
Sample circuit 1

- 4-bit counter
  - synchronous reset
  - negative-logic reset, initialize or reset the value of register cnt to zero if RST_X is low



Sample Verilog HDL Code





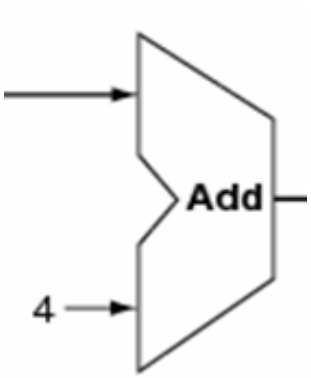
counter.v

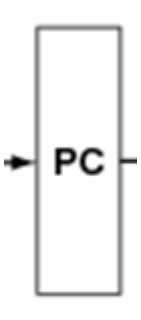
```
 8  module top();
 9    reg CLK, RST_X;
10    wire [3:0] w_cnt;
11
12    initial begin CLK = 1; forever #50 CLK = ~CLK; end
13    initial begin RST_X = 0; #240 RST_X = 1; end
14    initial #800 $finish();
15    initial begin
16      $dumpfile("wave.vcd");
17      $dumpvars(0, cnt1);
18    end
19    always @(posedge CLK) $write("cnt1: %d %x¥n", RST_X, w_cnt);
20
21    counter cnt1(CLK, RST_X, w_cnt);
22  endmodule
23
24
25  module counter(CLK, RST_X, cnt);
26    input  wire CLK, RST_X;
27    output reg [3:0] cnt;
28
29    always @(posedge CLK) begin
30      if(!RST_X) cnt <= #5 0;
31      else       cnt <= #5 cnt + 1;
32    end
33  endmodule
```

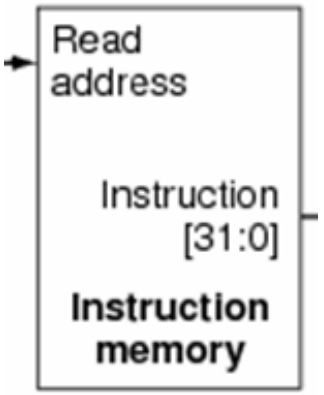
Single-cycle implementation of processors

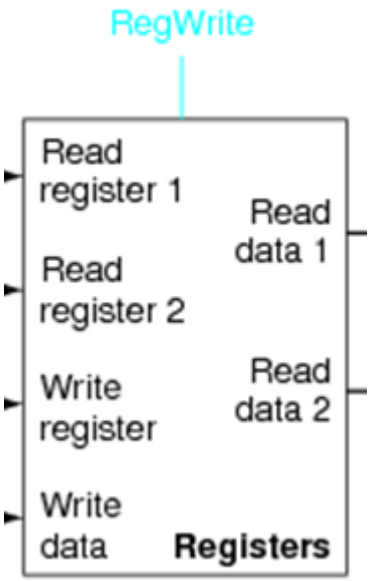
- Single-cycle implementation also called single clock cycle implementation is the implementation in which an instruction is executed in one clock cycle. While easy to understand, it is too slow to be practical.

Some building blocks of processor datapath







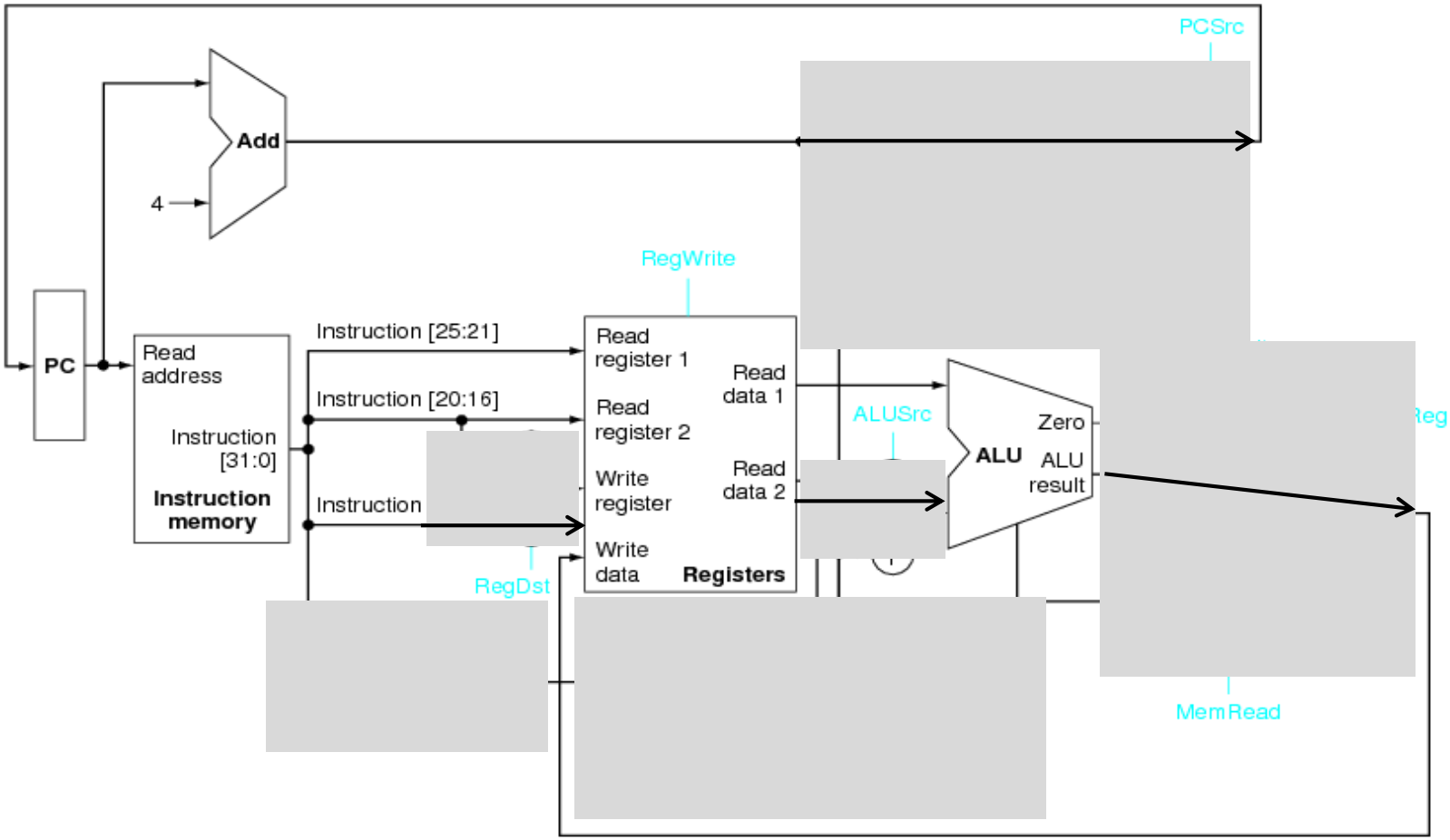


We use 8K word memory.

Datapath of single-cycle processor supporting ADD

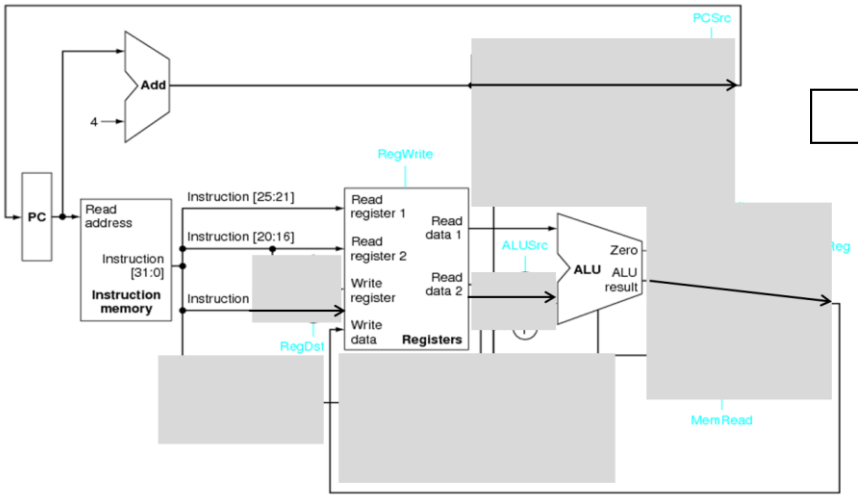
IR[25:21]	IR[20:16]	IR[15:11]			
op	rs	rt	rd	shamt	funct

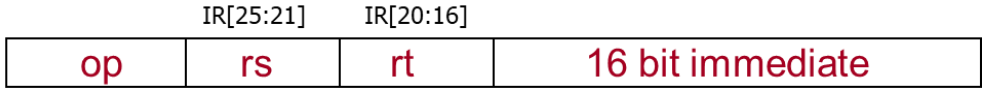
```
0x800    add $t0, $s1, $s2   [ add $8, $17, $18 ]
```



