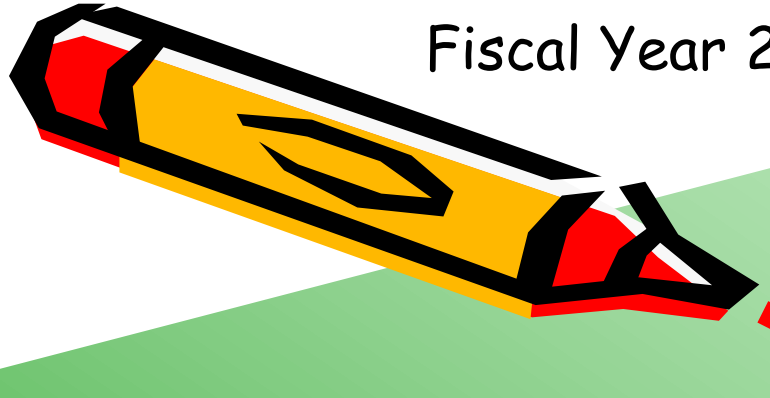


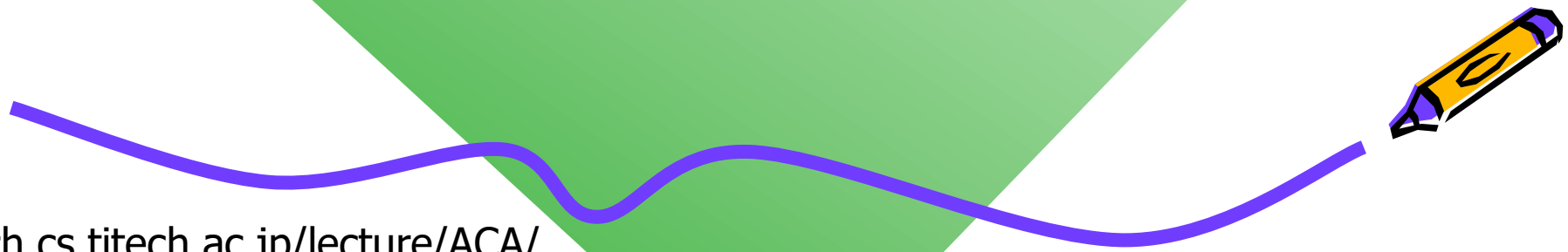
Fiscal Year 2024

Ver. 2024-12-16a

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

Advanced Computer Architecture

3. HDL, single-cycle processor

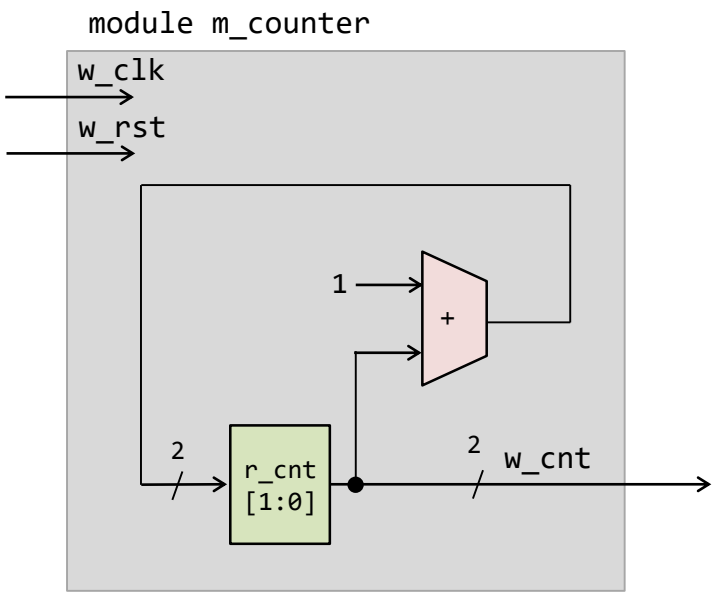


www.arch.cs.titech.ac.jp/lecture/ACA/
Room No. W8E-308, Lecture (Face-to-face)
Mon 13:30-15:10, Thr 13:30-15:10

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

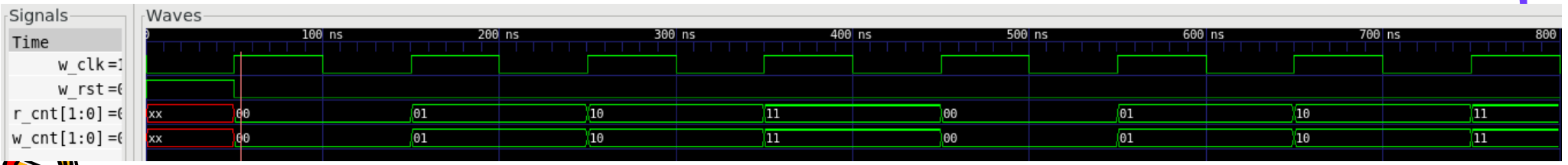
Sample circuit 2

- 2-bit counter as a sample sequential circuit



circuit2.v

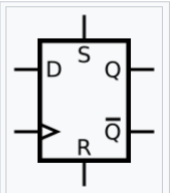
```
`timescale 1ns/100ps
module top();
  reg r_clk = 0;
  always #50 r_clk = ~r_clk;
  reg r_rst = 1;
  always @(posedge r_clk) r_rst <= 0;
  wire [1:0] w_cnt;
  m_counter m1 (r_clk, r_rst, w_cnt);
  initial begin $dumpfile("dump.vcd"); $dumpvars(0); end
  initial #800 $finish;
endmodule

module m_counter (
  input wire w_clk,
  input wire w_rst,
  output wire [1:0] w_cnt
);
  reg [1:0] r_cnt;
  always@(posedge w_clk) r_cnt <= (w_rst) ? 0 : r_cnt + 1;
  assign w_cnt = r_cnt;
endmodule
```



Register (D flip-flop)

D flip-flop [edit]

The D flip-flop is widely used, and known as a "data" flip-flop. The D flip-flop captures the value of the D-input at a definite portion of the clock cycle (such as the rising edge of the clock). That captured value becomes the Q output. At other times, the output Q does not change.[23][24] The D flip-flop can be viewed as a memory cell, a zero-order hold, or a delay line.[25]



D flip-flop symbol

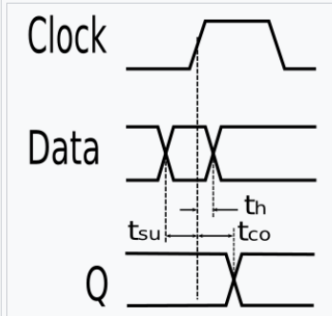
Truth table:

Clock	D	Q_{next}
Rising edge	0	0
Rising edge	1	1
Non-rising	X	Q

(*X* denotes a *don't care* condition, meaning the signal is irrelevant)

Timing parameters [edit]

The input must be held steady in a period around the rising edge of the clock known as the aperture. Imagine taking a picture of a frog on a lily-pad.[28] Suppose the frog then jumps into the water. If you take a picture of the frog as it jumps into the water, you will get a blurry picture of the frog jumping into the water—it's not clear which state the frog was in. But if you take a picture while the frog sits steadily on the pad (or is steadily in the water), you will get a clear picture. In the same way, the input to a flip-flop must be held steady during the **aperture** of the flip-flop.



