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/
W621 講義室　　月 10:45-12:15, 木 9:00-12:15

アナウンス

- 次回の演習(5) 2018年5月10日（木）までに，演習(1)～演習(4)の内容を終わらせること.
- 次回の演習(5)からは，プロセッサをターゲットとする演習.

Inside code056.v

- ディジタル時計の秒を2つの seven-segment LED に点滅させる回路を記述し，合成し，FPGAにコンフィギュレーションする.
  - 1秒間隔で 00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, … 58, 59, 00, 01, … と表示.
- 正しい回路が完成したら，担当のTAを呼んで確認してもらう(Check Point 3).

code056.v

```
module m_main (w_clk, r_sg, r_an);
  input  wire w_clk;
  output reg [6:0] r_sg;  // cathode segments
  output reg [7:0] r_an;  // common anode


  // do it yourself
  // Please use m_7segled in code019.v




endmodule
```

Hint for Check Point 3

- サイクル毎に 00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, … 58, 59, 00, 01, … と更新するモジュール.

code057.v

```
module m_top ();
  reg r_clk=0; initial forever #2 r_clk = ~r_clk;
  m_main m_main0 (r_clk);
  always@(posedge r_clk) #1 $write("%3d %d %d¥n", $time, m_main0.r_dig2, m_main0.r_dig1);
endmodule

module m_main (w_clk);
  input  wire w_clk;
  reg [3:0] r_dig1=0; // the first  digit
  reg [3:0] r_dig2=0; // the second digit
  always @(posedge w_clk) r_dig1 <= (r_dig1>=9) ? 0 : r_dig1 + 1;
  always @(posedge w_clk) if (r_dig1>=9) r_dig2 <= (r_dig2>=5) ? 0 : r_dig2 + 1;
endmodule
```

```
  3  0  1
  7  0  2
 11  0  3
 15  0  4
 19  0  5
 23  0  6
 27  0  7
 31  0  8
 35  0  9
 39  1  0
 43  1  1
 47  1  2
.
.
.
223  5  6
227  5  7
231  5  8
235  5  9
239  0  0
243  0  1
247  0  2
251  0  3
255  0  4
.
.
```

Hint for Check Point 3

- w_ud が1の時に 00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, … 58, 59, 00, 01, … と更新するモジュール.

code058.v

```
module m_top ();
  reg r_clk=0; initial forever #2 r_clk = ~r_clk;
  m_main m_main0 (r_clk);
  always@(posedge r_clk) #1 $write("%3d %d %d¥n", $time, m_main0.r_dig2, m_main0.r_dig1);
endmodule

module m_main (w_clk);
  input  wire w_clk;
  reg [31:0] r_cnt=0;
  always @(posedge w_clk) r_cnt <= (r_cnt>=1) ? 0 : r_cnt + 1;
  wire w_ud = (r_cnt==0); // wire for r_dig1,2 updating

  reg [3:0] r_dig1=0; // the first  digit
  reg [3:0] r_dig2=0; // the second digit
  always @(posedge w_clk) if (w_ud) r_dig1 <= (r_dig1>=9) ? 0 : r_dig1 + 1;
  always @(posedge w_clk) if (w_ud & r_dig1>=9) r_dig2 <= (r_dig2>=5) ? 0 : r_dig2 + 1;
endmodule
```

```
  3  0  1
  7  0  1
 11  0  2
 15  0  2
 19  0  3
 23  0  3
 27  0  4
 31  0  4
 35  0  5
 39  0  5
 43  0  6
 47  0  6
 51  0  7
 55  0  7
 59  0  8
.
.
.
463  5  8
467  5  9
471  5  9
475  0  0
479  0  0
483  0  1
487  0  1
491  0  2
.
.
```

Inside code066.v

- シリアル通信を用いた送受信回路.
- Tera Term から入力された文字をTera Termに送信する回路 code066.v を作成せよ.
  - main.xdc を main07.xdc の内容になるように修正する.
  - モジュール m_UartRx を用いて1バイトのデータを受信する. code063.v の様に入力されたデータを LED に表示すること.
  - 受信したデータをモジュール m_UartTx を用いて送信する.
- 正しい回路が完成したら, 担当のTAを呼んで確認してもらう(Check Point 4).