$17 = 3
$18 = 4

Verilog HDL Code of proc01



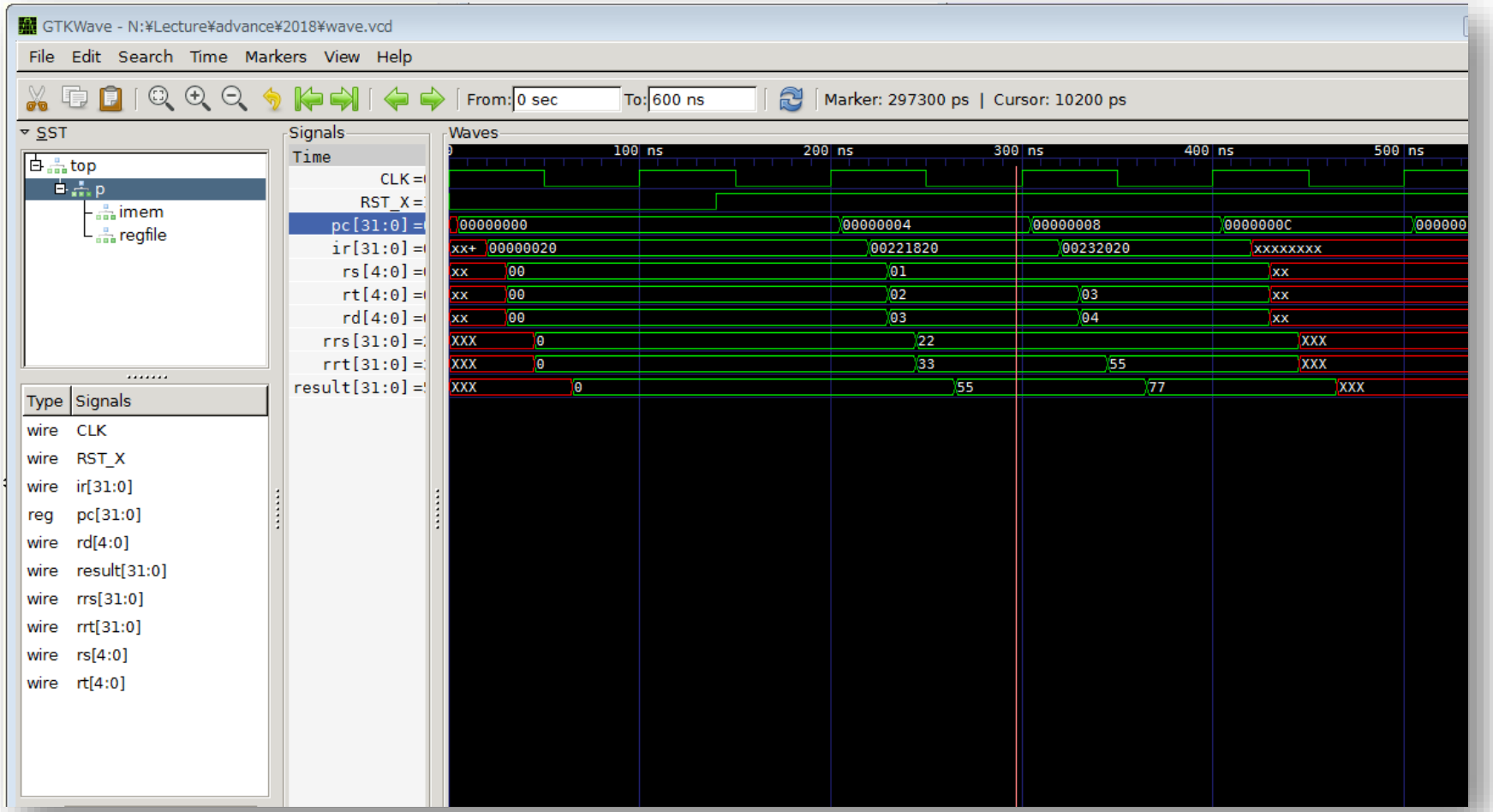


I format

```
module PROCESSOR_01(CLK, RST_X);
  input wire CLK, RST_X;

  reg  [31:0] pc;
  wire [31:0] ir;
  wire [31:0] rrs, rrt;

  always @(posedge CLK) pc <= #5 (!RST_X) ? 0 : pc + 4;

  IMEM imem(CLK, pc, ir); /* instruction memory */

  wire [4:0]  #10 rs = ir[25:21];
  wire [4:0]  #10 rt = ir[20:16];
  wire [4:0]  #10 rd = ir[15:11];
  wire [31:0] #20 result = rrs + rrt; /* ALU */

  GPR regfile(CLK, rs, rt, rd, result, 1, rrs, rrt); /* register file */
endmodule
```

proc01.v

Waveform of proc01



this slide is to be used as a whiteboard

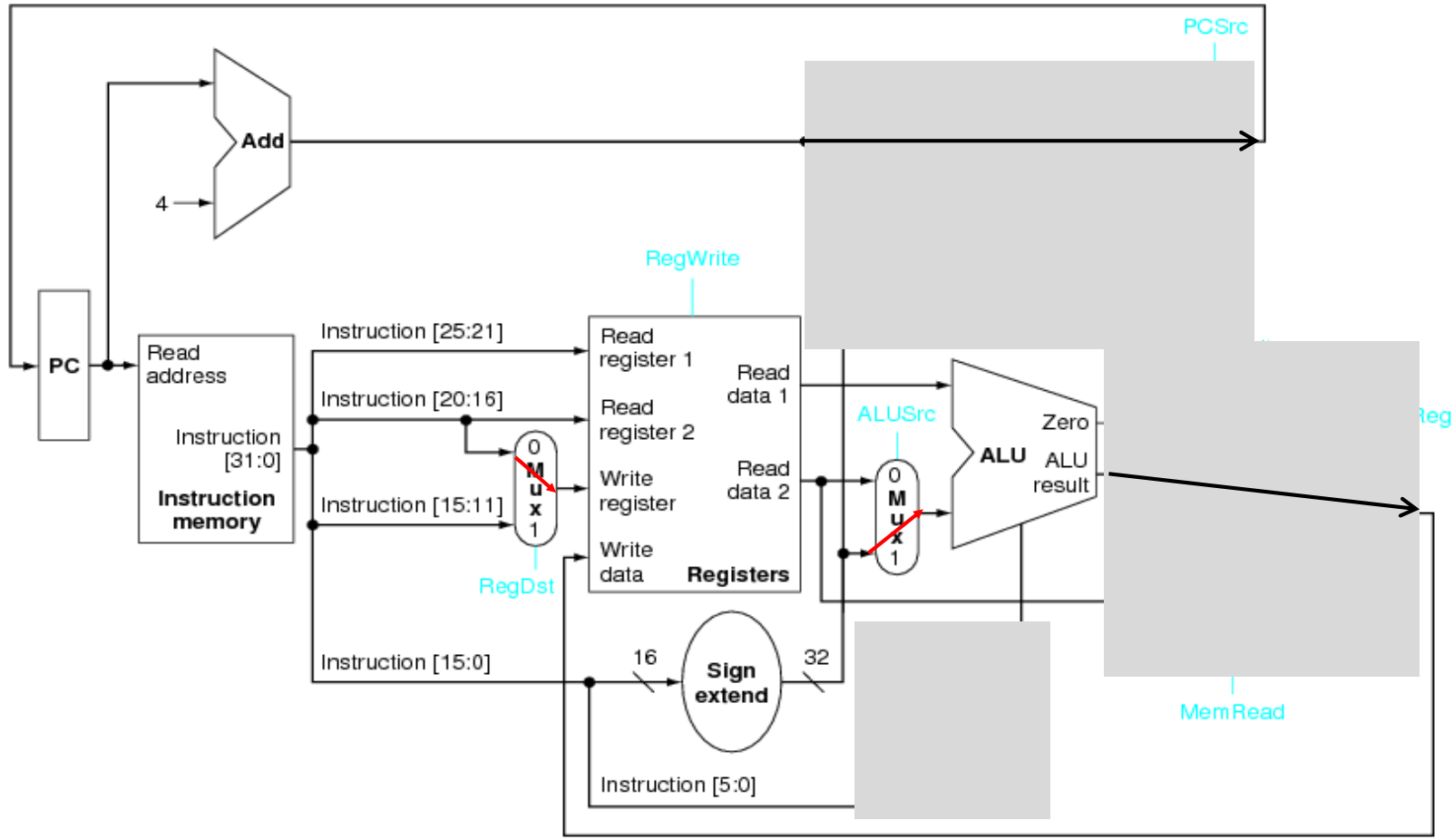
Datapath of processor supporting ADD and ADDI

IR[25:21] IR[20:16]

op	rs	rt	16 bit immediate

I format

```
0x804   addi $t1, $t0, 3      [ addi $9, $8, 3 ]
```



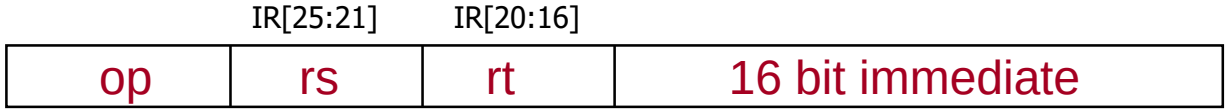
$8 = 7

Assignment 2

1. Design a single-cycle processor supporting MIPS add, addi instructions in Verilog HDL. Please download proc01.v from the support page and refer to it.
2. Verify the behavior of designed processor using following assembly code
   - `add  $0, $0, $0   # {6'h0, 5'd0, 5'd0, 5'd0, 5'd0, 6'h20}`
   - `addi $7, $0, 3    # {6'h8, 5'd0, 5'd7, 16'd3}`
   - `addi $8, $0, 5    # {6'h8, 5'd0, 5'd8, 16'd5}`
   - `add  $9, $7, $8   # {6'h0, 5'd7, 5'd8, 5'd9, 5'd0, 6'h20}`
3. Submit a report printed on A4 paper at the beginning of the next lecture on Monday. Or,
   Submit your report in a PDF file via E-mail (kise [at] c.titech.ac.jp) by the beginning of the next lecture on Monday.
   - The report should include a block diagram, a source code in Verilog HDL, and obtained waveforms of your design.