Flip-flop setup, hold and clock-to-output timing parameters

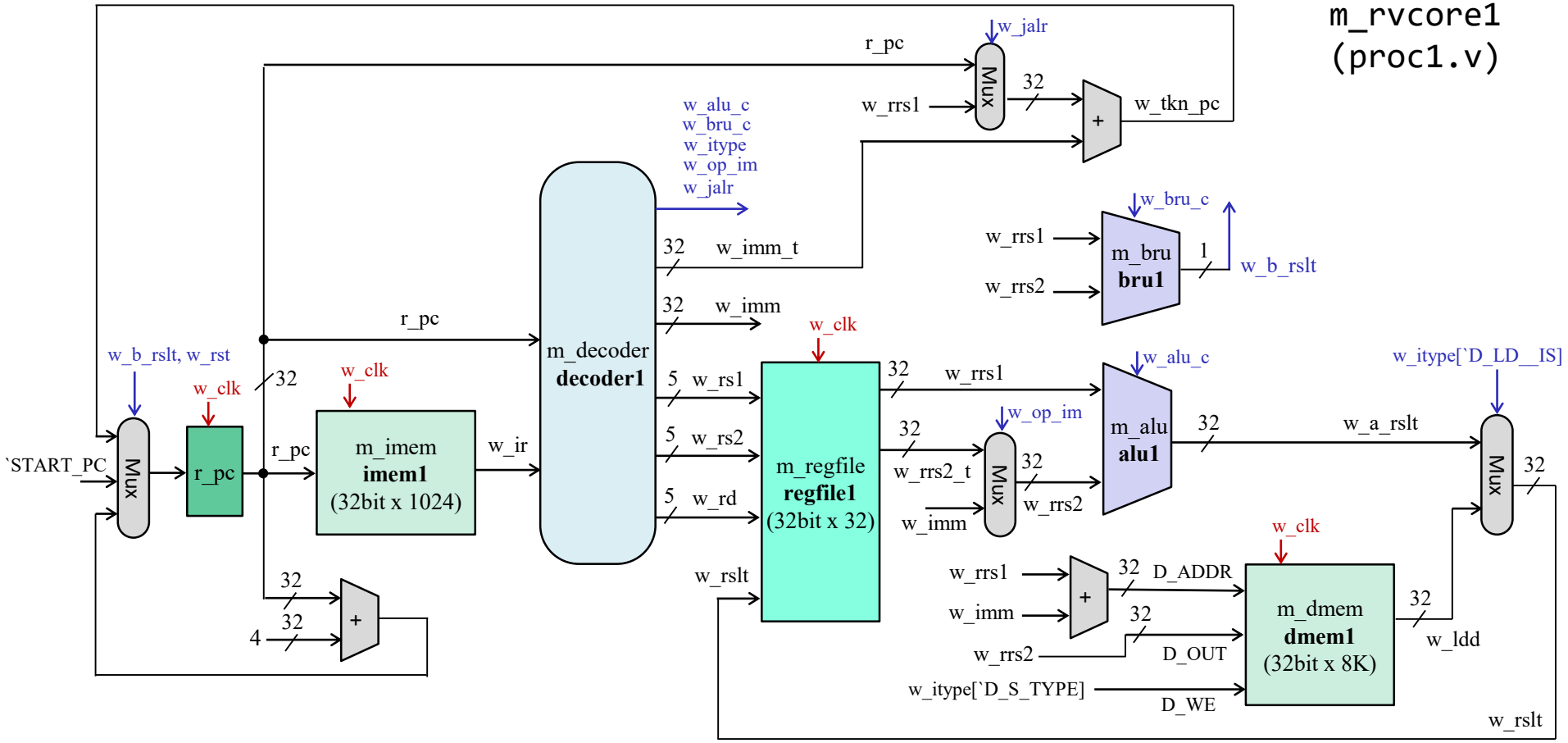
Wikipedia

Setup time is the minimum amount of time the data input should be held steady **before** the clock event, so that the data is reliably sampled by the clock.

Hold time is the minimum amount of time the data input should be held steady **after** the clock event, so that the data is reliably sampled by the clock.

m_rvcore (RV32I, single-cycle processor)

- around 40MHz operating frequency for Arty A7 FPGA board
- lb, lbu, lh, lhu, sb, sh are not supported



Steps in processing an instruction

- IF: Instruction Fetch
  fetch an instruction from instruction memory or instruction cache
- ID: Instruction Decode
  decode an instruction and read input operands from register file
- EX: Execution
  perform operation, calculate an address of lw/sw
- MEM: Memory Access
  access data memory or data cache for lw/sw
- WB: Write Back
  write operation result and loaded data to register file

Sample assembly code in RISC-V

- sample assembly code in the instruction memory
- the leftmost number is the instruction memory address where the instruction is stored
- the first register x0 is zero register with hardwiring 0

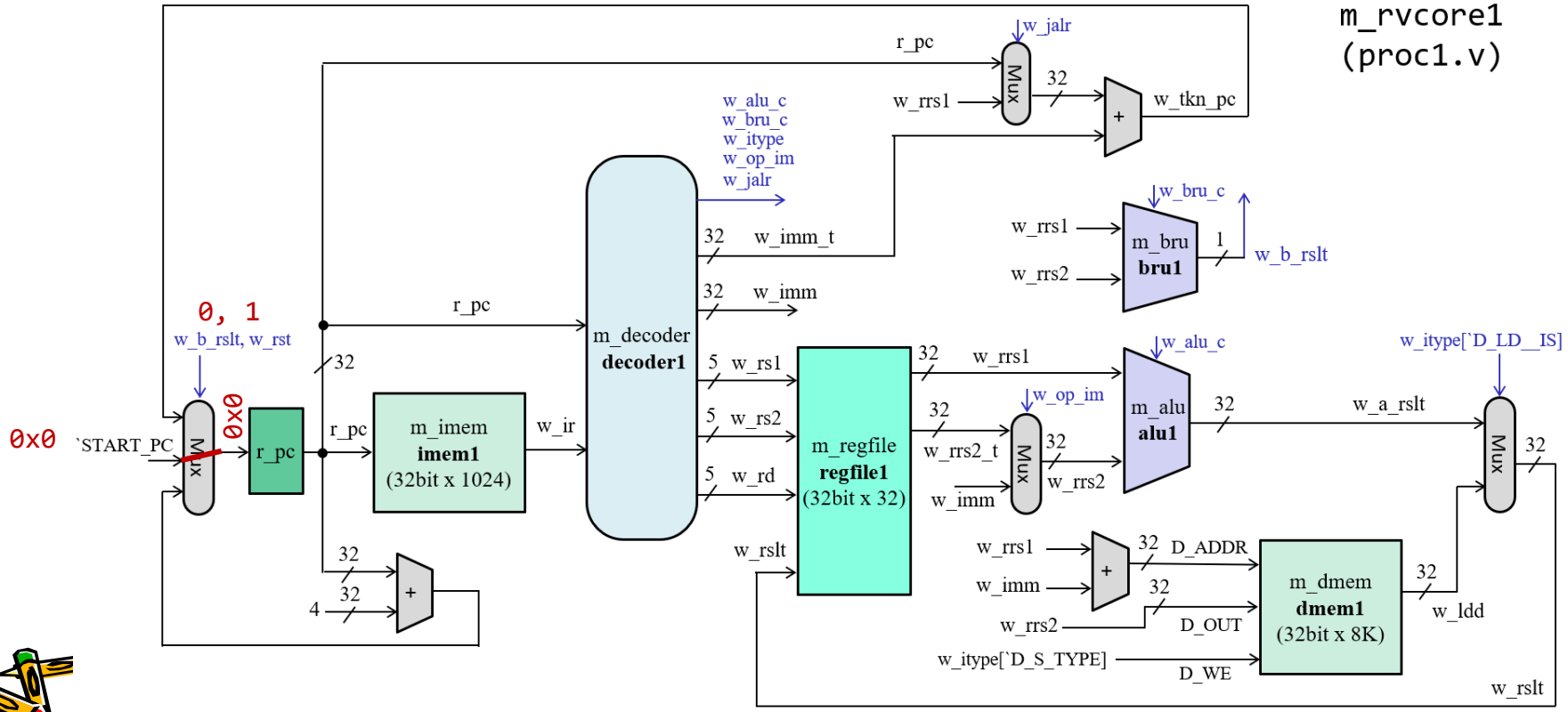
```
0x00  L1: addi x5, x0, 2      # x5 = 2
0x04      addi x6, x0, 3      # x6 = 3
0x08      add  x7, x5, x6     # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)     # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)     # x8 = mem[0 + 32]
0x14      add  x9, x8, x5     # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1     # go to L1 if x5!=x6
```

```
0x00200293
0x00300313
0x006283B3
0x02702023
0x02002403
0x005404B3
0xFE6294E3
```

Processing behabior of rvcore1

```
0x00  L1: addi x5, x0, 2      # x5 = 2
0x04      addi x6, x0, 3      # x6 = 3
0x08      add  x7, x5, x6     # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)     # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)     # x8 = mem[0 + 32]
0x14      add  x9, x8, x5     # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1     # go to L1 if x5!=x6
```

