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

Ver. 2023-04-27a

# 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

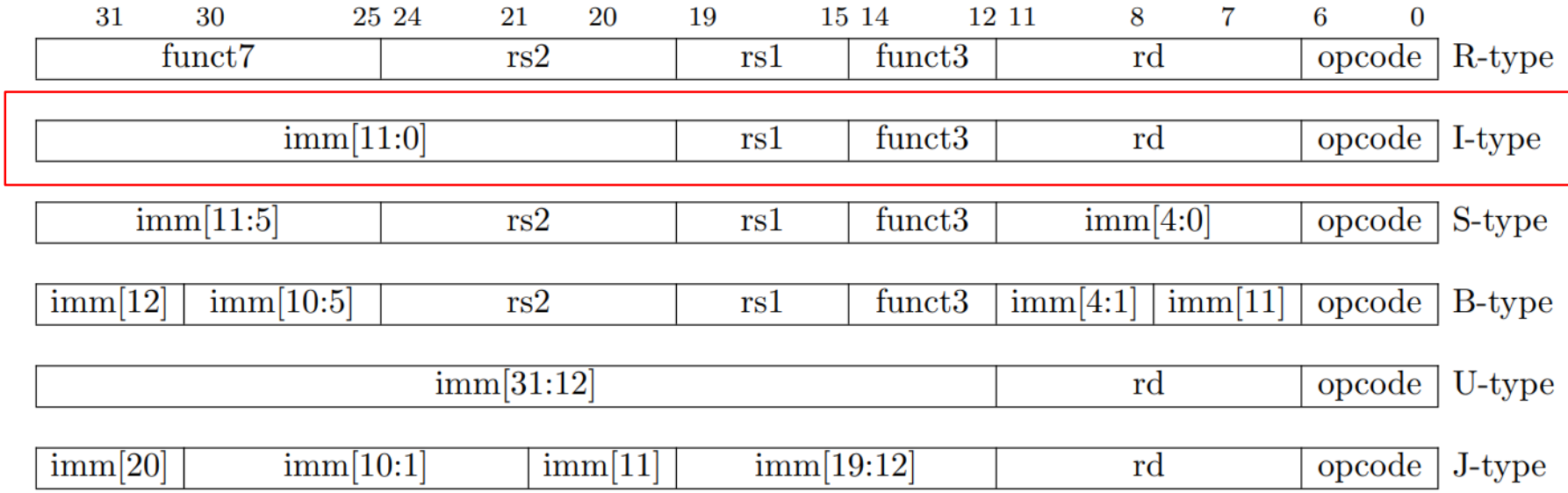



Figure 2.3: RISC-V base instruction formats showing immediate variants.

# 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)
```

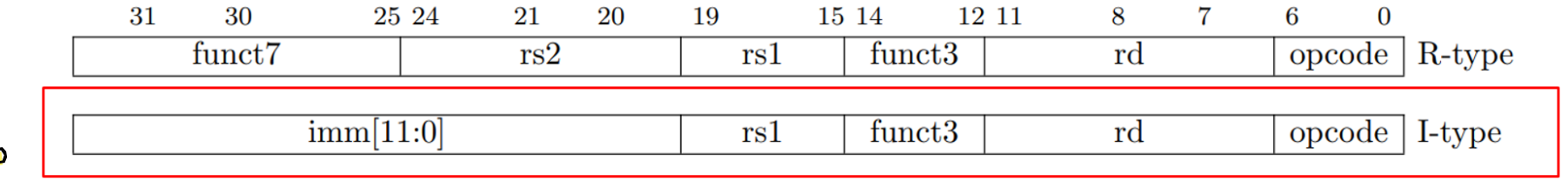
- Machine format (I format):


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

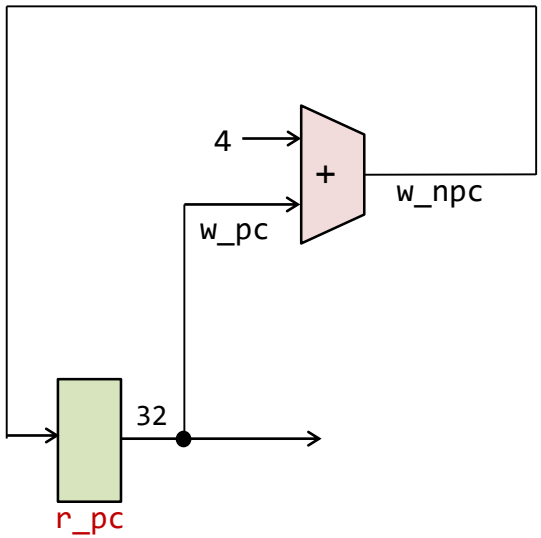
```
reg signed [31:0] x [0:31];  // signed registers
wire       [31:0] w_ir;      // instruction
wire signed [31:0] w_sext_imm   = {{20{w_ir[31]}}, w_ir[31:20]};
wire       [31:0] w_sext_imm_u = {{20{w_ir[31]}}, w_ir[31:20]};