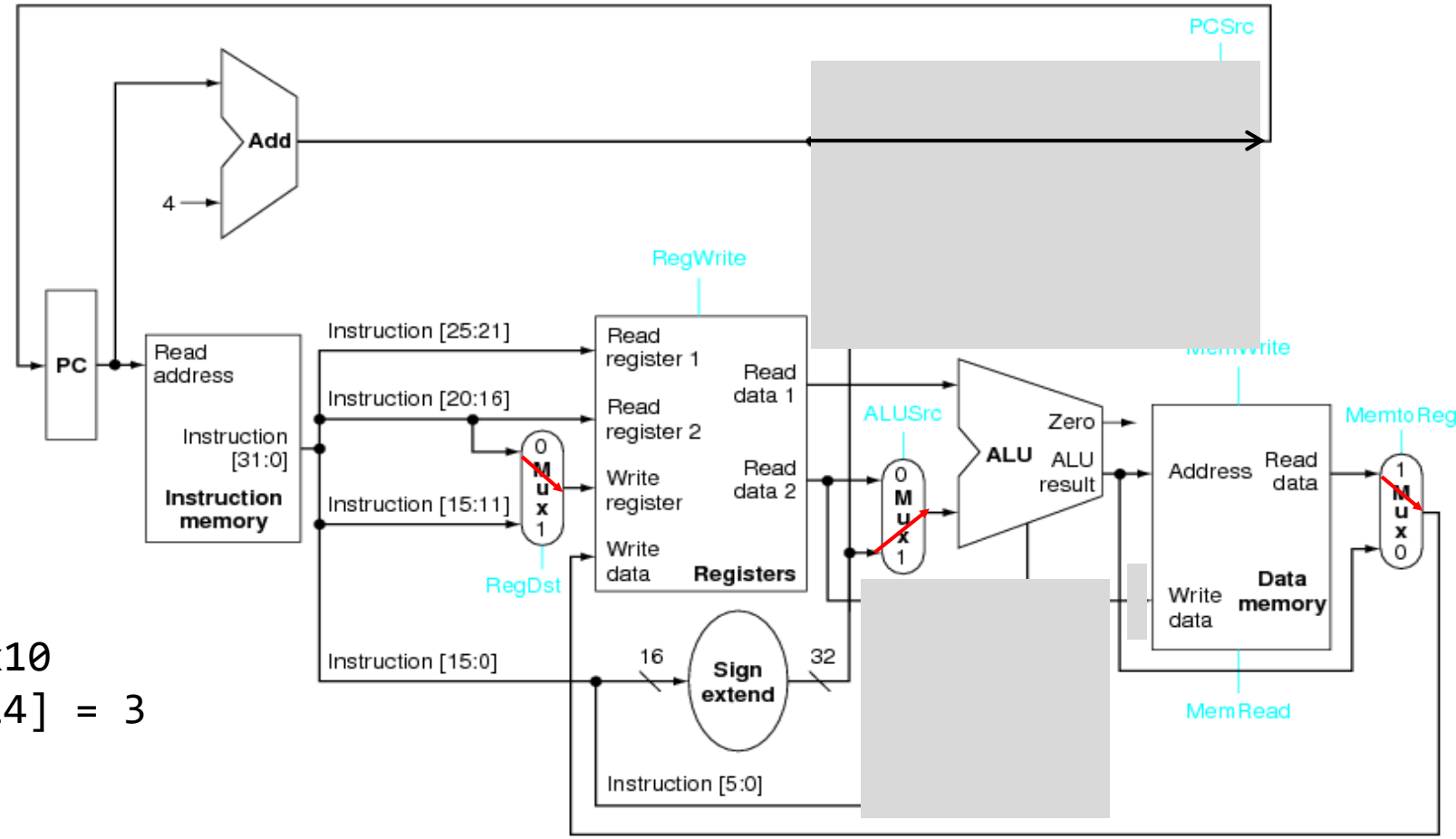
this slide is to be used as a whiteboard

Datapath of processor supporting ADD, ADDI, LW



I format

```
0x808   lw $t2, 4($t0)        [ lw $10, 4($8) ]
```

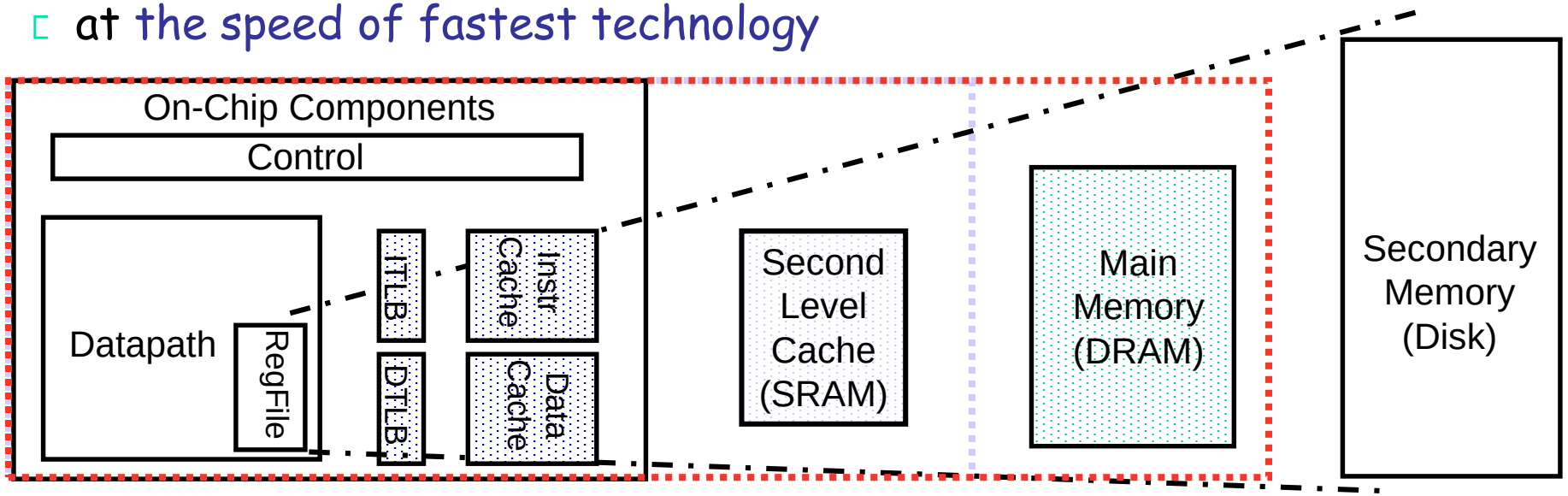



```
$8 = 0x10
mem[0x14] = 3
```

this slide is to be used as a whiteboard

A Typical Memory Hierarchy

- By taking advantage of the principle of locality in time and space
  - Present much memory in the cheapest technology
  - at the speed of fastest technology

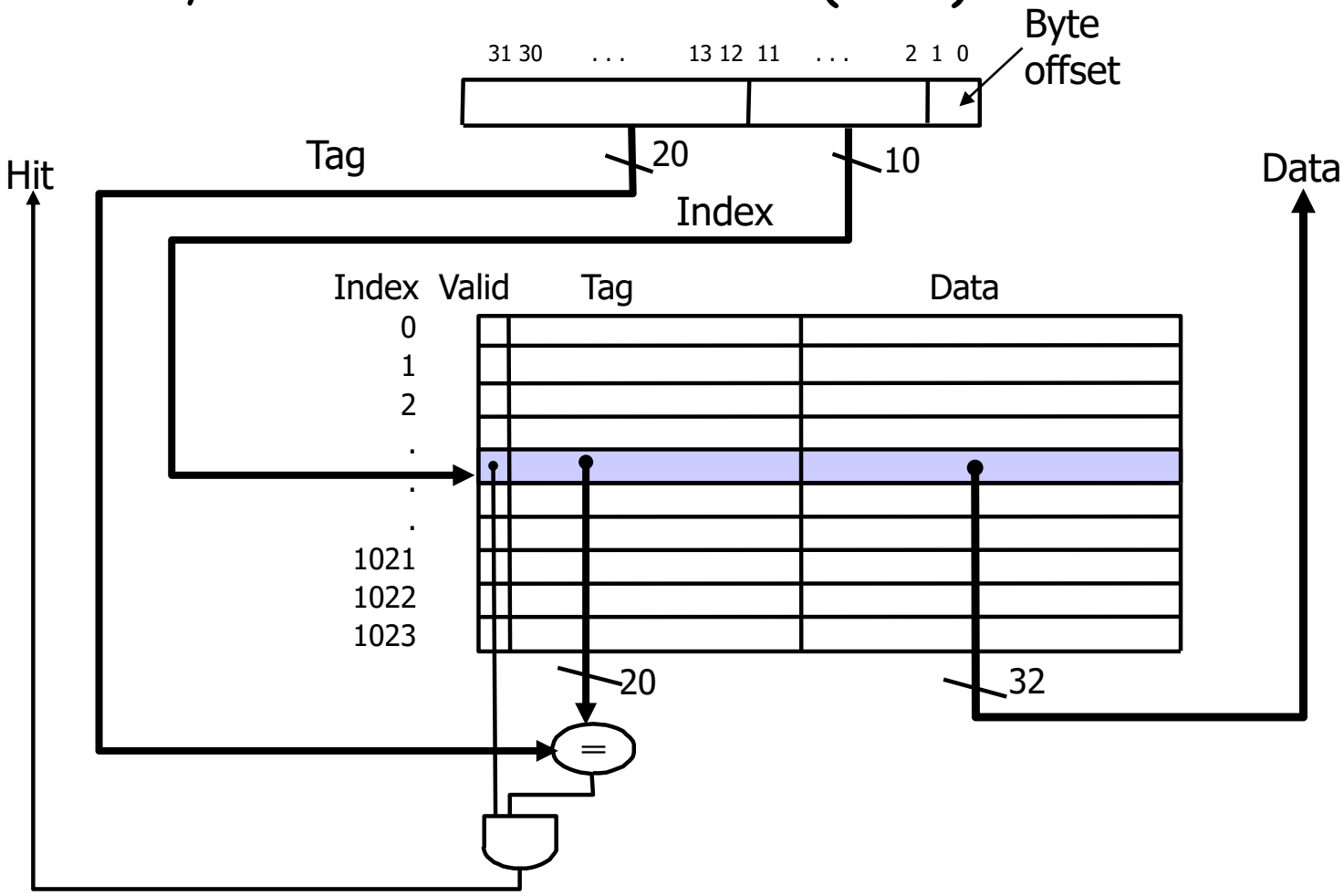


Speed (%cycles):	½'s	1's	10's	100's	1,000's
Size (bytes):	100's	K's	10K's	M's	G's to T's
Cost:	highest				lowest