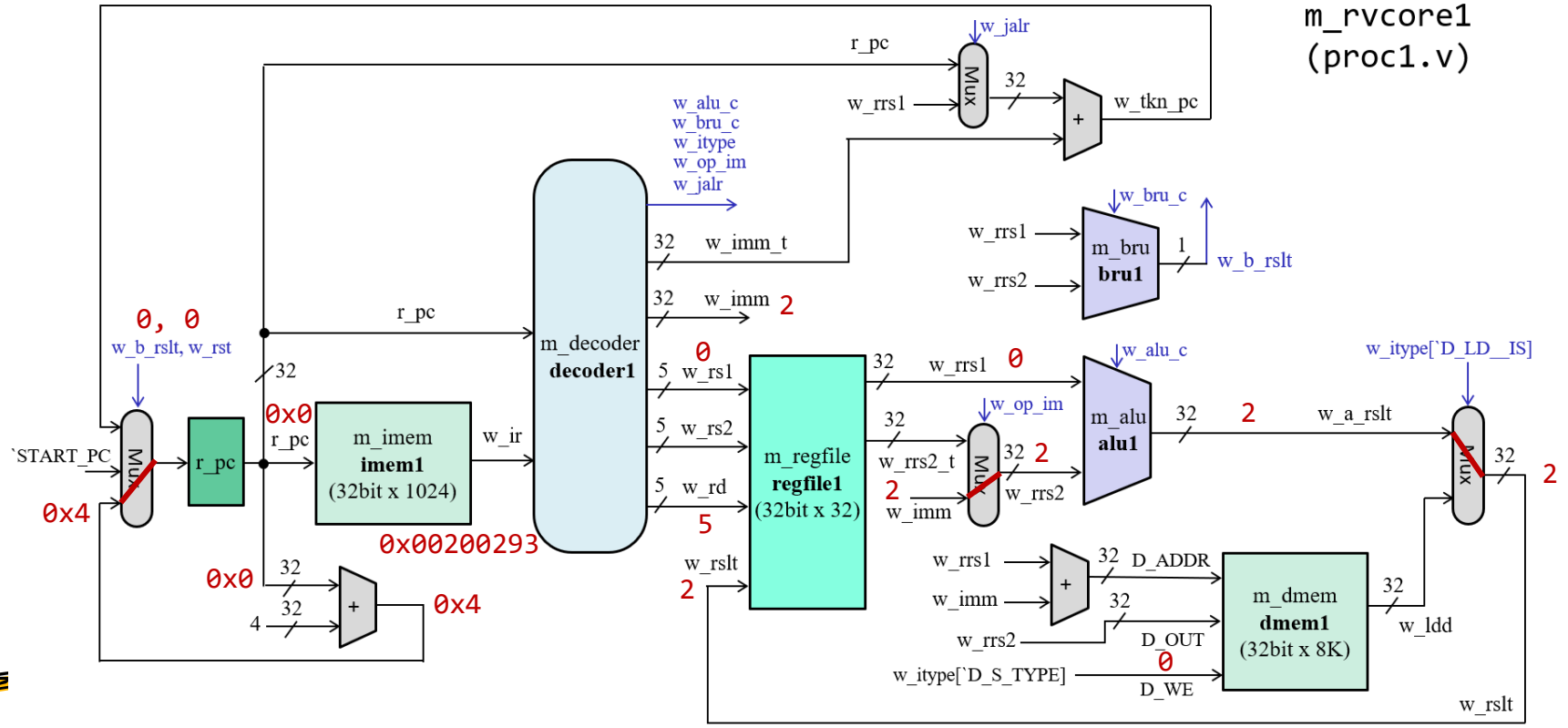
- cycle count 0 (cc0) at 50nsec



Processing behabior of rvcore1

```
0x00  L1: addi x5, x0, 2      # x5 = 2
0x04      addi x6, x0, 3      # x6 = 3
0x08      add  x7, x5, x6     # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)     # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)     # x8 = mem[0 + 32]
0x14      add  x9, x8, x5     # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1     # go to L1 if x5!=x6
```

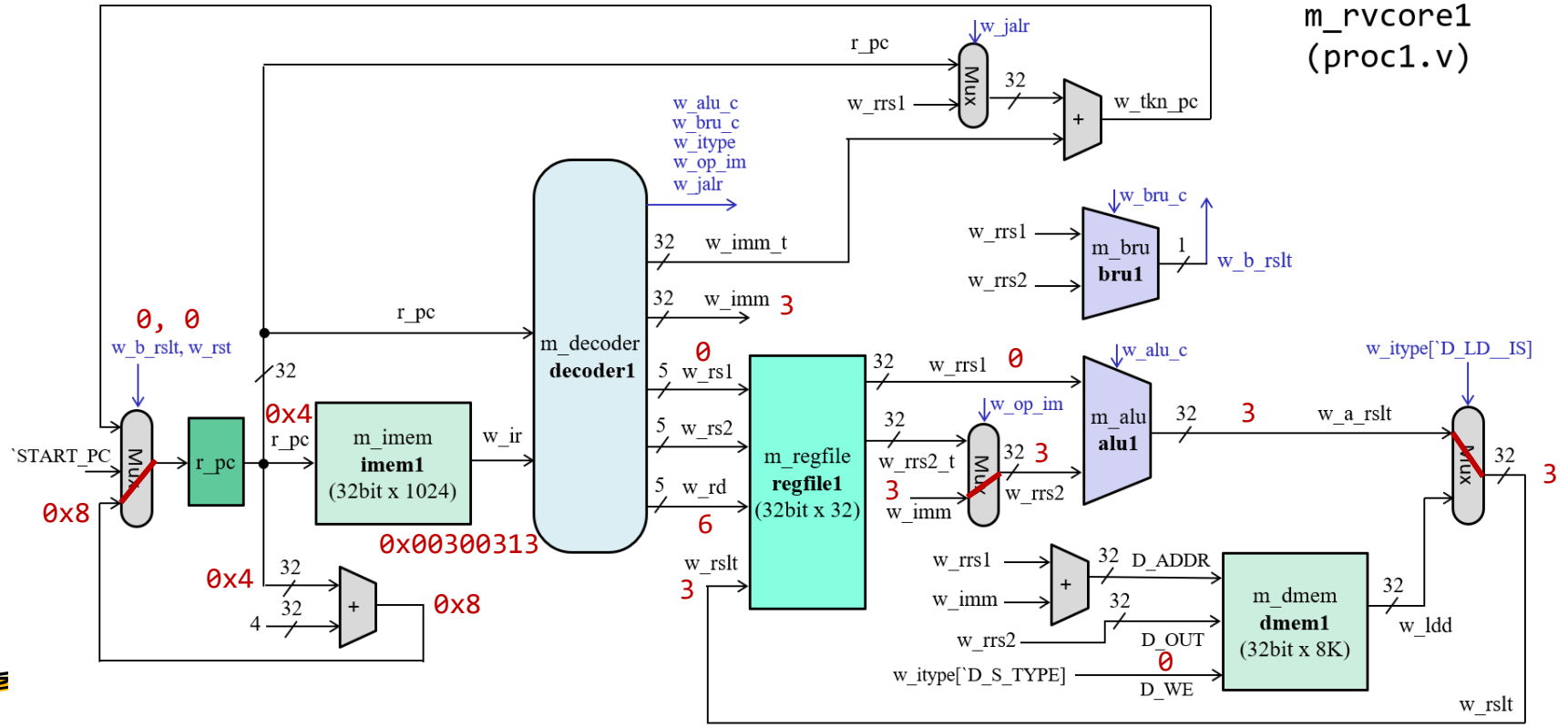
- cycle count 1 (cc1) at 150nsec
- executing
  `addi x5, x0, 2`
  of address 0x00



Processing behabior of rvcore1

```
0x00  L1: addi x5, x0, 2      # x5 = 2
0x04      addi x6, x0, 3      # x6 = 3
0x08      add  x7, x5, x6     # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)     # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)     # x8 = mem[0 + 32]
0x14      add  x9, x8, x5     # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1     # go to L1 if x5!=x6
```

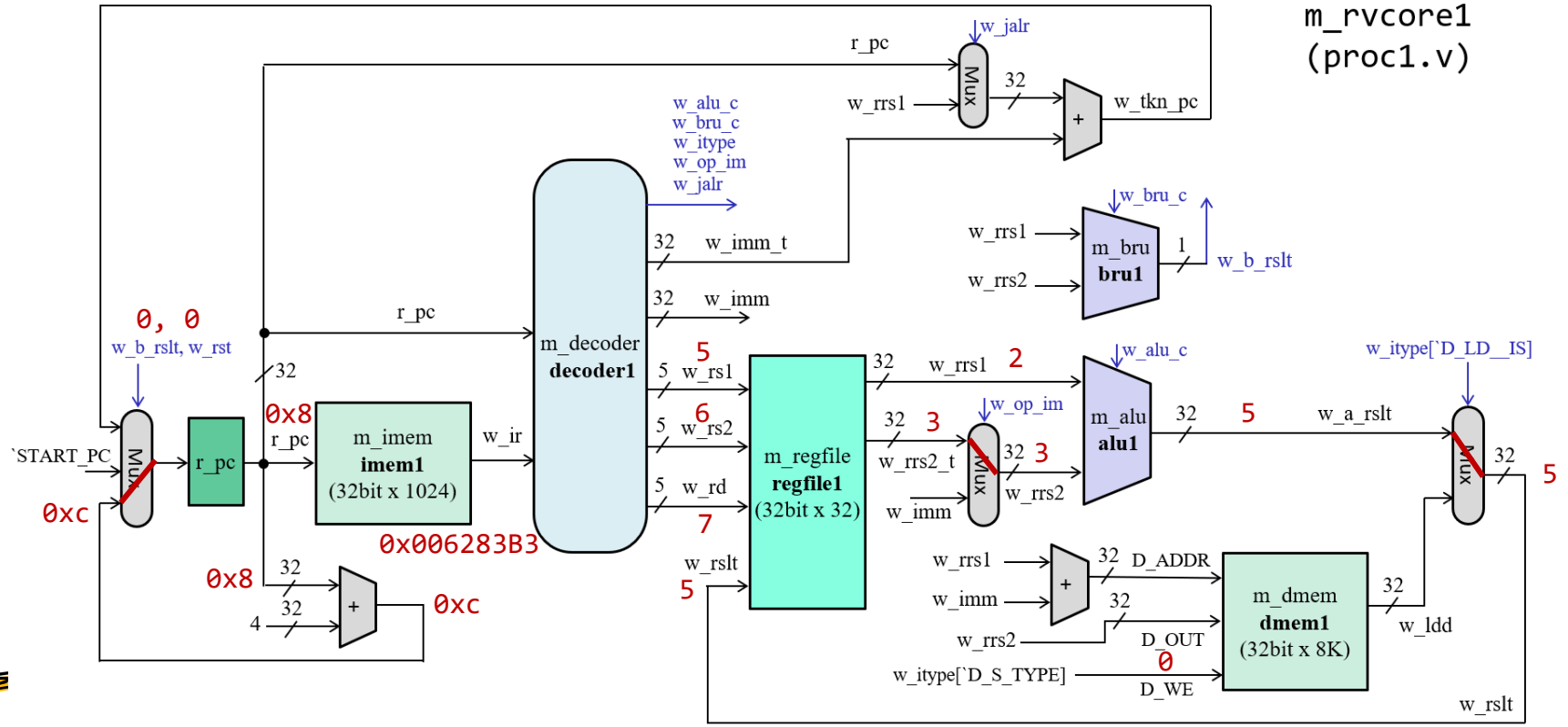
- cycle count 2 (cc2) at 250nsec
- executing
  `addi x6, x0, 3`
  of address 0x04