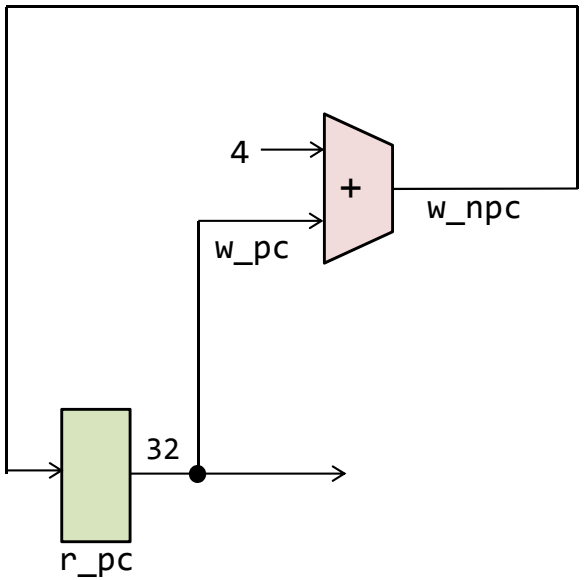
main07.xdc

```
set_property -dict { PACKAGE_PIN E3  IOSTANDARD LVCMOS33} [get_ports { w_clk }];
create_clock -add -name sys_clk -period 10.00 -waveform {0 5} [get_ports {w_clk}];

set_property -dict { PACKAGE_PIN D4  IOSTANDARD LVCMOS33} [get_ports { w_txd }];
set_property -dict { PACKAGE_PIN C4  IOSTANDARD LVCMOS33} [get_ports { w_rxd }];

set_property -dict { PACKAGE_PIN H17 IOSTANDARD LVCMOS33} [get_ports { w_led[0] }];
set_property -dict { PACKAGE_PIN K15 IOSTANDARD LVCMOS33} [get_ports { w_led[1] }];
set_property -dict { PACKAGE_PIN J13 IOSTANDARD LVCMOS33} [get_ports { w_led[2] }];
set_property -dict { PACKAGE_PIN N14 IOSTANDARD LVCMOS33} [get_ports { w_led[3] }];
set_property -dict { PACKAGE_PIN R18 IOSTANDARD LVCMOS33} [get_ports { w_led[4] }];
set_property -dict { PACKAGE_PIN V17 IOSTANDARD LVCMOS33} [get_ports { w_led[5] }];
set_property -dict { PACKAGE_PIN U17 IOSTANDARD LVCMOS33} [get_ports { w_led[6] }];
set_property -dict { PACKAGE_PIN U16 IOSTANDARD LVCMOS33} [get_ports { w_led[7] }];
```

Hint for Check Point 4

- 制約ファイル main.xdc の記述ミスが頻繁に起こっているので，制約ファイルを講義のサポートページからダウンロードできるようにした.
- Windowsでサポートページを開いて，ファイルをダウンロードし，それを main.xdc として上書きする.