TLB: Translation Lookaside Buffer

MIPS Direct Mapped Cache Example

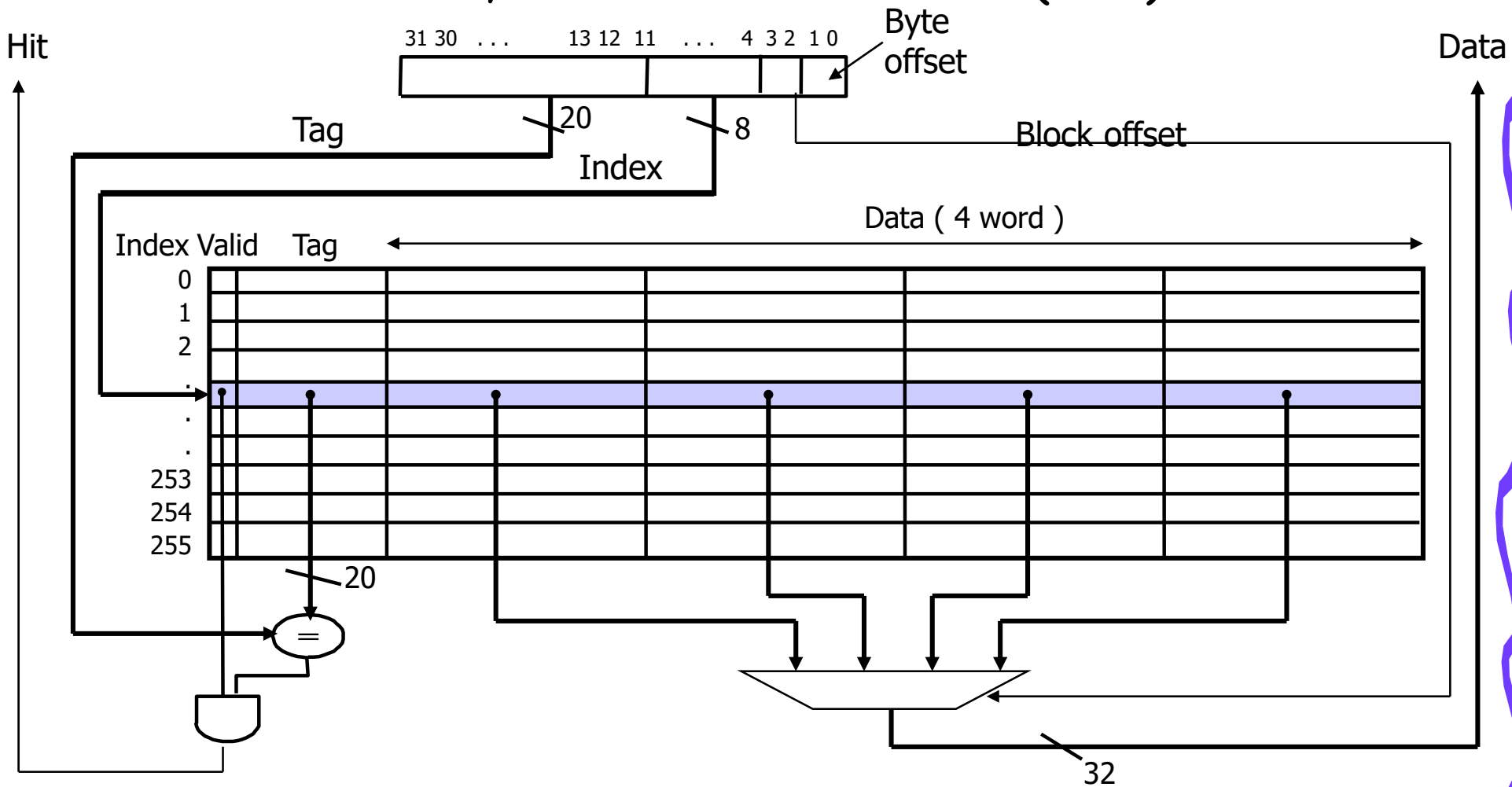
- One word/block, cache size = 1K words (4KB)



What kind of locality are we taking advantage of?

Multiword Block Direct Mapped Cache

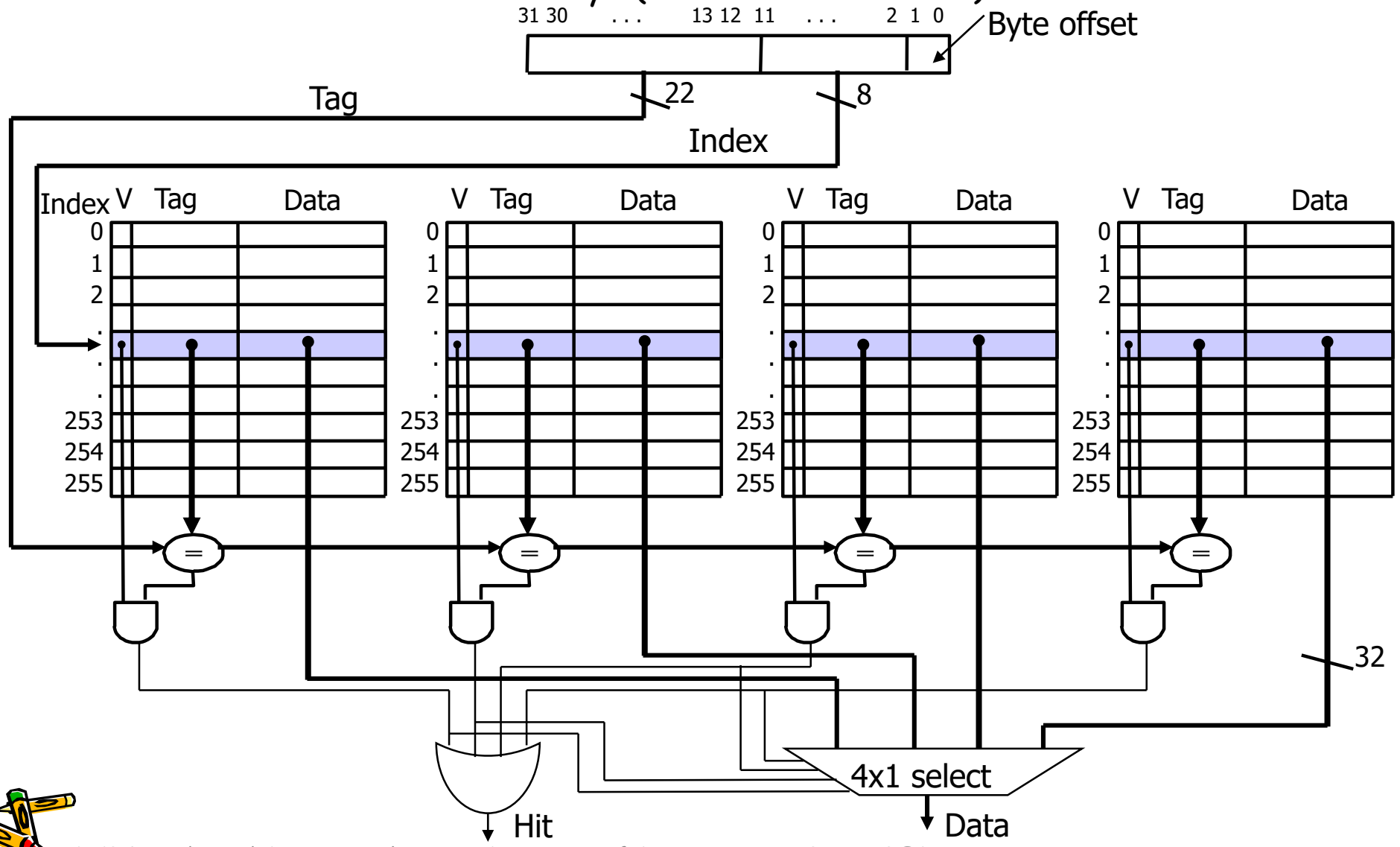
- Four words/block, cache size = 1K words (4KB)



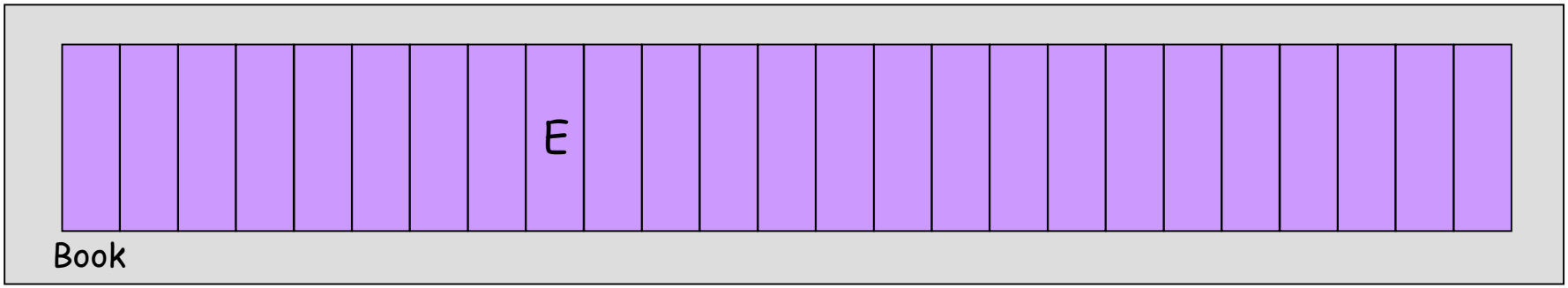
What kind of locality are we taking advantage of?