Processing behabior of rvcore1

```
0x00  L1: addi x5, x0, 2      # x5 = 2
0x04      addi x6, x0, 3      # x6 = 3
0x08      add  x7, x5, x6     # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)     # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)     # x8 = mem[0 + 32]
0x14      add  x9, x8, x5     # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1     # go to L1 if x5!=x6
```

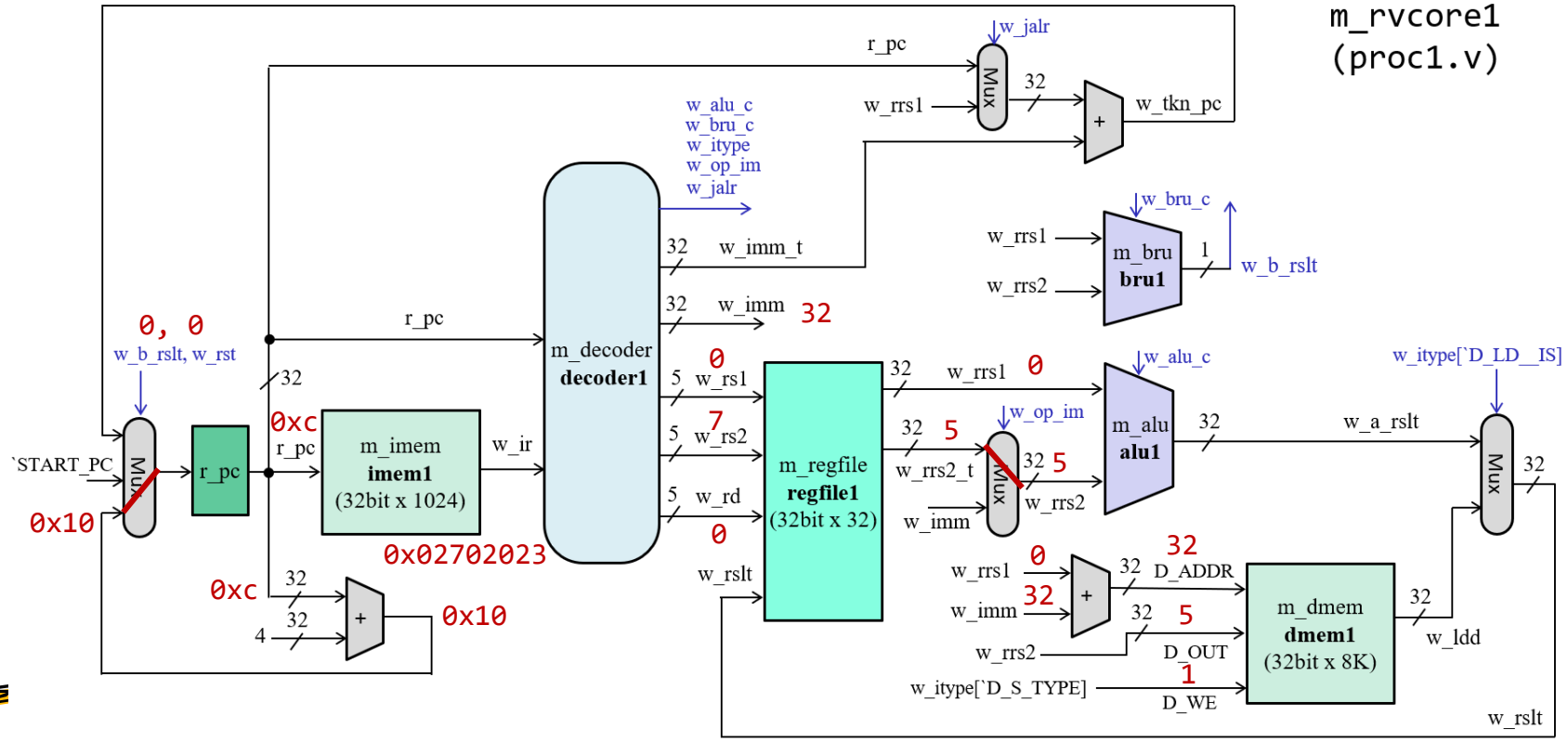
- cycle count 3 (cc3) at 350nsec
- executing
  `add x7, x5, x6`
  of address 0x08



Processing behabior of rvcore1

```
0x00  L1: addi x5, x0, 2      # x5 = 2
0x04      addi x6, x0, 3      # x6 = 3
0x08      add  x7, x5, x6     # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)     # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)     # x8 = mem[0 + 32]
0x14      add  x9, x8, x5     # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1     # go to L1 if x5!=x6
```

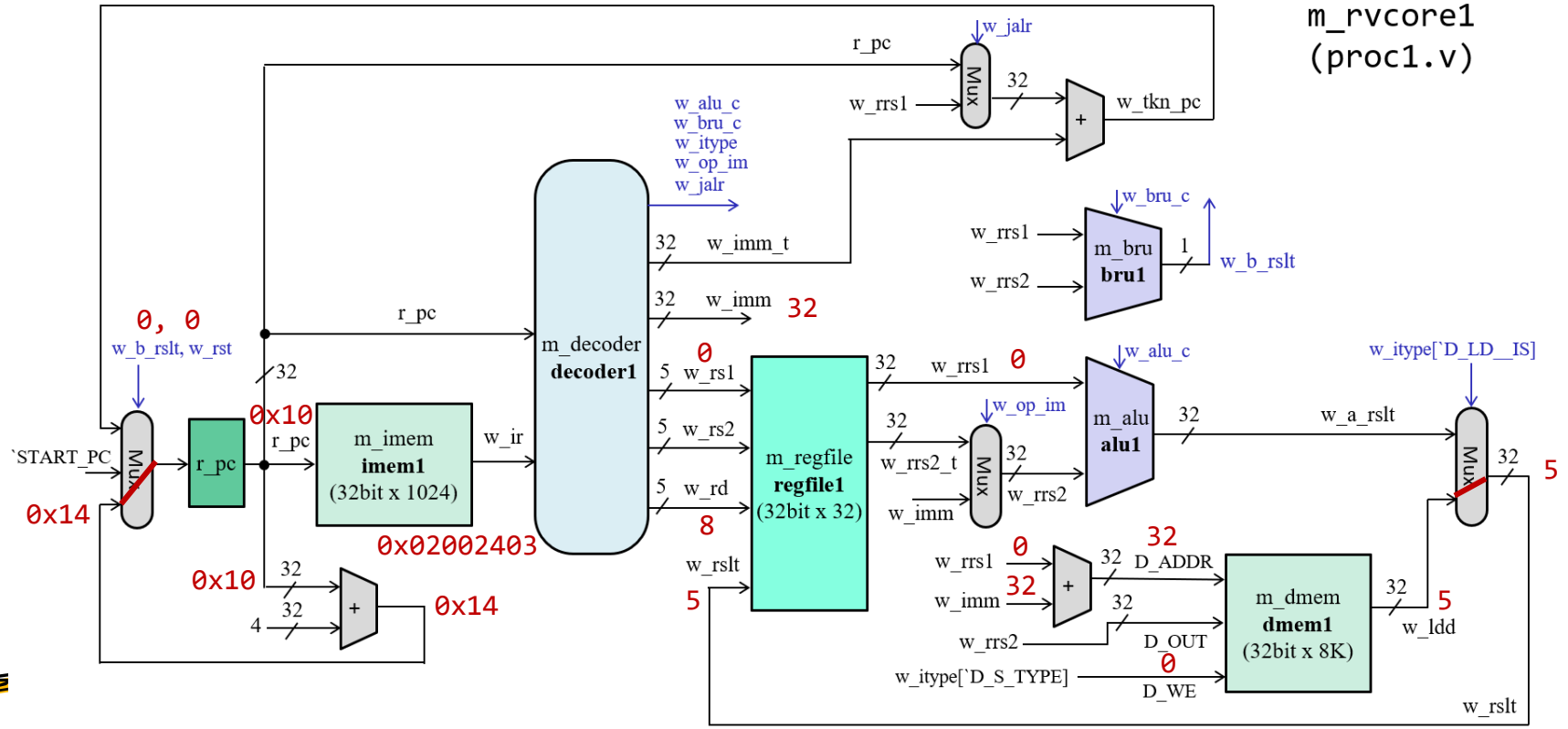
- cycle count 4 (cc4) at 450nsec
- executing
  `sw  x7, 32(x0)`
  of address 0x0c