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.

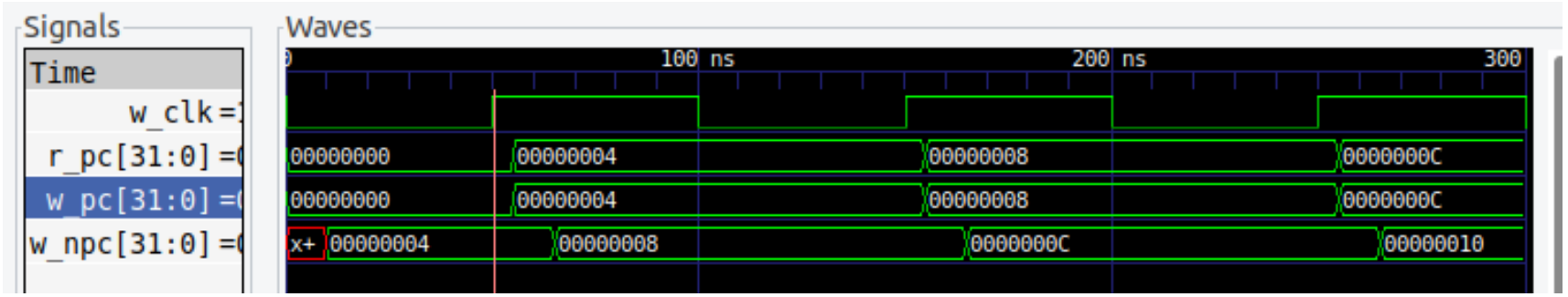
# m_proc01 プロセッサの設計と実装に向けた一歩




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
```



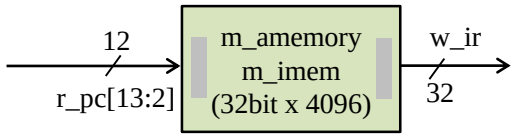
# m_amemory 非同期式メモリの記述とシミュレーション

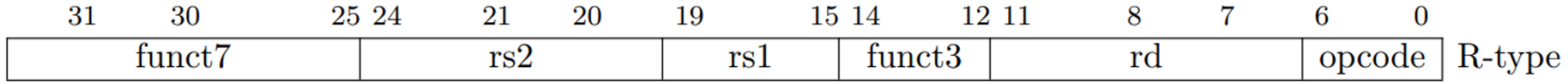
- Verilog HDLでは，ビット幅Bでワード数Wのメモリ m を reg [B-1:0] m [0:W-1] として宣言できる．
- 読み出す動作でクロック信号を利用しないメモリを非同期メモリ (asynchronous memory) と呼ぶ．