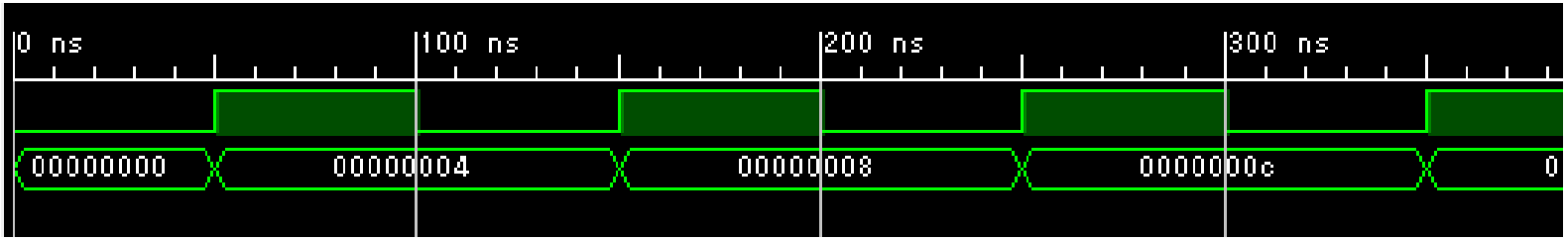
Sample Circuit 1



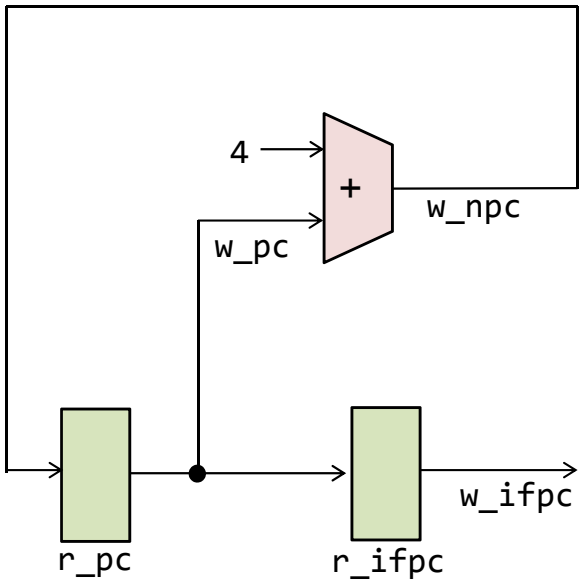
code081.v

```
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);
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] w_npc = w_pc + 4;
  always@(posedge w_clk) r_pc <= w_npc;
endmodule
```

```
  1 00000000
 51 00000004
151 00000008
251 0000000c
351 00000010
451 00000014
551 00000018
651 0000001c
751 00000020
851 00000024
951 00000028
```