Processing behabior of rvcore1

```
0x00  L1: addi x5, x0, 2      # x5 = 2
0x04      addi x6, x0, 3      # x6 = 3
0x08      add  x7, x5, x6     # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)     # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)     # x8 = mem[0 + 32]
0x14      add  x9, x8, x5     # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1     # go to L1 if x5!=x6
```

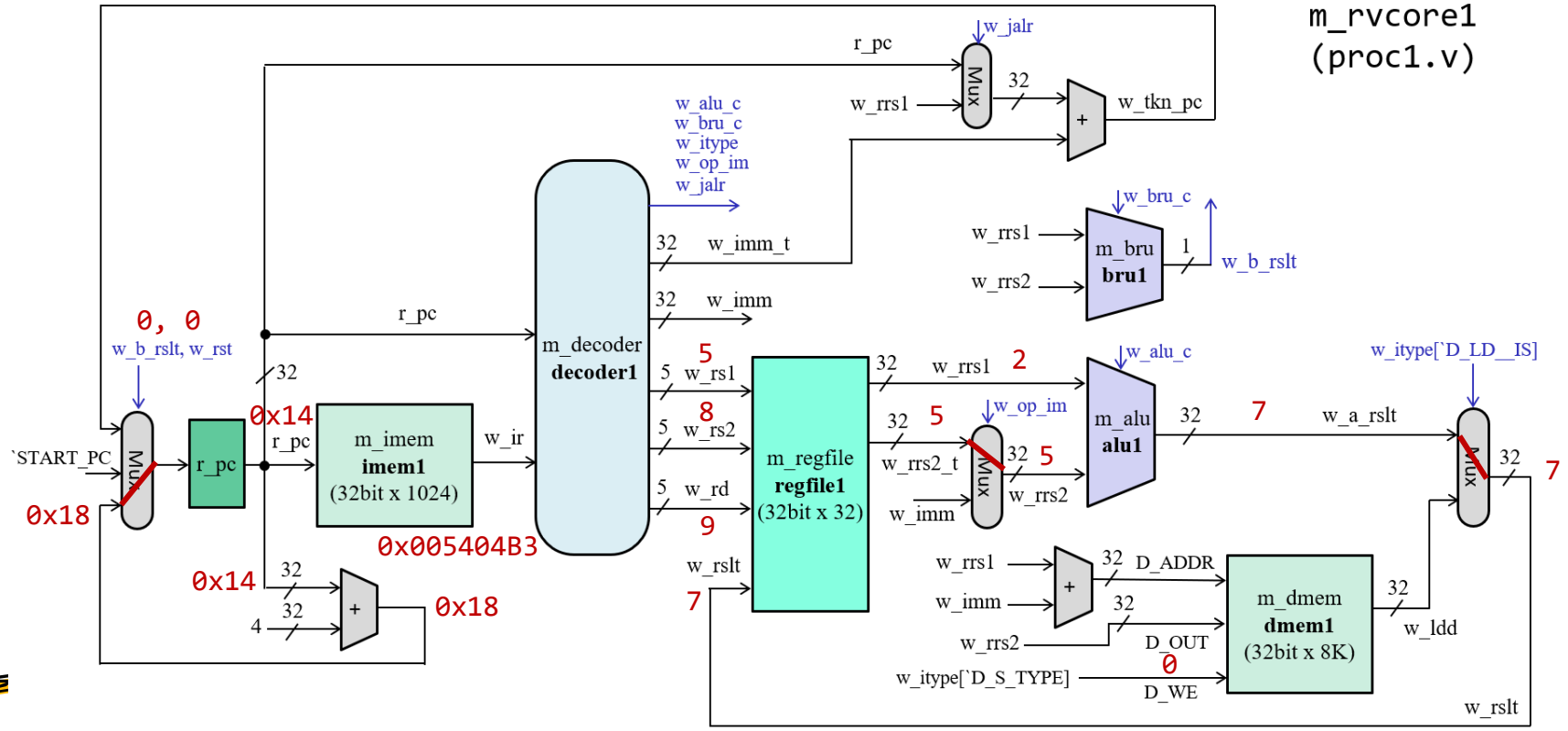
- cycle count 5 (cc5) at 550nsec
- executing
  `lw x8, 32(x0)`
  of address 0x10



Processing behabior of rvcore1

```
0x00  L1: addi x5, x0, 2      # x5 = 2
0x04      addi x6, x0, 3      # x6 = 3
0x08      add  x7, x5, x6     # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)     # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)     # x8 = mem[0 + 32]
0x14      add  x9, x8, x5     # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1     # go to L1 if x5!=x6
```

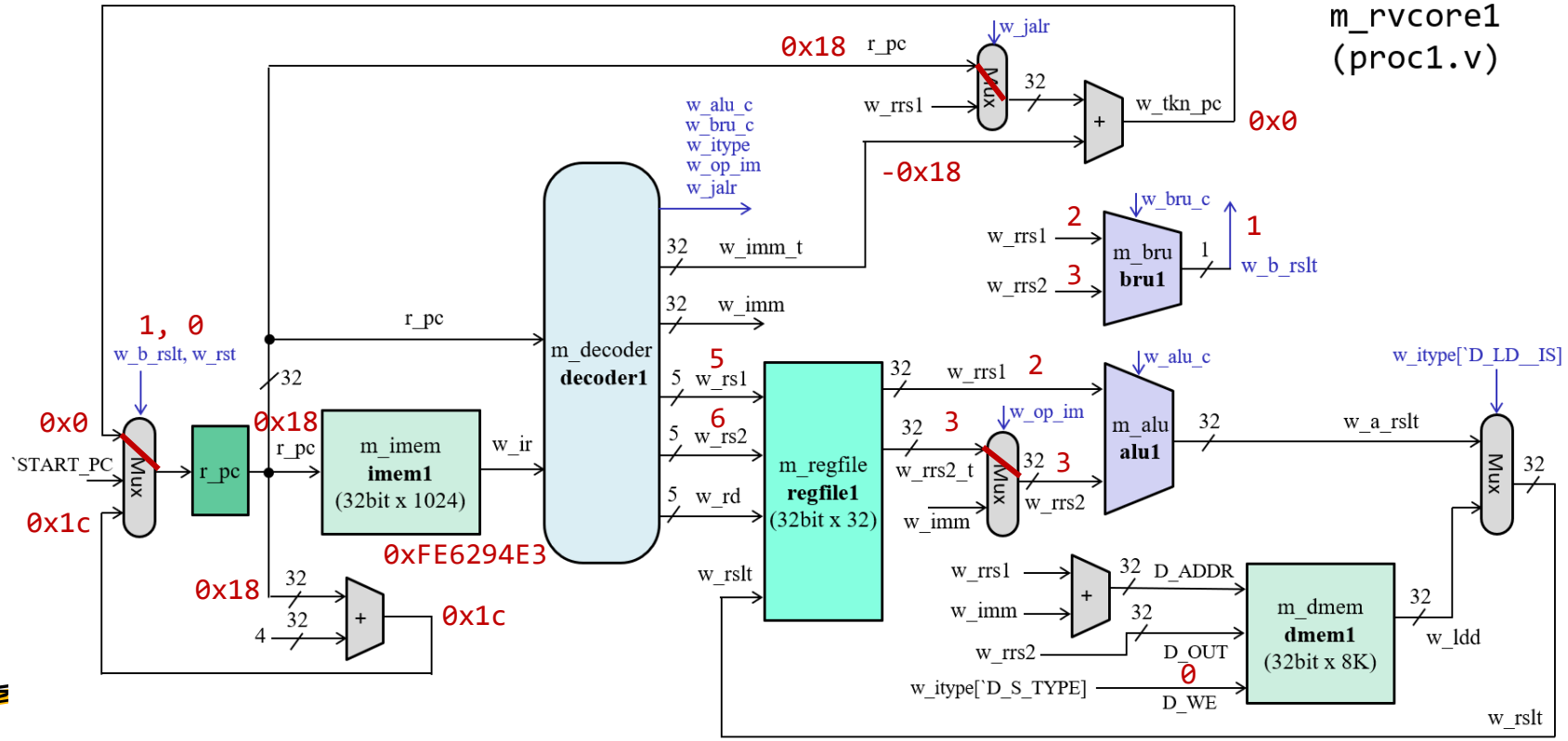
- cycle count 6 (cc6) at 650nsec
- executing
  `add x9, x8, x5`
  of address 0x14