- 非同期式メモリの記述例を示す．シミュレーションでの読み出しの遅延を 20nsec とした．w_addr で指定されたアドレスの内容を読み出す．posedge w_clk のタイミングで，w_we (write enable) が1の時に，w_addr で指定されたアドレスに w_din (data in) の値を書き込む．
- このコードを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

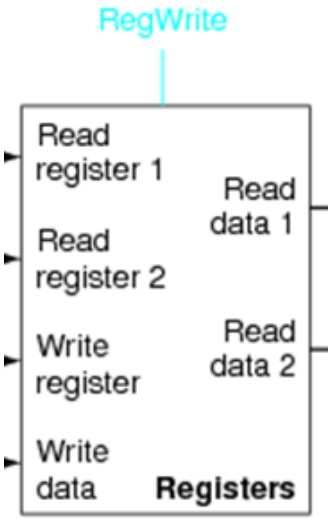
code111.v



# 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．

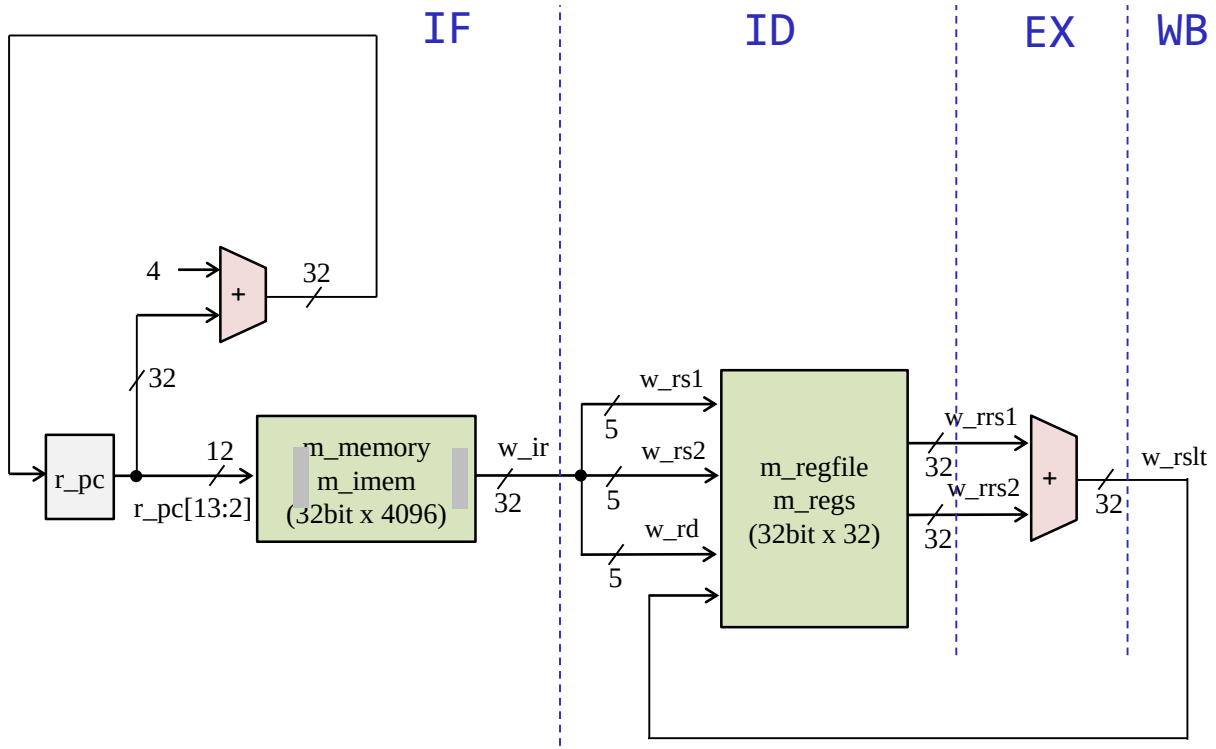


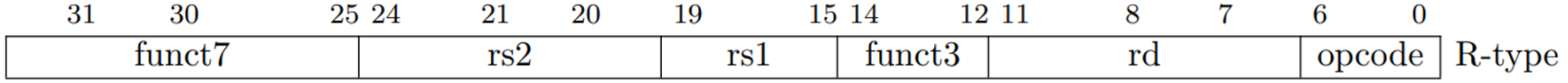

code112.v