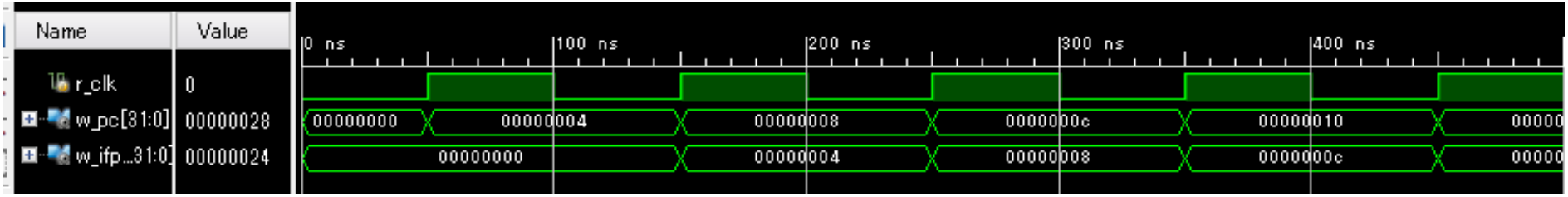
Sample Circuit 2



code082.v

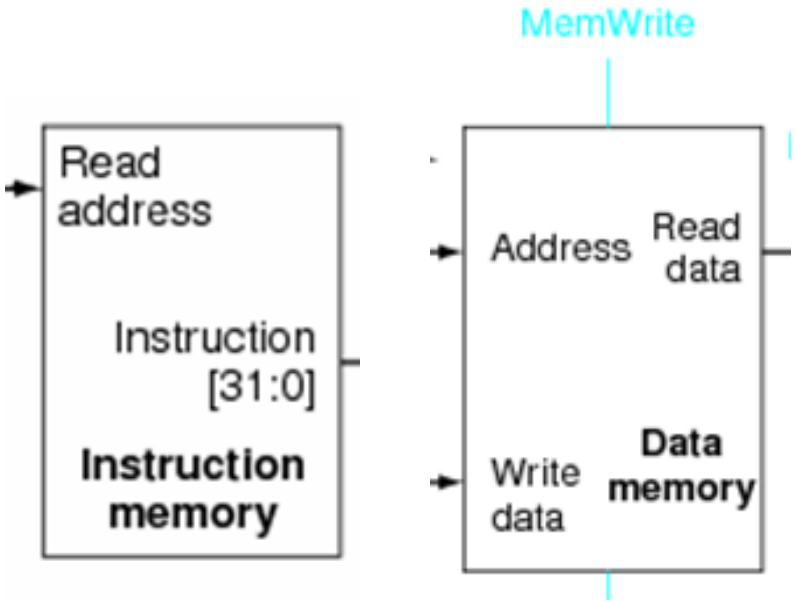
```
module m_top ();
  reg r_clk=0; initial forever #50 r_clk = ~r_clk;
  wire [31:0] w_pc, w_ifpc;
  m_main m_main0 (r_clk, w_pc, w_ifpc);
  always@(*) #1 $write("%3d %x %x¥n", $time, w_pc, w_ifpc);
endmodule

module m_main (w_clk, w_pc, w_ifpc);
  input  wire w_clk;
  output wire [31:0] w_pc, w_ifpc;
  reg [31:0] r_pc = 0;
  reg [31:0] r_ifpc = 0;

  assign w_ifpc = r_ifpc;
  assign w_pc = r_pc;
  wire [31:0] w_npc = w_pc + 4;
  always@(posedge w_clk) r_pc   <= w_npc;
  always@(posedge w_clk) r_ifpc <= w_pc;
endmodule
```

```
  1 00000000 00000000
 51 00000004 00000000
151 00000008 00000004
251 0000000c 00000008
351 00000010 0000000c
451 00000014 00000010
551 00000018 00000014
651 0000001c 00000018
751 00000020 0000001c
851 00000024 00000020
951 00000028 00000024
```



Computer Memory

- Read-only memory (ROM)
- Random-access memory (RAM)
  - Verilog HDLでは，ビット幅Bでワード数Wのメモリcm_ramを reg [B-1:0] cm_ram [0:W-1] として宣言できる.