Processing behabior of rvcore1

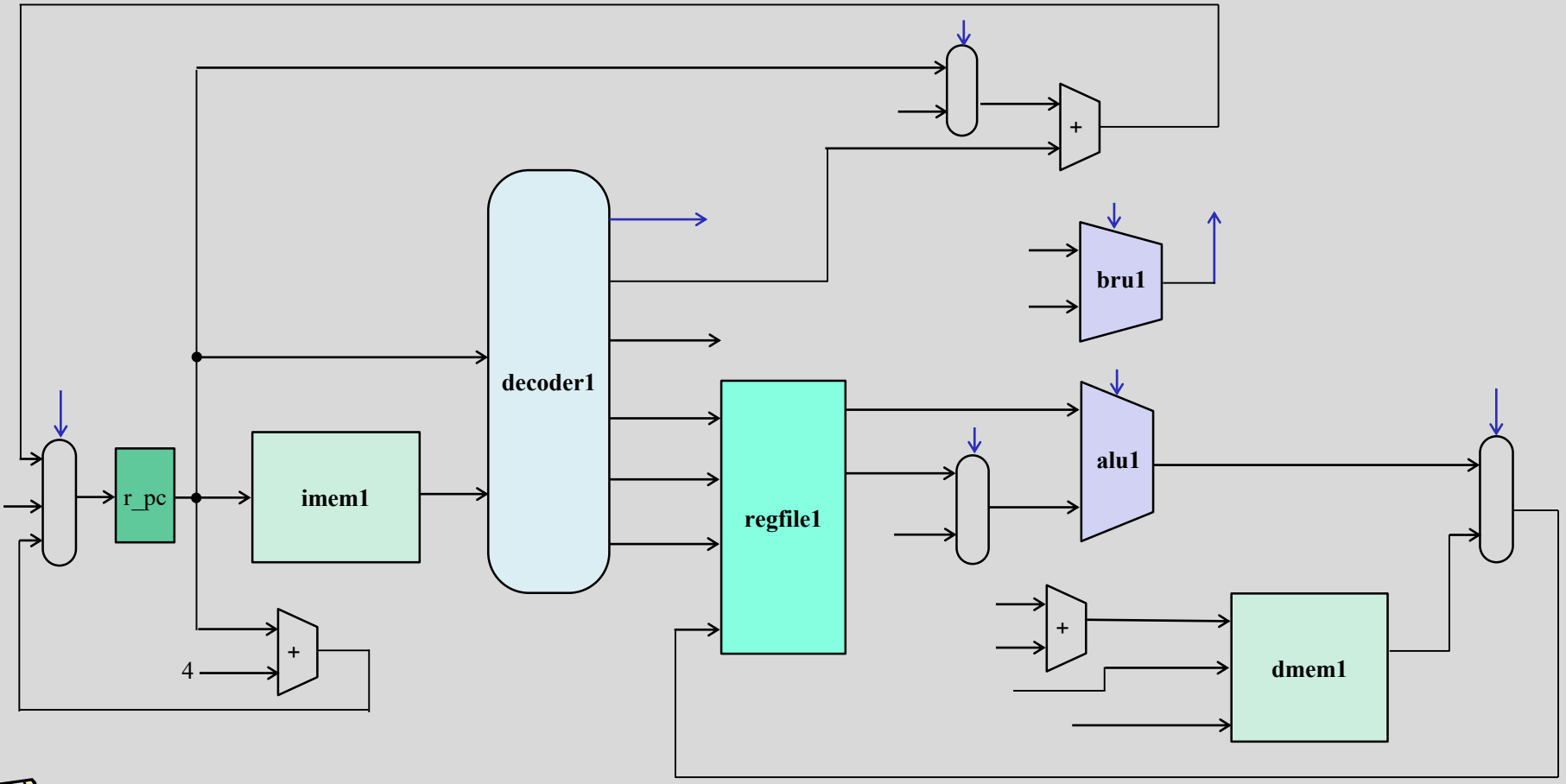
```
0x00  L1: addi x5, x0, 2      # x5 = 2
0x04      addi x6, x0, 3      # x6 = 3
0x08      add  x7, x5, x6     # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)     # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)     # x8 = mem[0 + 32]
0x14      add  x9, x8, x5     # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1     # go to L1 if x5!=x6
```

- cycle count 7 (cc7) at 750nsec
- executing
  bne x5, x6, L1
  of address 0x18



Exercise 1

- Draw the block diagram of pvcore1 and write the line number of source code where that hardware is described.



m_rvcore (RV32I, single-cycle processor)

```
21  /***** subset of RV32I where LB, LH, LBU, LHU, SB, SH are not supported              *****/
22  /*************************************************************************************/
23  module m_rvcore ( ///// RVCore Simple Version
24      input  wire        w_clk,  // clock signal
25      input  wire        w_rst,  // reset signal
26      output wire [31:0] D_ADDR, // data memory, address
27      output wire [31:0] D_OUT,  // data memory, output data
28      output wire        D_WE    // data memory, write enable
29  );
30      reg  [31:0] r_pc;
31      wire        w_jalr, w_op_im, w_b_rslt;
32      wire [9:0]  w_itype;
33      wire [10:0] w_alu_c;
34      wire [6:0]  w_bru_c;
35      wire [4:0]  w_rs1, w_rs2, w_rd;
36      wire [31:0] w_ir, w_rrs1, w_rrs2_t, w_rrs2, w_imm_t, w_imm;
37      wire [31:0] w_a_rslt, w_rslt, w_tkn_pc, w_ldd;
38
39      m_imem imem1 (w_clk, r_pc, w_ir);
40
41      m_decoder decoder1 (r_pc, w_ir, w_rd, w_rs1, w_rs2,
42                          w_op_im, w_itype, w_jalr, w_alu_c, w_bru_c, w_imm_t, w_imm);
43
44      m_regfile regfile1 (w_clk, w_rs1, w_rs2, w_rrs1, w_rrs2_t, w_rd, w_rslt);
45      assign w_rrs2 = (w_op_im) ? w_imm : w_rrs2_t;
46
47      m_alu alu1 (w_rrs1, w_rrs2, w_alu_c, w_a_rslt);
48      m_bru bru0 (w_rrs1, w_rrs2, w_bru_c, w_b_rslt);
49
50      assign D_ADDR = w_rrs1 + w_imm;
51      assign D_OUT  = w_rrs2;
52      assign D_WE   = w_itype[`D_S_TYPE];
53      m_dmem dmem1 (w_clk, D_WE, D_ADDR, D_OUT, w_ldd);
54
55      assign w_rslt = (w_itype[`D_LD__IS]) ? w_ldd : w_a_rslt;
56
57      assign w_tkn_pc = ((w_jalr) ? w_rrs1 : r_pc) + w_imm_t;
58      always @(posedge w_clk) r_pc <= (w_rst) ? `START_PC : (w_b_rslt) ? w_tkn_pc : r_pc + 4;
59  endmodule
```