```
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

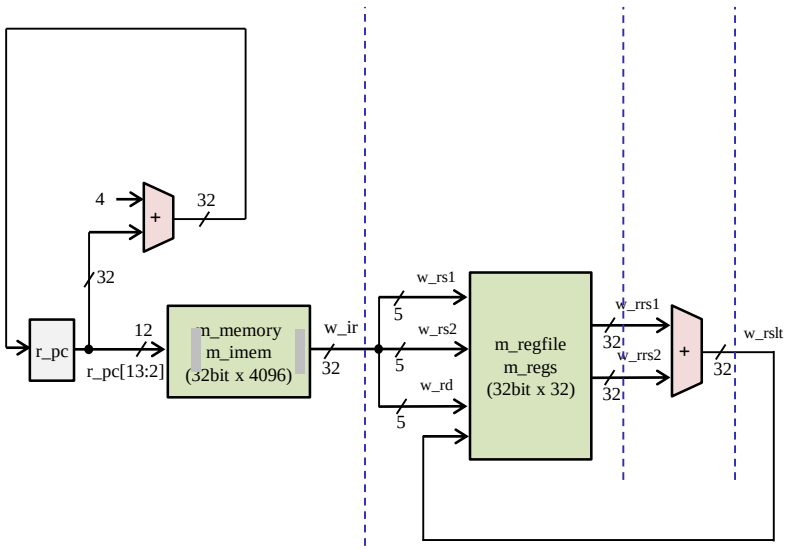
- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), ライトバック(WB) の処理をおこなう加算命令 (add) のみに対応したプロセッサのブロック図