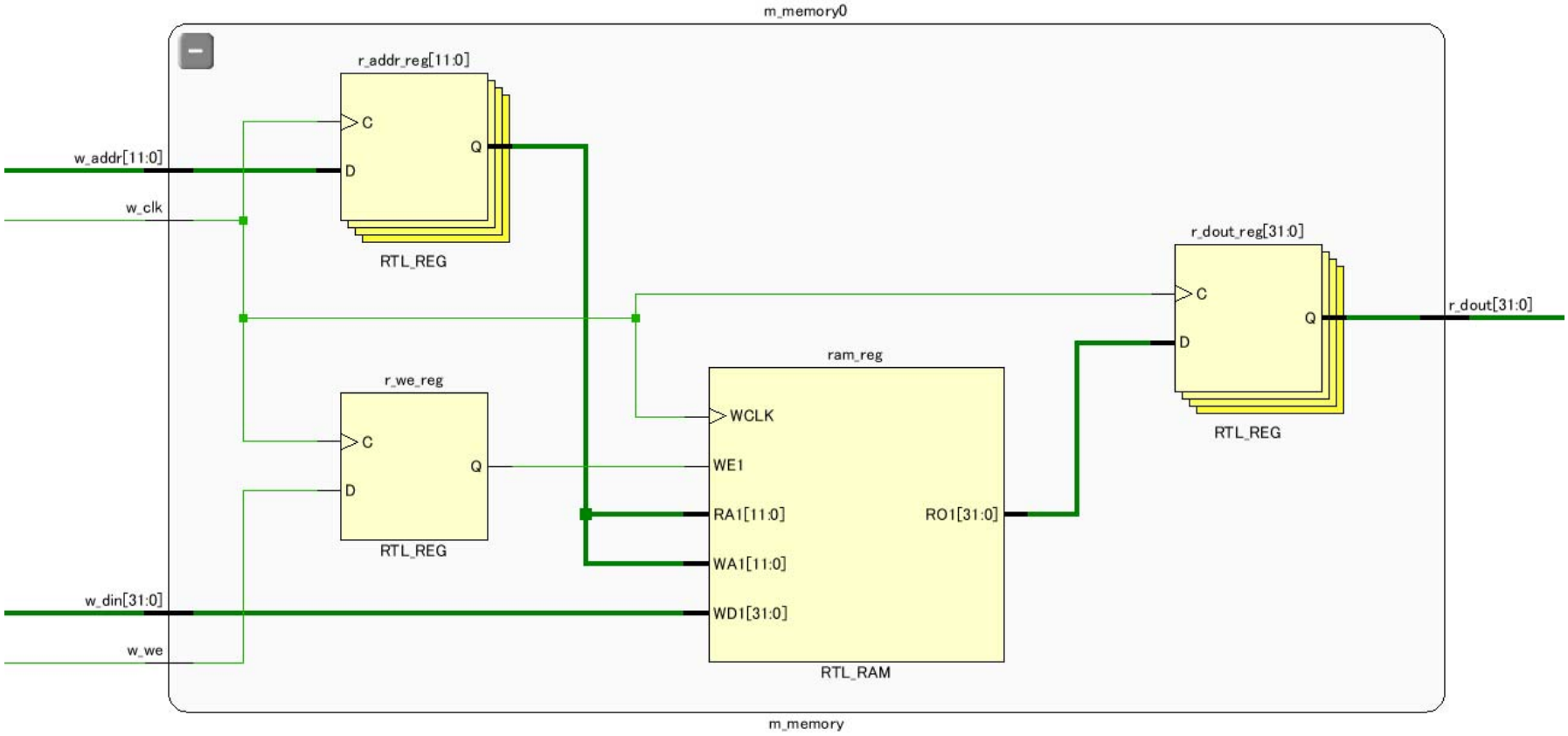


code083.v

```
module m_memory (w_clk, w_addr, w_we, w_din, r_dout);
  input  wire w_clk, w_we;
  input  wire [11:0] w_addr;
  input  wire [31:0] w_din;
  output reg  [31:0] r_dout;
  reg r_we=0;
  reg [11:0] r_addr=0;
  reg [31:0] r_din=0;
  reg [31:0] cm_ram [0:4095]; // 4K word (4096 x 32bit) memory
  always @(posedge w_clk) begin
    r_addr <= w_addr;
    r_din  <= w_din;
    r_we   <= w_we;
    r_dout <= cm_ram[r_addr];
    if (r_we) cm_ram[r_addr] <= w_din;
  end
  initial begin
    r_dout = 0;
    cm_ram[0] = {6'h0, 5'd0, 5'd0, 5'd0, 5'h0, 6'h20}; // add $0, $0, $0
    cm_ram[1] = {6'h0, 5'd2, 5'd1, 5'd2, 5'h0, 6'h20}; // add $2, $2, $1
    cm_ram[2] = {6'h0, 5'd3, 5'd1, 5'd3, 5'h0, 6'h20}; // add $3, $3, $1
    cm_ram[3] = {6'h0, 5'd4, 5'd4, 5'd4, 5'h0, 6'h20}; // add $4, $4, $4
  end
endmodule
```

Inside module m_memory

- reg [31:0] cm_ram [0:4095] として宣言したメモリが RTL_RAM (Block RAM, BRAM)にマップされている.