Four-Way Set Associative Cache

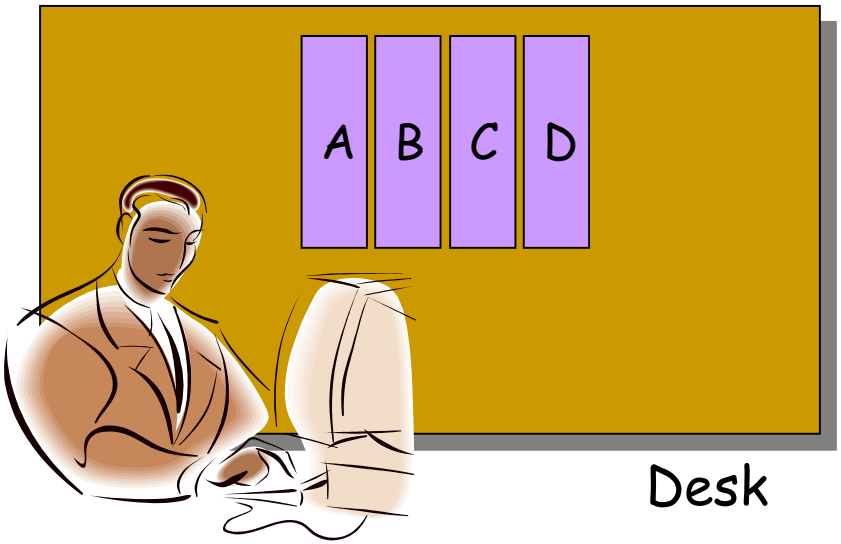
- 2^8 = **256 sets** each with four ways (each with one block)



Cache Associativity & Replacement Policy



Bookshelf



Costs of Set Associative Caches

- N-way set associative cache costs
  - N comparators (delay and area)
  - MUX delay (set selection) before data is available
  - Data available after set selection and Hit/Miss decision.
- When a miss occurs,
  which way's block do we pick for replacement ?
  - Least Recently Used (LRU):
    the block replaced is the one that has been unused for the longest time
    - Must have hardware to keep track of when each way's block was used
    - For 2-way set associative, takes one bit per set →
      set the bit when a block is referenced
      (and reset the other way's bit)
  - Random

this slide is to be used as a whiteboard

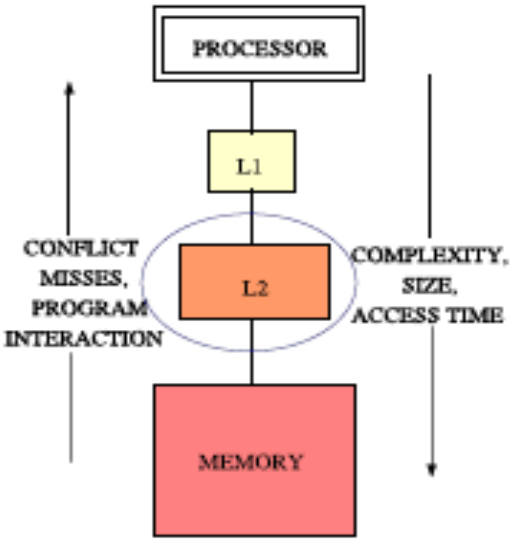
Recommended Reading

- Emulating Optimal Replacement with a Shepherd Cache
  - Kaushik Rajan, Govindarajan Ramaswamy, Indian Institute of Science
  - MICRO-40, pp. 445-454, 2007
  - Session 8: Cache Replacement Policies

- *A quote:*
  "The inherent temporal locality in memory accesses is filtered out by the L1 cache. As a consequence, an L2 cache with LRU replacement incurs significantly higher misses than the optimal replacement policy (OPT). We propose to narrow this gap through a novel replacement strategy that mimics the replacement decisions of OPT."

Memory Hierarchy Design

Memory Hierarchy



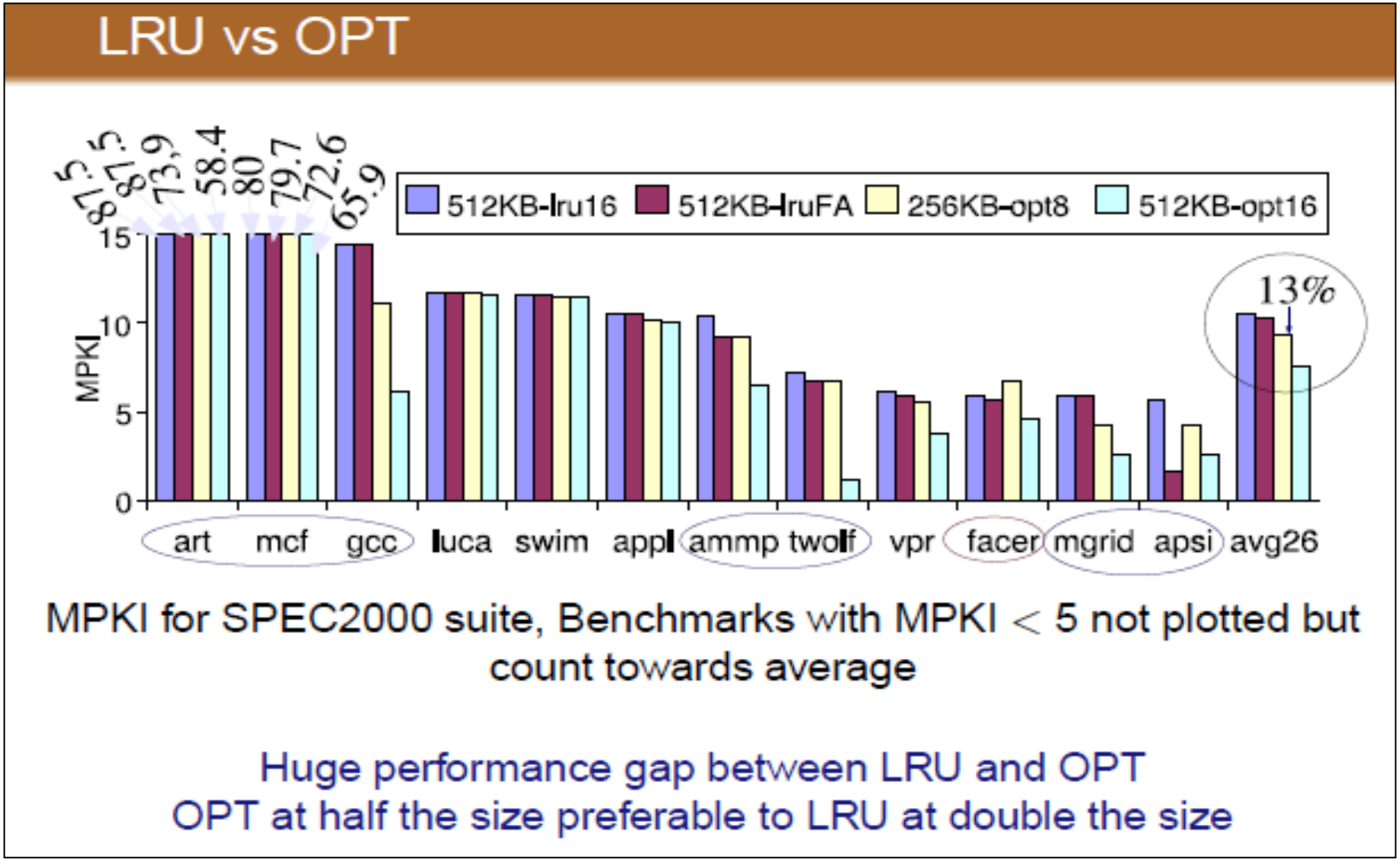
L2 and lower caches

- Objective : Need to reduce expensive memory accesses
- Design : Large size, Higher associativity, Complex design
- Problem : Do not interact with program directly and observe filtered temporal locality

- High Associativity $\Longrightarrow$ replacement policy crucial to performance
- L1 cache services temporal accesses $\Longrightarrow$ Lack of temporal accesses at L2 $\Longrightarrow$ LRU replacement inefficient
- Replacement decisions are taken off the processor critical path

Emulating Optimal Replacement with a Shepherd Cache, MICRO-2007

LRU has room for improvement



MPKI: Miss Per Kilo Instructions

Emulating Optimal Replacement with a Shepherd Cache, MICRO-2007

OPT: Optimal Replacement Policy

The Optimal Replacement Policy

1. Replacement Candidates : On a miss any replacement policy could either choose to replace any of the lines in the cache or choose not to place the miss causing line in the cache at all.
2. Self Replacement : The latter choice is referred to as a self-replacement or a cache bypass