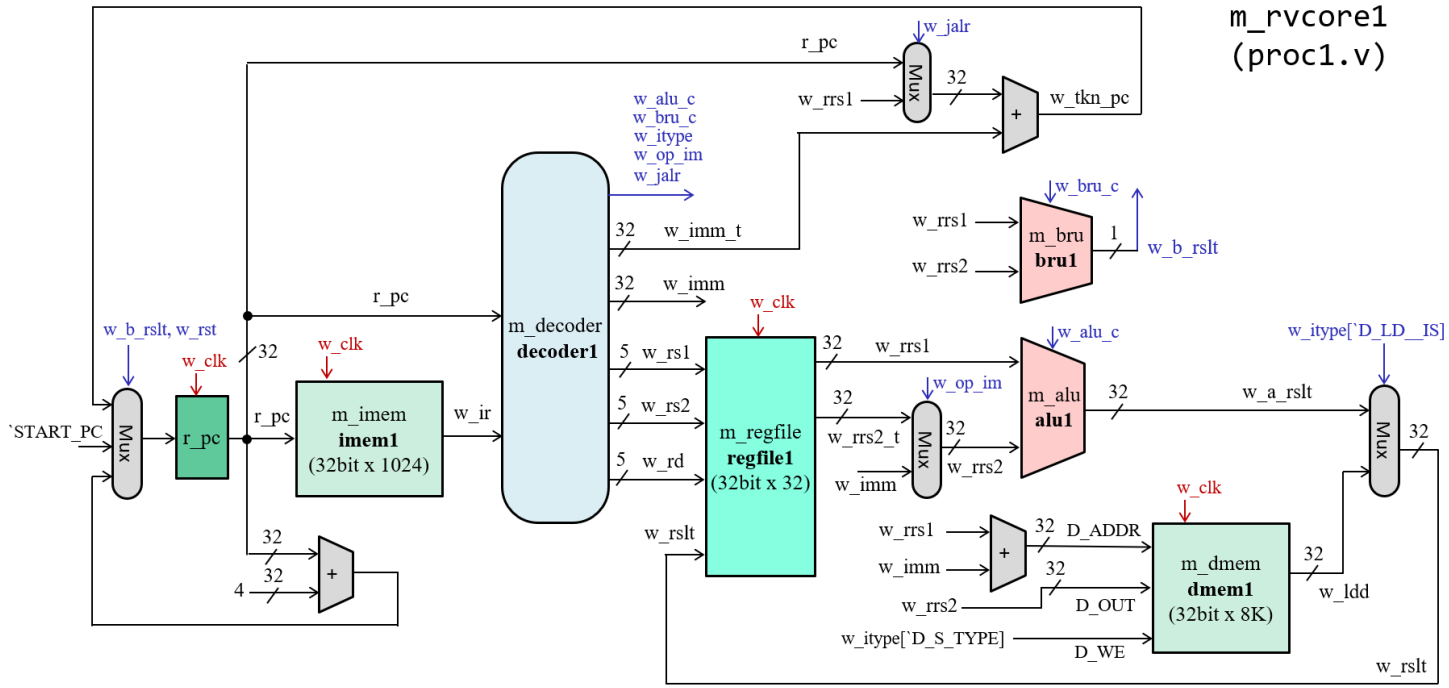
Simulation in the ACRi room environment

```
$ cd
$ mkdir aca
$ cd aca
$ cp /home/tu_kise/aca/circuit1.v .
$ iverilog circuit1.v
$ ./a.out

$ /usr/bin/gtkwave dump.vcd
```

```
$ cd
$ cd aca
$ cp -r /home/tu_kise/aca/rvcore1 .
$ cd rvcore1
$ make
$ make run
```

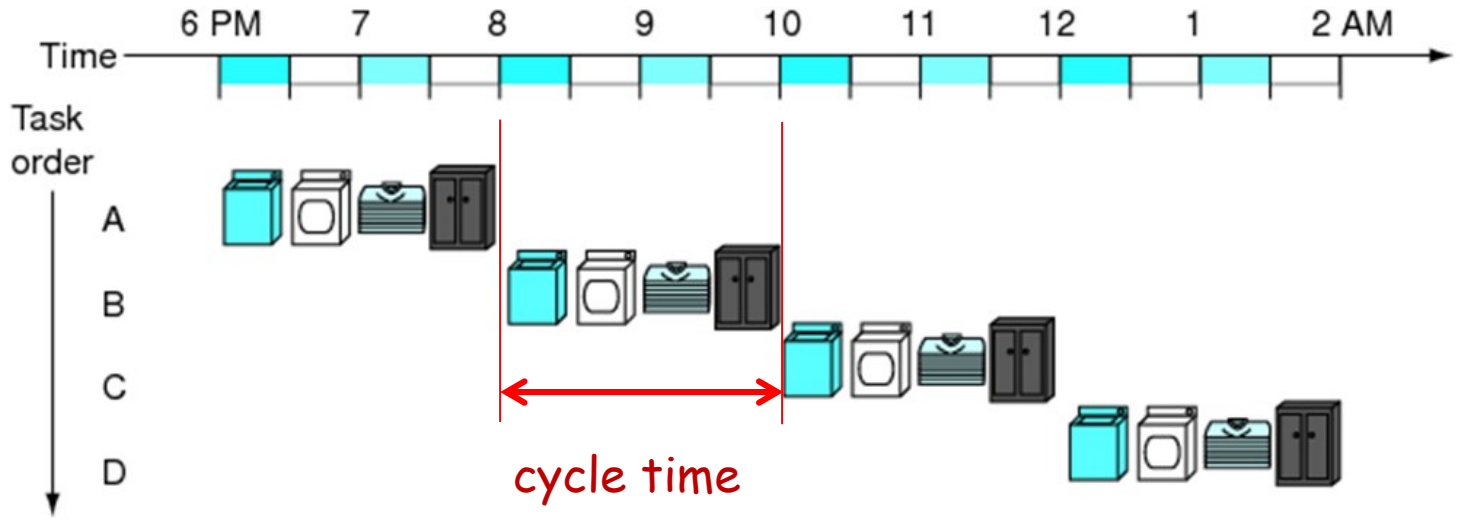
Single-cycle implementation of processors

- 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.
  It is useful as a baseline for lectures.



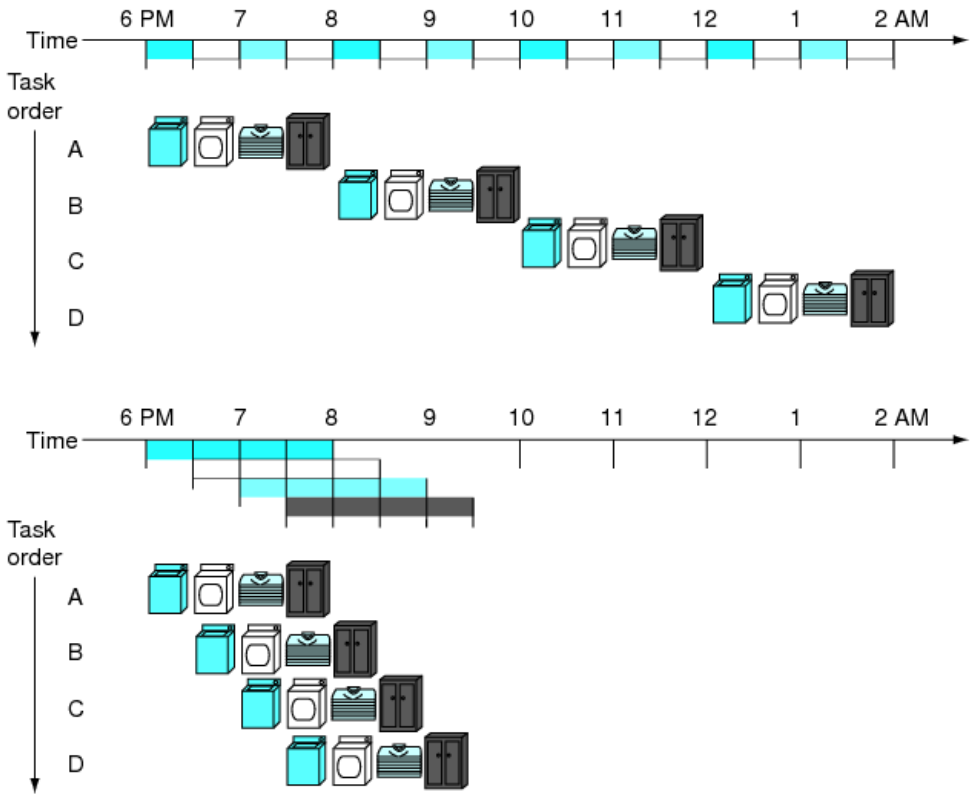
Single-cycle implementation of laundry

- (A) Ann, (B) Brian, (C) Cathy, and (D) Don each have dirty clothes to be *washed*, *dried*, *folded*, and *put away*, each taking 30 minutes.
- The cycle time (the time from the end of one load to the end of the next one) is 2 hours.
- For four loads, the sequential laundry takes 8 hours.



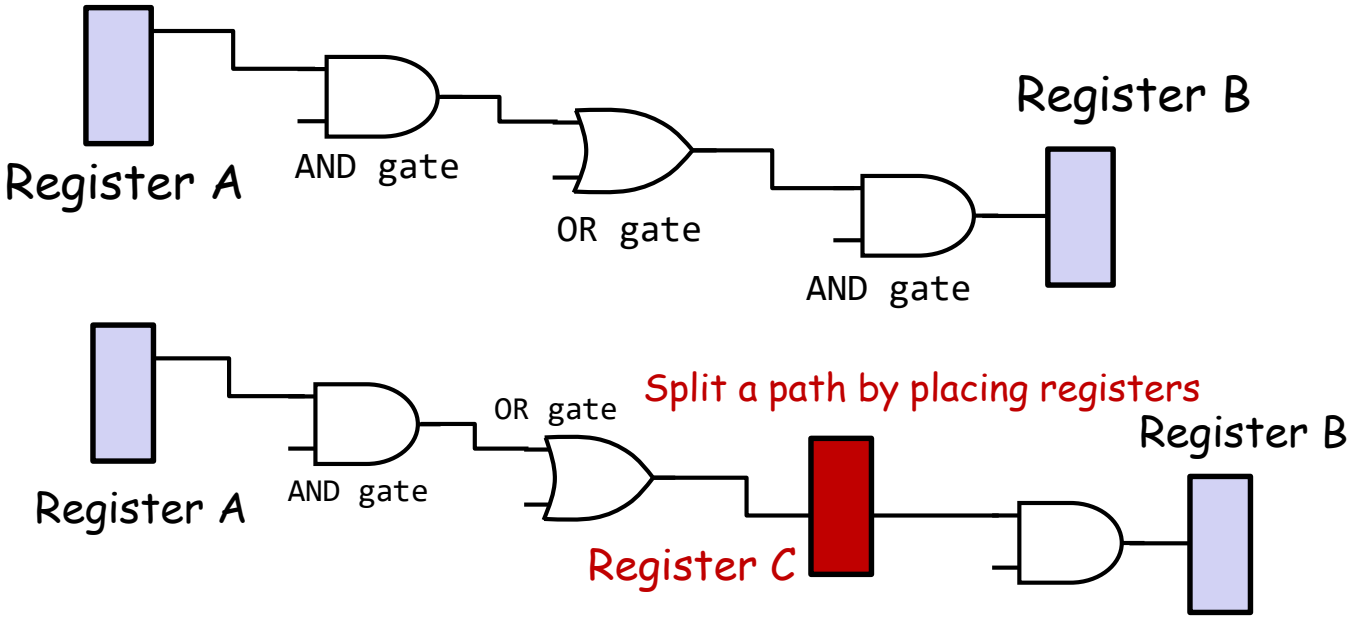
Single-cycle implementation and pipelining