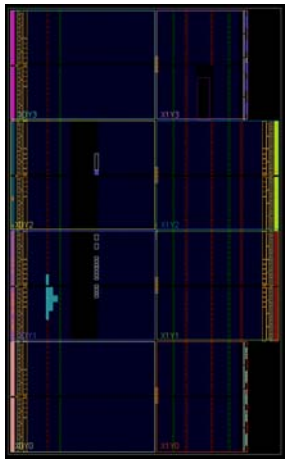


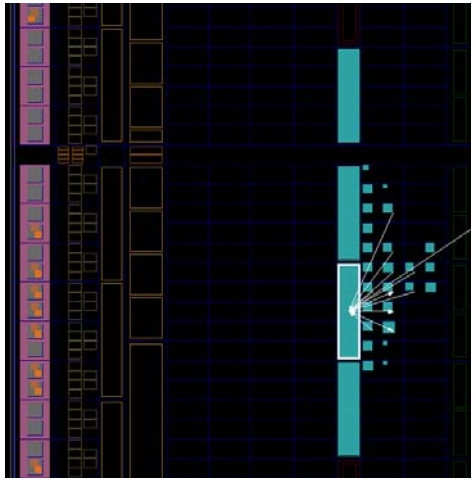
Inside module m_memory

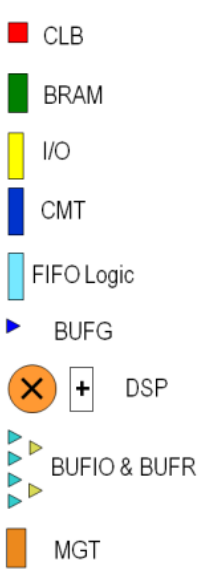
表 4: Artix-7 FPGA の機能一覧

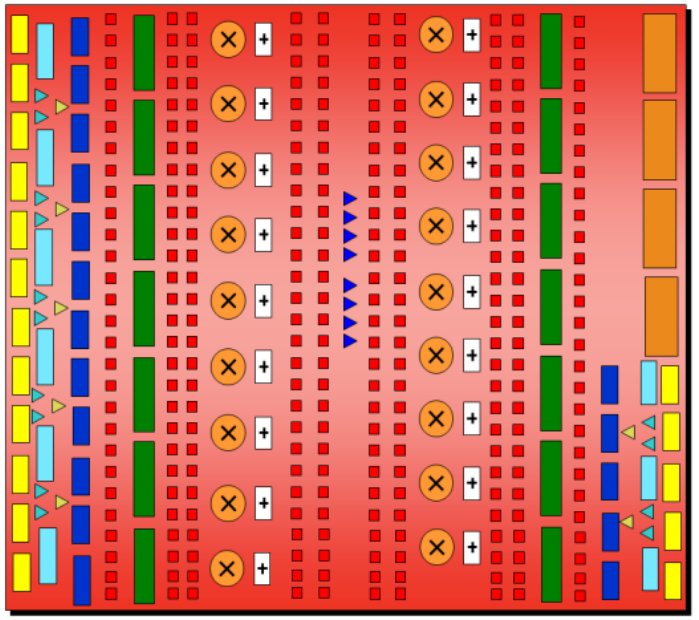
デバイス	ロジックセル	コンフィギャラブルロジックブロック (CLB)		DSP48E1 スライス[2]	ブロック RAM ブロック[3]			CMT[4]	PCIe[5]	GTP	XADC ブロック	総 I/O バンク[6]	最大ユーザー I/O[7]
		スライス数[1]	最大分散 RAM (Kb)		18Kb	36Kb	最大 (Kb)						
XC7A12T	12,800	2,000	171	40	40	20	720	3	1	2	1	3	150
XC7A15T	16,640	2,600	200	45	50	25	900	5	1	4	1	5	250
XC7A25T	23,360	3,650	313	80	90	45	1,620	3	1	4	1	3	150
XC7A35T	33,280	5,200	400	90	100	50	1,800	5	1	4	1	5	250
XC7A50T	52,160	8,150	600	120	150	75	2,700	5	1	4	1	5	250
XC7A75T	75,520	11,800	892	180	210	105	3,780	6	1	8	1	6	300
XC7A100T	101,440	15,850	1,188	240	270	135	4,860	6	1	8	1	6	300
XC7A200T	215,360	33,650	2,888	740	730	365	13,140	10	1	16	1	10	500