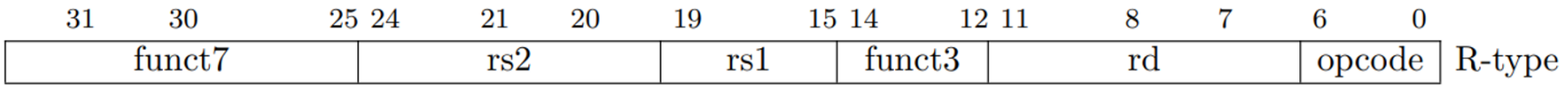
# m_proc02 addを処理するシングルサイクルのプロセッサ

- 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
```

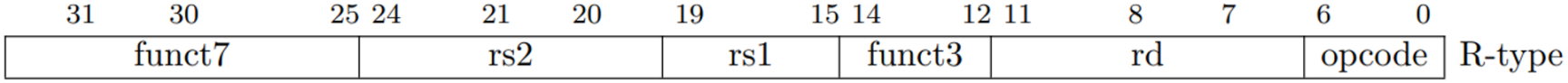


# m_proc02 addを処理するシングルサイクルのプロセッサ

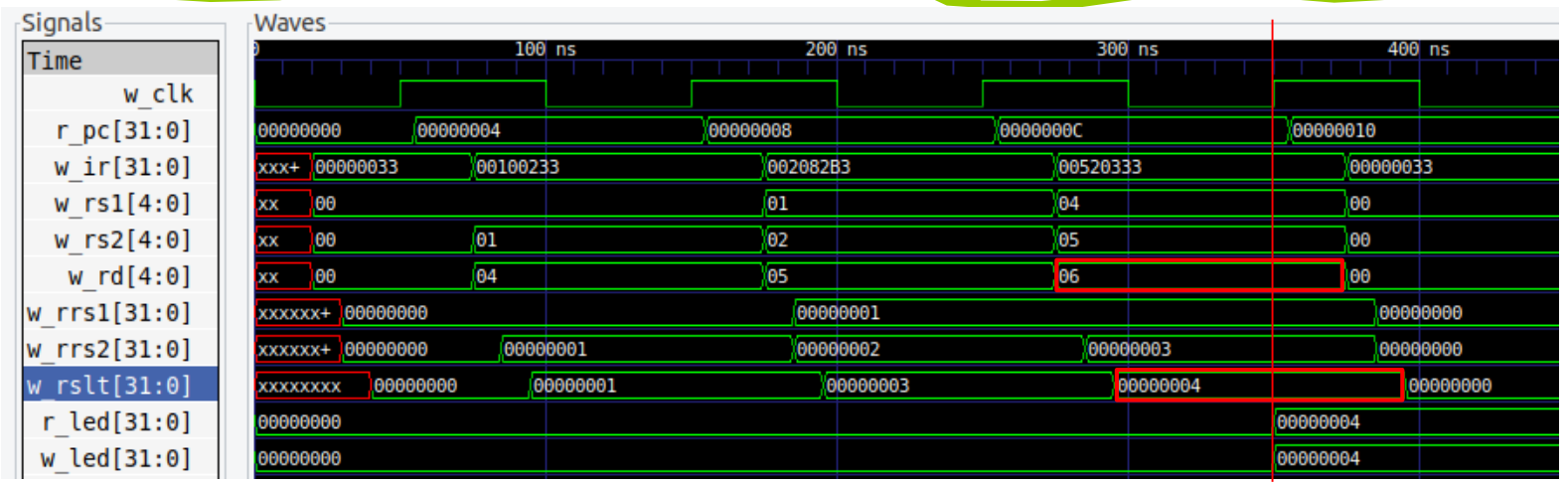
- code113.v をシミュレーションして，その波形を確認すること.
- 命令フェッチ(IF), デコードとオペランドフェッチ(ID), 実行(EX), ライトバック(WB) の処理をおこなう加算命令 (add) のみに対応したプロセッサ
- m_proc02 のインスタンス名を p とする. p の内部の r_pc は，ピリオドを用いて p.r_pc として参照できる.
- 同様に，p に含まれるインスタンス m_reg の内部の r[1] は， p.m_reg.r[1] として参照できる。

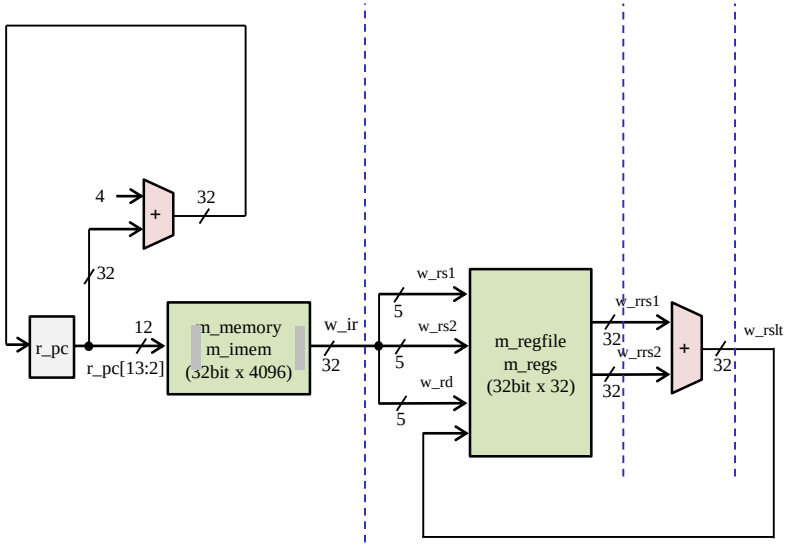
code113.v の一部