Optimal Replacement Policy

On a miss replace the candidate to which an access is least imminent [Belady1966,Mattson1970,McFarling-thesis]

3. Lookahead Window : Window of accesses between miss causing access and the access to the least imminent replacement candidate. Single pass simulation of OPT make use of lookahead windows to identify replacement candidates and modify current cache state [Sugumar-SIGMETRICS1993]

Emulating Optimal Replacement with a Shepherd Cache, MICRO-2007

Example of Optimal Replacement Policy

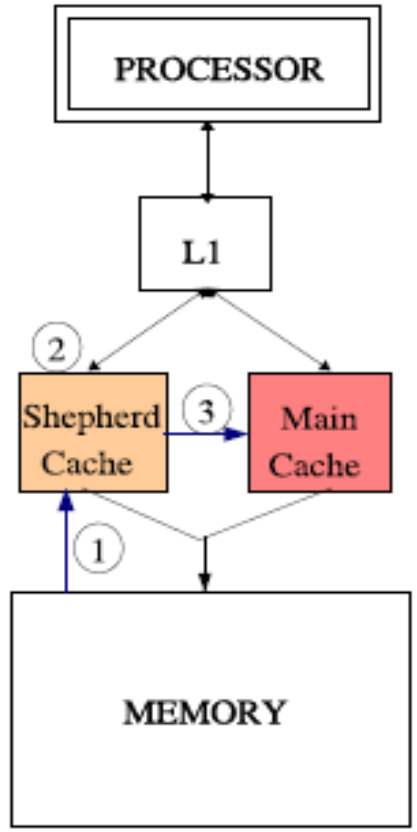
Understanding OPT

Access Sequence	A_5	A_1	A_6	A_3	A_1	A_4	A_5	A_2	A_5	A_7	A_6	A_8
OPT order for A_5		0		1		2	3	4				
OPT order for A_6				0	1	2	3				4	

- Consider 4 way associative cache with one set initially containing lines (A_1, A_2, A_3, A_4), consider the access stream shown in table
- Access A_5 misses, replacement decision proceeds as follows
  1. Identify replacement candidates : $(A_1, A_2, A_3, A_4, A_5)$
  2. Lookahead and gather imminence order : shown in table, lookahead window circled
  3. Make replacement decision : A_5 replaces A_2
- A_6 self-replaces, lookahead window and imminence order in table

Shepherd Cache emulation OPT

Emulating OPT with a Shepherd Cache

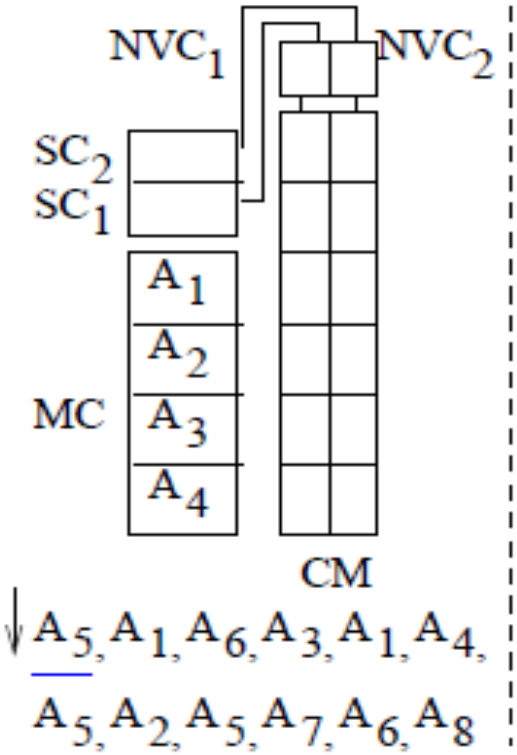


- Split the cache into two logical parts
  - Main Cache (MC) for which optimal replacement is emulated
  - Shepherd Cache (SC) used to provide a lookahead and guide replacements from MC towards OPT
- Operation
  1. Buffer lines temporarily in SC before moving them to MC, SC acts as a FIFO buffer
  2. While in SC, gather imminence information and emulate lookahead
  3. When forced out of SC, make an MC replacement based on the gathered imminence order

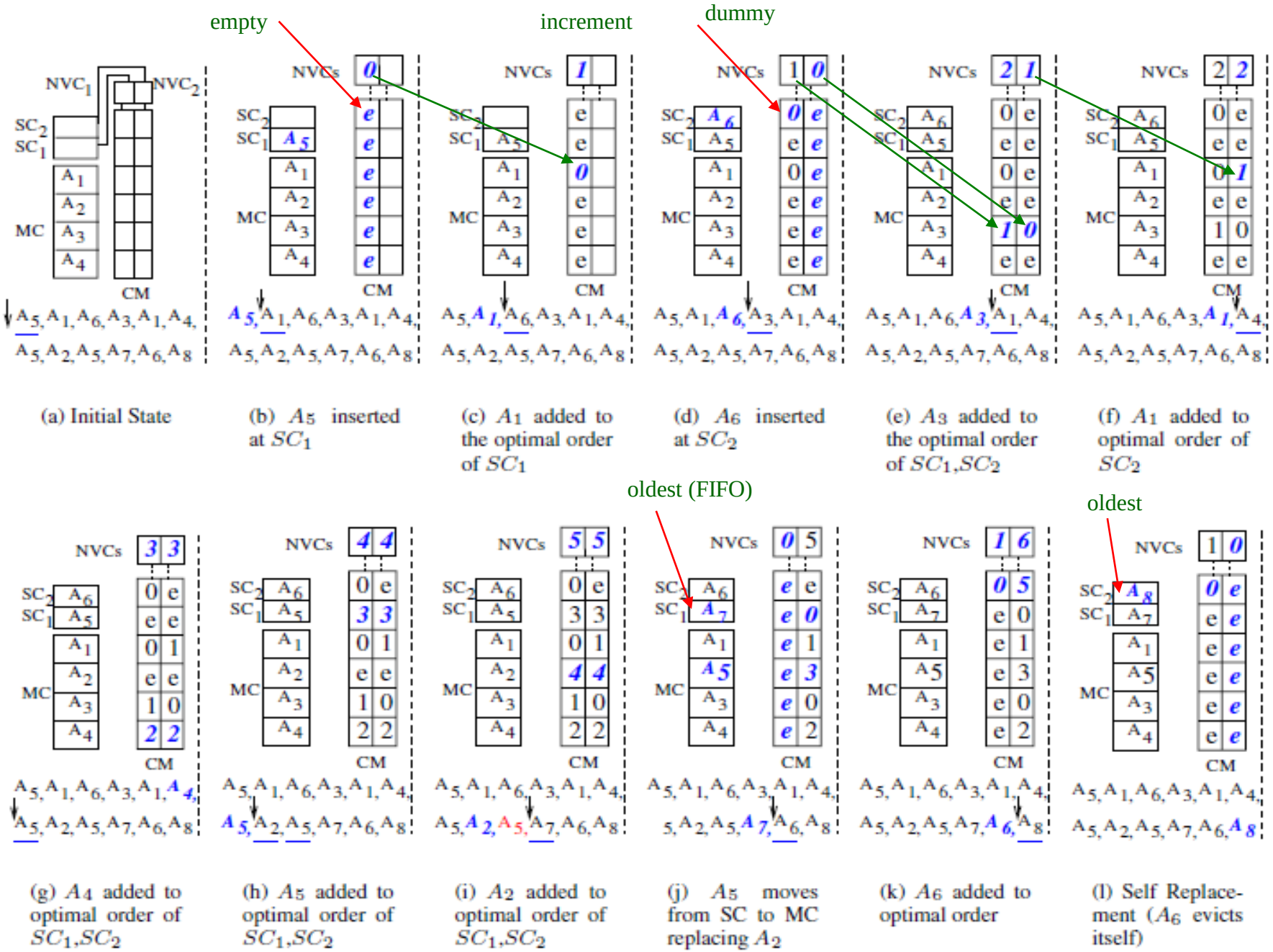
Emulating Optimal Replacement with a Shepherd Cache, MICRO-2007