注記:
1. 7 シリーズ FPGA の各スライスには、4 つの LUT と 8 つのフリップフロップが含まれ、一部のスライスでのみ LUT を分散 RAM または SRL として使用できます。
2. 各 DSP スライスには 25 × 18 乗算器、加算器、アキュムレータが 1 つずつ含まれます。
3. ブロック RAM は基本的に 36Kb ですが、2 つの独立した 18Kb ブロックとしても使用できます。
4. 各 CMT には MMCM と PLL が 1 つずつ含まれます。
5. Artix-7 FPGA の PCI Express 用インターフェイス ブロックは最高 ×4 Gen 2 をサポートします。
6. コンフィギュレーション バンク 0 は含まれません。
7. 記載の数値に GTP トランシーバーは含まれません。







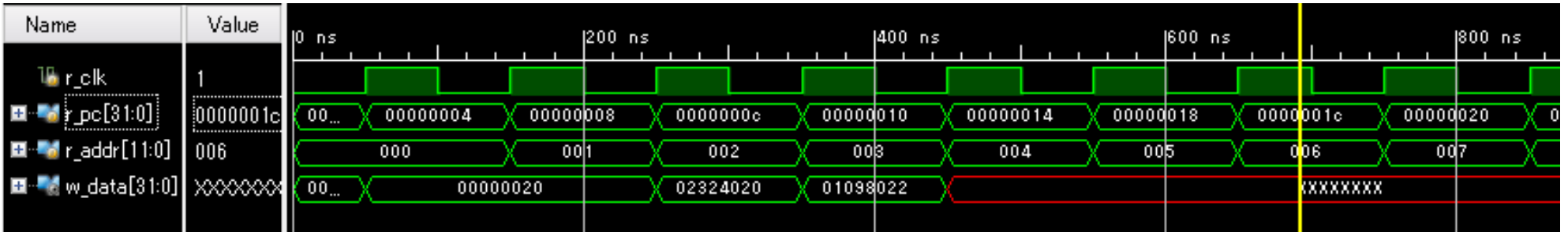


Inside module m_memory

code084.v

```
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 <= r_pc + 4;

  wire [31:0] w_data;
  m_memory m_memory0 (r_clk, r_pc[13:2], 0, 0, w_data);

  always@(*) #1 $write("%3d %d %x¥n", $time, r_pc, w_data);
endmodule
```

```
module m_memory (w_clk, w_addr, w_we, w_din, r_dout);
  input  wire w_clk, w_we;
  input  wire [11:0] w_addr;
  input  wire [31:0] w_din;
  output reg  [31:0] r_dout;
  reg r_we=0;
  reg [11:0] r_addr=0;
  reg [31:0] r_din=0;
  reg [31:0] cm_ram [0:4095]; // 4K word (4096 x 32bit) memory
  always @(posedge w_clk) begin
    r_addr <= w_addr;
    r_din  <= w_din;
    r_we   <= w_we;
    r_dout <= cm_ram[r_addr];
    if (r_we) cm_ram[r_addr] <= w_din;
  end
  initial begin
    r_dout = 0;
    cm_ram[0] = {6'h0, 5'd0, 5'd0, 5'd0, 5'h0, 6'h20}; // add $0, $0, $0
    cm_ram[1] = {6'h0, 5'd2, 5'd1, 5'd2, 5'h0, 6'h20}; // add $2, $2, $1
    cm_ram[2] = {6'h0, 5'd3, 5'd1, 5'd3, 5'h0, 6'h20}; // add $3, $3, $1
    cm_ram[3] = {6'h0, 5'd4, 5'd4, 5'd4, 5'h0, 6'h20}; // add $4, $4, $4
  end
endmodule
```