- When the washing of load A is finished at 6:30 p.m., another washing of load B starts.
- Pipelined laundry takes 3.5 hours just using the same hardware resources. The cycle time is 30 minutes.
- What is the latency (execution time) of each load?



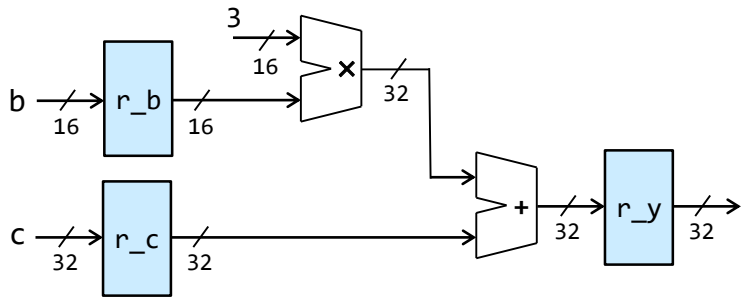
Clock rate is mainly determined by

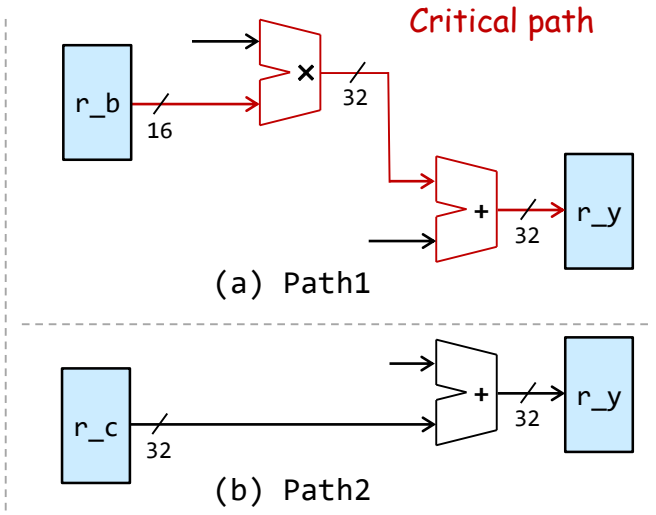
- Switching speed of gates (transistors)
- The number of levels of gates
  - The maximum number of gates cascaded in series in any combinational logics.
  - In this example, the number of levels of gates is 3.
- Wiring delay and fanout



Pipelining example: multiply-add operation (1)

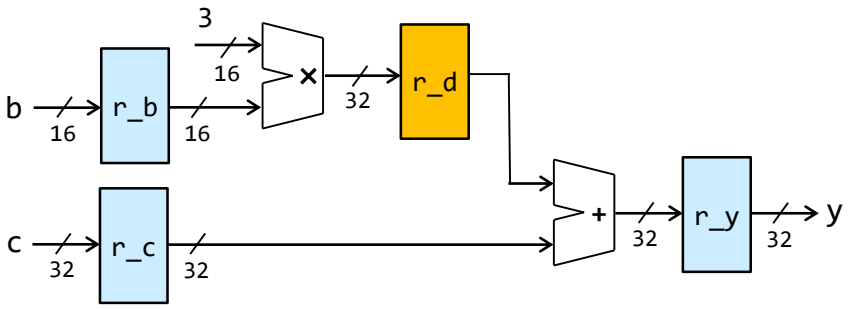
- As an example of pipelining, we will see a multiply-add circuit.
- r_b, r_c are input registers and r_y is output register of the circuit.
- This has two paths named path1 and path2, and path1 is the critical path to determine the maximum operating frequency.

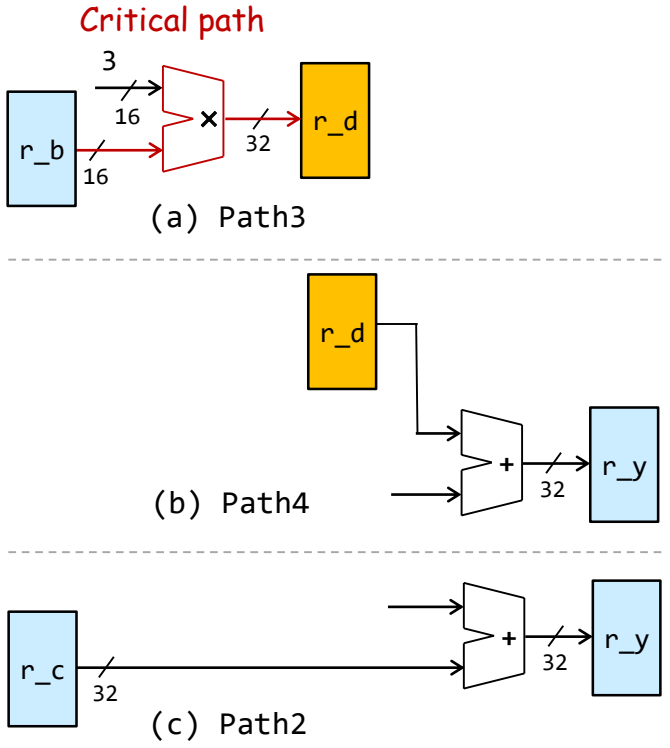




Pipelining example: multiply-add operation (2)

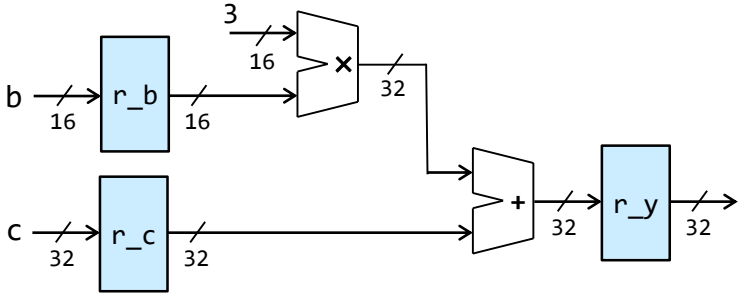
- By inserting register r_d, the critical path can be divided into Path3 and Path4.
- As a result, the new critical path becomes Path3.
- This has the disadvantage that input b and c in the same clock cycle cannot be processed.



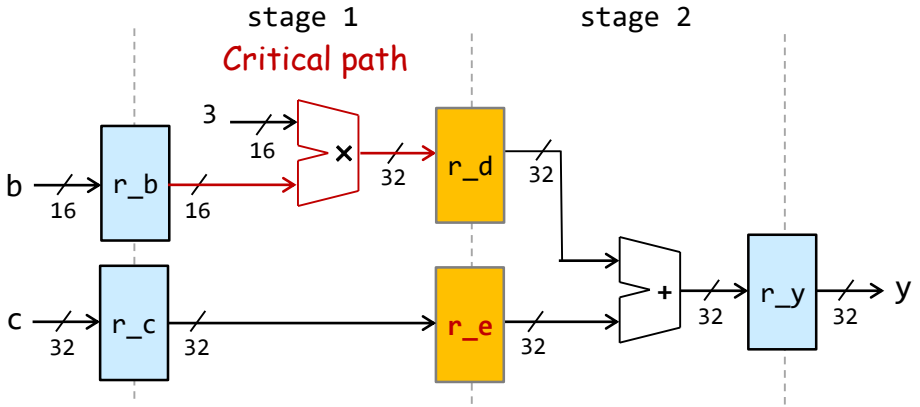


Pipelining example: multiply-add operation (3)

- To overcome this drawback, we insert register r_e.
- This realizes a pipeline with stages 1 and 2.
  A set of registers between two adjacent stages are called a pipeline register.



(a) original multiply-add circuit



(b) two-stage pipelined circuit