Shepherd Cache Overview

Overview of Shepherd Caching

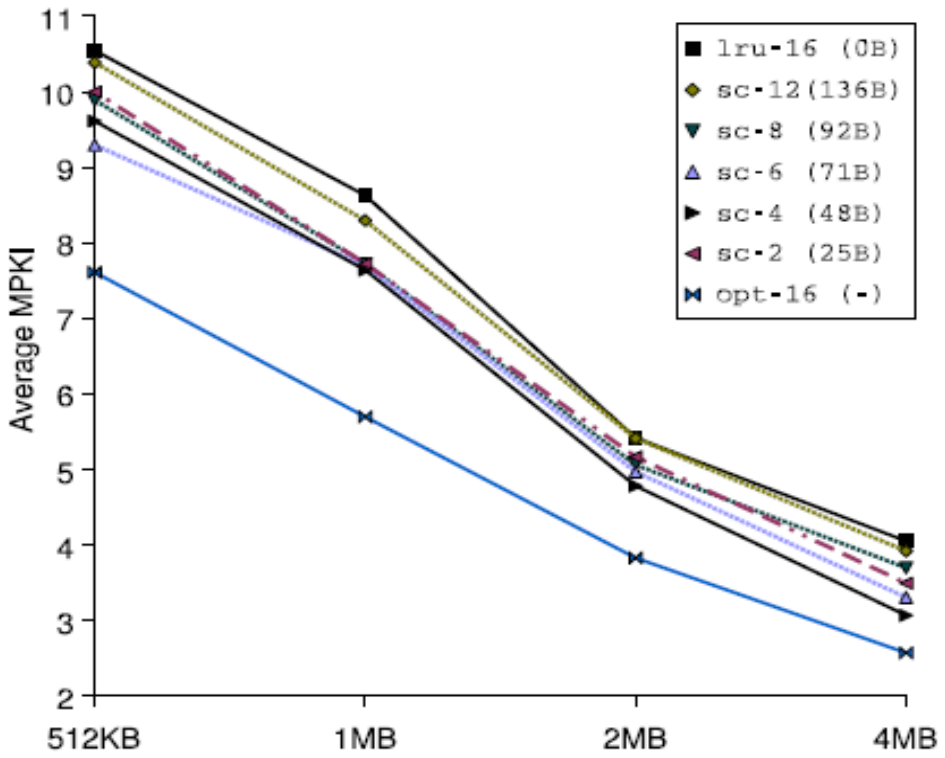


- To emulate MC with 4 ways per set and 2 SC ways per set
- To gather imminence order add a counter matrix (CM)
- CM has one column per SC way to track imminence order w.r.t to it
- CM has one row per SC and MC line as any of them can be a replacement candidate
- Each column has one Next Value Counter (NVC) to track the next value to assign along column


empty
increment
dummy
oldest (FIFO)
oldest
NVC1
NVC2
NVCs
SC2
SC1
MC
CM
(a) Initial State
(b) A5 inserted at SC1
(c) A1 added to the optimal order of SC1
(d) A6 inserted at SC2
(e) A3 added to the optimal order of SC1,SC2
(f) A1 added to optimal order of SC2
(g) A4 added to optimal order of SC1,SC2
(h) A5 added to optimal order of SC1,SC2
(i) A2 added to optimal order of SC1,SC2
(j) A5 moves from SC to MC replacing A2
(k) A6 added to optimal order
(l) Self Replacement (A6 evicts itself)

Shepherd cache bridges 32 – 52% of the gap

Bridging the performance gap



Avg MPKI over SPEC2000 suite

Bridging the LRU-OPT gap

- *SC-4 bridges 32-52% of gap*
- SC moves closer to OPT as cache size increases

MPKI: Miss Per Kilo Instructions

Emulating Optimal Replacement with a Shepherd Cache, MICRO-2007

this slide is to be used as a whiteboard

this slide is to be used as a whiteboard

Course number: CSC.T341

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
Hardware Description Language: Combinational 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/
Zoom
月曜日 3-4時限 10:45-12:25,　木曜日 1-2時限8:50-10:30, 3-4時限10:45-12:25

Sample Verilog HDL code

- ACRi Room にログインする.
  - /home/tu_kise/cld/lec2/ にサンプルコードがあるので、自分のディレクトリにコピーする.

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

- code001.v をシミュレーションする.

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

Inside code001.v

- 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)に一度だけ実行されることを指定する.
- $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.

Inside code002.v

- main.vをcode002.vの内容となるように入力して，シミュレーションする．
- 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
```

Inside code003.v

- main.vをcode003.vの内容となるように入力して，シミュレーションする．
- 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
```

Inside code005.v

- main.vをcode005.vの内容となるように入力して，シミュレーションする．
- 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だけ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だけ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．

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，コードはサポートページを参照してください．

Inside code006.v

- main.vをcode006.vの内容となるように入力して，シミュレーションする．
- $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
```

Inside code007.v

- main.vをcode007.vの内容となるように入力して，シミュレーションする．
- 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
```

Inside code008.v

- main.vをcode008.vの内容となるように入力して，シミュレーションする．
- 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，コードはサポートページを参照してください．

Inside code009.v

- main.vをcode009.vの内容となるように入力して，シミュレーションする．
- 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等に用いると良い．

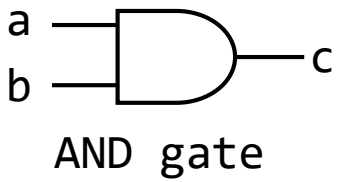
code009.v

```
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
```

Inside code011.v

- 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への値0の代入を表す. wire型にノンブロッキング代入は使えない.



AND gate

code011.v

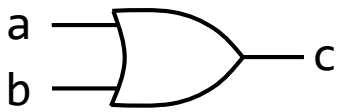
```
module main ();
  reg a, b;
  wire c;
  assign c = a & b;

  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
```

Inside code012.v

- | 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

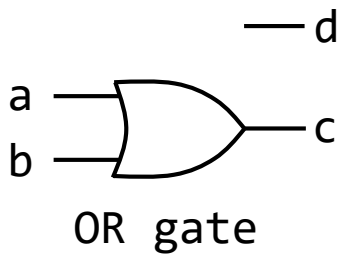
```
module main ();
  reg a, b;
  wire c;
  assign c = a | b;

  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
```

Inside code013.v

- 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された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．
- code012.v の | の部分を他の論理演算子や算術演算子に変更してシミュレーションすること．



code013.v

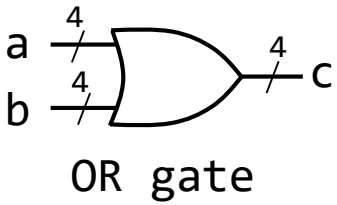
```
module main ();
  reg a, b;
  wire c, d;
  assign c = a | b;

  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
```

Inside code014.v

- 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 が利用できる．



code014.v

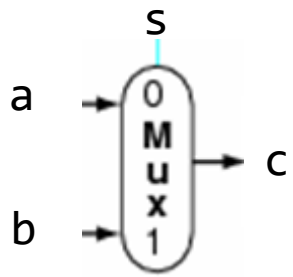
```
module main ();
  reg  [3:0] a, b;
  wire [3:0] c;
  assign c = a | b;

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

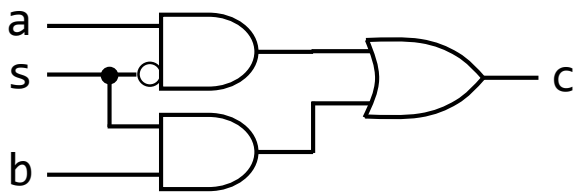
Simulation output

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

Inside code015.v

- 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

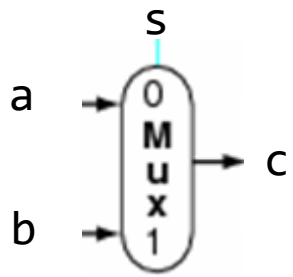
```
module main ();
  reg  a, b, s;
  wire c;
  assign c = (a & ~s) | (s & b);
  initial begin
    #10 s <= 0; a <= 0; b <= 0;
    #10 s <= 0; a <= 0; b <= 1;
    #10 s <= 0; a <= 1; b <= 0;
    #10 s <= 0; a <= 1; b <= 1;
    #10 s <= 1; a <= 0; b <= 0;
    #10 s <= 1; a <= 0; b <= 1;
    #10 s <= 1; 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
```

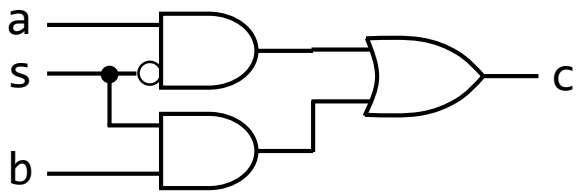
Simulation output

```
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
```

Inside code016.v

- 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

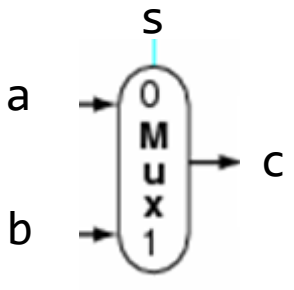
```
11: 0 0 0 -> 0
21: 0 0 1 -> 0
31: 0 1 0 -> 1
41: 0 1 1 -> 1
51: 1 0 0 -> 0
61: 1 0 1 -> 1
71: 1 1 0 -> 0
81: 1 1 1 -> 1
```

Inside code017.v

- モジュールをインスタンス化する例を示す.
- モジュール名とインスタンス名を記述して, その後に入出力端子名を列挙する. 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 を生成する.

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

module m_mux (a, b, s, c);
  input  wire a, b, s;
  output wire c;
  assign c = s ? b : a;
endmodule
```



Simulation output

```
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
```

Inside code010.v

- main.vをcode010.vの内容となるように入力して，シミュレーションする．
- 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．

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，コードはサポートページを参照してください．

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
```

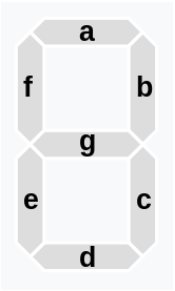
Inside code018.v

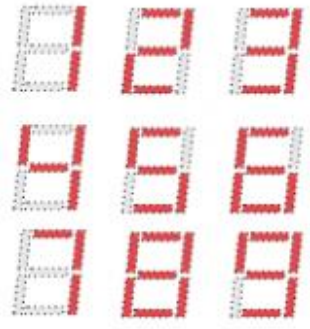
- 0～9を表示する seven-segment LED decoder の例を示す.
- 場合分けの処理を記述するための case文 がある. 記述はC言語と同様.
- モジュールm_7segledでは, 入力の値により, 点灯させるLEDのビットを1とする.
  - r_led の MSBから, LEDのabcdefgのセグメントを割り当てる.

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
```

```
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
```

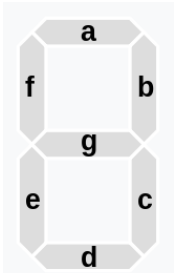
Inside code019.v

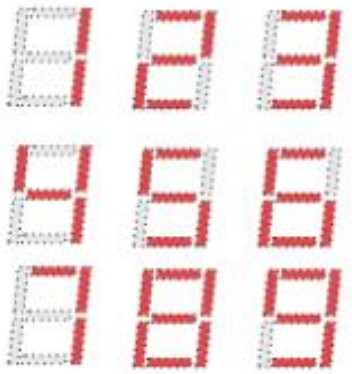
- Seven-segment LED decoder の別の例を示す.
- 関係演算子(==)は等しい時に1'b1となり，そうでなければ1'b0となる.
- code019.v ではreg型は使っていない．このため，m_7segled からは組合せ回路が合成される.
- code018.v の m_7segled から，組合せ回路が合成されるか？順序回路が合成されるか？
  - reg型の信号が常にレジスタに合成されるという訳ではない.