```
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
```

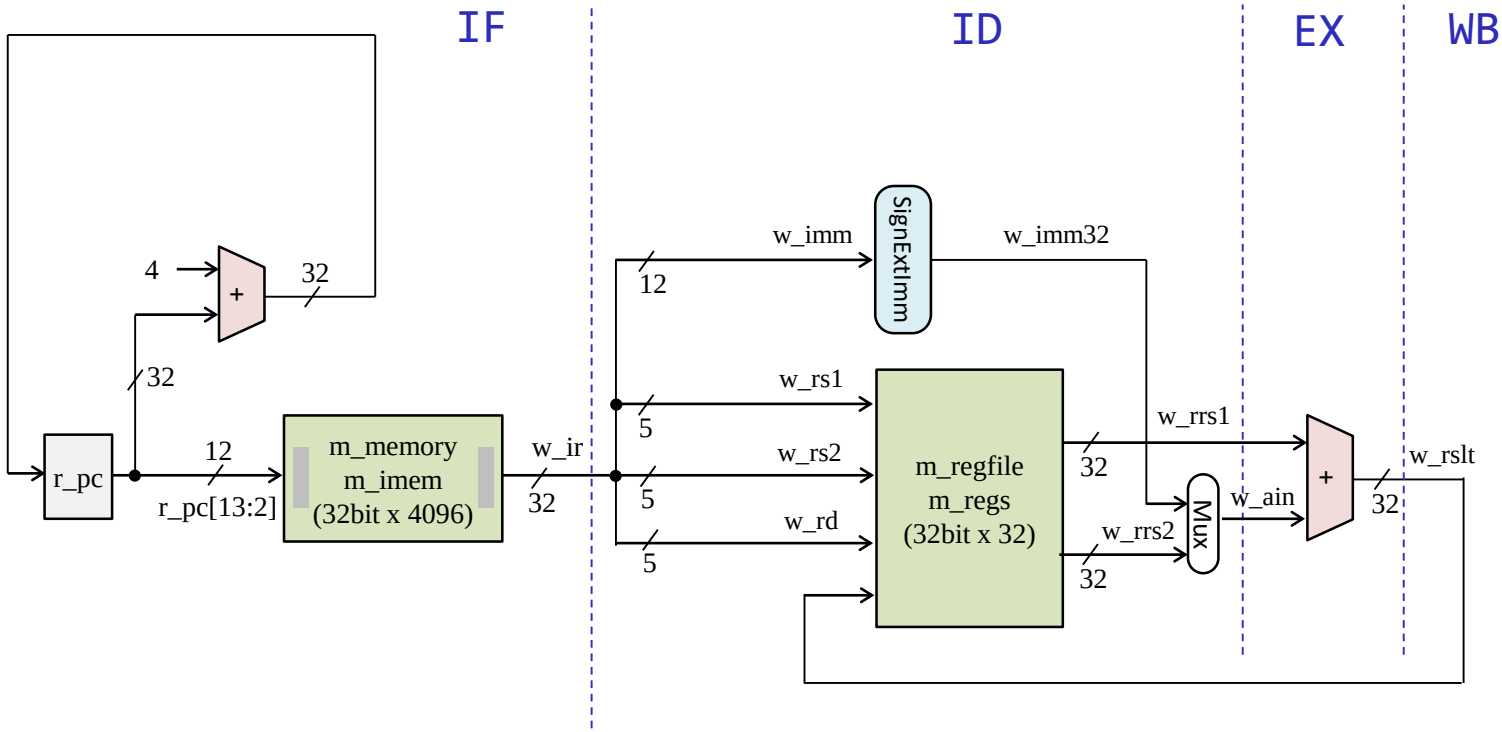
```
module m_proc02 (w_clk, w_ce, w_led);
  input  wire w_clk, w_ce;
  output wire [31:0] w_led;

  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
```

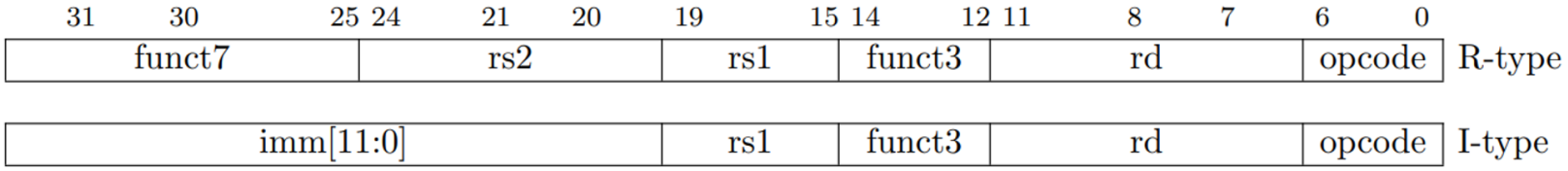
w_led の出力の値はどうなるか？

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



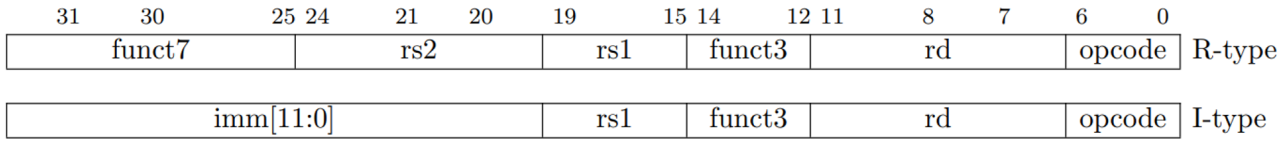
# m_proc03 add と addi を処理するプロセッサ

- 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



# 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