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
```





Inside code020.v

- 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
```

Inside code021.v

- 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

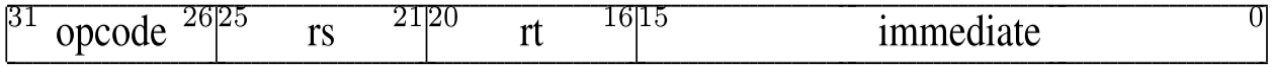
```
module m_7segled (w_in, r_led);
  input  wire [3:0] w_in;
  output reg  [6:0] r_led;
  always @(*) begin
    case (w_in)
      4'd0   : r_led <= 7'b1111110;
      4'd1   : r_led <= 7'b0110000;
      4'd2   : r_led <= 7'b1101101;
      4'd3   : r_led <= 7'b1111001;
      4'd4   : r_led <= 7'b0110011;
      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
```

Inside code022.v

- 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．MIPSアーキテクチャの機械命令で用いられるI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'h1464fffe;
  wire [5:0]  w_op;
  wire [4:0]  w_rs;
  wire [4:0]  w_rt;
  wire [15:0] w_imm;
  initial begin
    #1 $display(" %x -> %x %x %x %x", r_ir, w_op, w_rs, w_rt, w_imm);
    $display(" %b -> ", r_ir);
    $display(" %b %b %b %b", w_op, w_rs, w_rt, w_imm);
  end
  m_decode m_decode0 (r_ir, w_op, w_rs, w_rt, w_imm);
endmodule

module m_decode (w_ir, w_op, w_rs, w_rt, w_imm);
  input  wire [31:0] w_ir;
  output wire [5:0]  w_op;
  output wire [4:0]  w_rs;
  output wire [4:0]  w_rt;
  output wire [15:0] w_imm;
  assign w_op  = w_ir[31:26];
  assign w_rs  = w_ir[25:21];
  assign w_rt  = w_ir[20:16];
  assign w_imm = w_ir[15:0];
endmodule
```

Simulation output

```
1464fffe -> 05 03 04 fffe
00010100011001001111111111111110 ->
000101 00011 00100 1111111111111110
```

code022.v

Inside code023.v

- 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.

Inside code024.v

- 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
```

Inside code025.v

- 論理シフト演算 (>>, <<) の例を示す．算術シフトについては別の演習で．
- 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
```

Inside code026.v

- 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
```

Course number: CSC.T341

コンピュータ論理設計
Computer Logic Design

3. 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/

Inside code051.v

- シミュレーションのためのクロックの記述例を示す.
- 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.
- 波形を確認することで、クロック周波数10MHz（クロック周期100ns）のクロックが生成されていることがわかる.

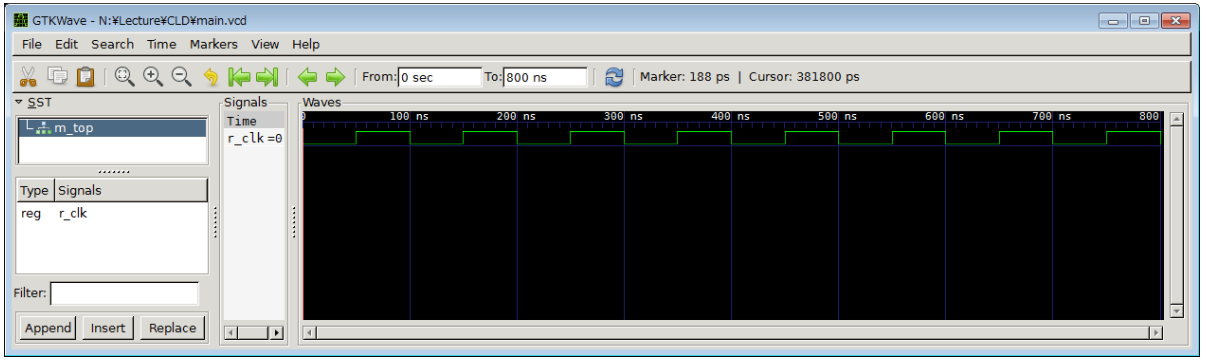
code051.v

```
`timescale 1ns/100ps
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



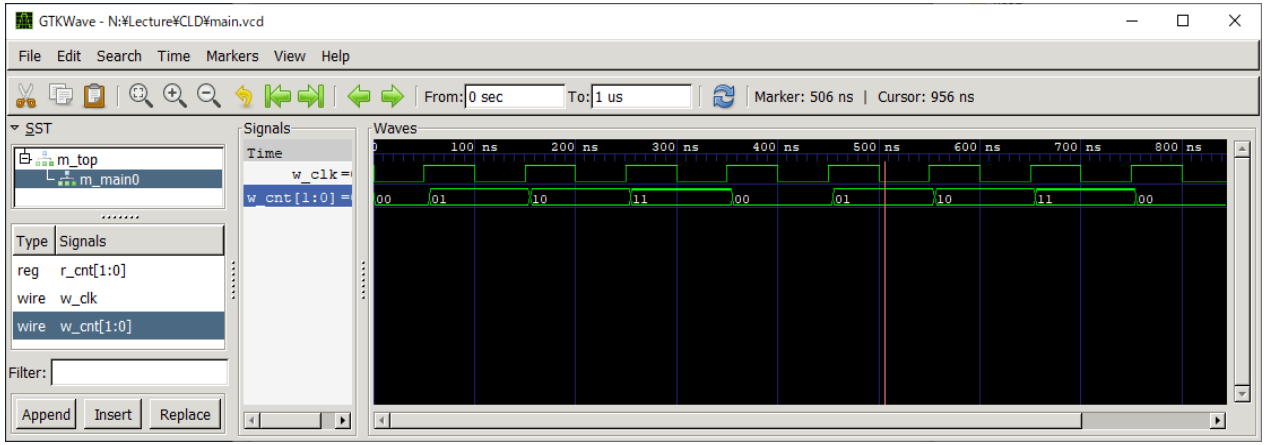
Inside code052.v

- 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、最初の値を 0 とする。
- 2ビットカウンタなので, 最大値3の次は0となる点に注意.
- iverilogでシミュレーションし, GTKWaveで波形を確認する.

code052.v

```
`timescale 1ns/100ps
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



Inside code053.v

- 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、最初の値を 0 とする。
- 2ビットカウンタなので, 最大値3の次は0となる点に注意.
- iverilogでシミュレーションし, GTKWaveで波形を確認する.

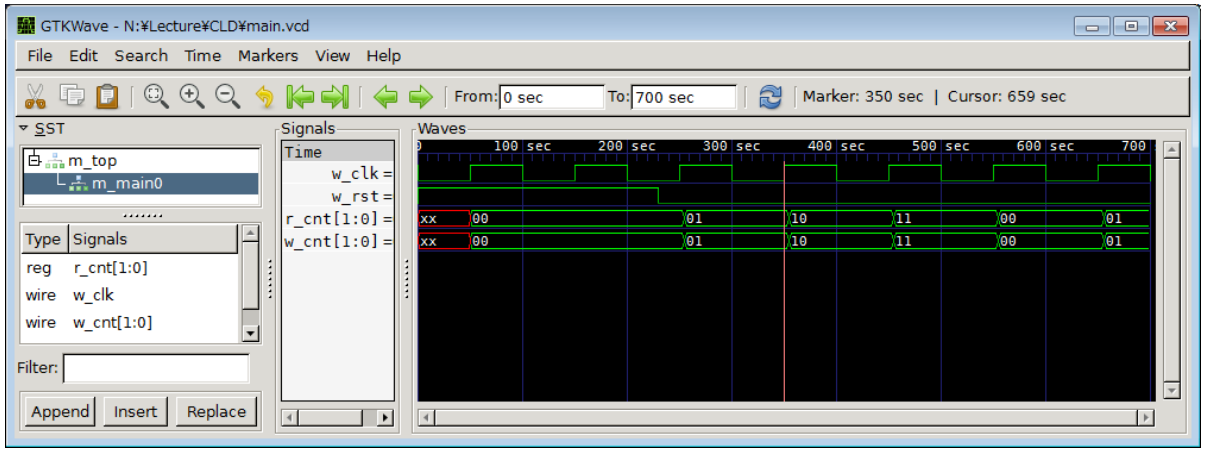
code053.v

```
`timescale 1ns/100ps
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
```

Inside code054.v

- 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の時には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.
- 2ビットカウンタなので，最大値3の次は0となる点に注意.
- iverilogでシミュレーションし，GTKWaveで波形を確認する.

Waveform



code054.v

```
`timescale 1ns/100ps
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
```

Inside code055.v and code056.v

- 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

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
```

Inside code057.v

- 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。
- 回路の出力を4ビットのLEDに変更する。

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
```