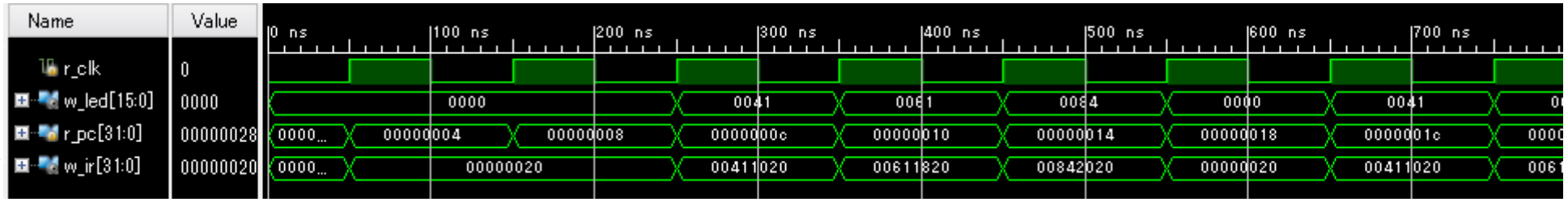
Inside module m_proc01

- プロセッサの実装に向けた最初の版．命令をフェッチ (fetch) するだけのモジュール．命令メモリから命令を読み出し，取得することを命令フェッチと呼ぶ．

code085.v

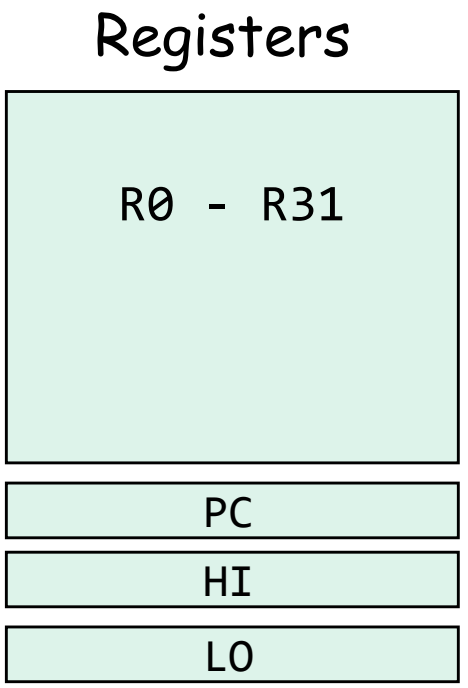
```
module m_top ();
  reg r_clk=0; initial forever #50 r_clk = ~r_clk;
  wire [15:0] w_led;
  m_proc01 m_proc01 (r_clk, 0, 1, w_led);
  always@(*) #1 $write("%3d %x¥n", $time, w_led);
endmodule


module m_proc01 (w_clk, w_btnu, w_btnd, w_led);
  input  wire w_clk, w_btnu, w_btnd;
  output wire [15:0] w_led;

  reg [31:0] r_pc = 0;
  always @(posedge w_clk) r_pc <= r_pc + 4;

  wire        w_we  = w_btnu;
  wire [11:0] w_addr= (w_btnu) ? r_pc[13:2] : {10'd0, r_pc[3:2]};
  wire [31:0] w_din = r_pc;
  wire [31:0] w_ir;
  m_memory m_imem (w_clk, w_addr, w_we, w_din, w_ir);

  assign w_led = (w_btnd) ? w_ir[31:16] : w_ir[15:0];
endmodule
```

```
module m_memory (w_clk, w_addr, w_we, w_din, r_dout);
  input  wire w_clk, w_we;
  input  wire [11:0] w_addr;
  input  wire [31:0] w_din;
  output reg  [31:0] r_dout;
  reg r_we=0;
  reg [11:0] r_addr=0;
  reg [31:0] r_din=0;
  reg [31:0] cm_ram [0:4095]; // 4K word (4096 x 32bit) memory
  always @(posedge w_clk) begin
    r_addr <= w_addr;
    r_din  <= w_din;
    r_we   <= w_we;
    r_dout <= cm_ram[r_addr];
    if (r_we) cm_ram[r_addr] <= w_din;
  end
  initial begin
    r_dout = 0;
    cm_ram[0] = {6'h0, 5'd0, 5'd0, 5'd0, 5'h0, 6'h20}; // add $0, $0, $0
    cm_ram[1] = {6'h0, 5'd2, 5'd1, 5'd2, 5'h0, 6'h20}; // add $2, $2, $1
    cm_ram[2] = {6'h0, 5'd3, 5'd1, 5'd3, 5'h0, 6'h20}; // add $3, $3, $1
    cm_ram[3] = {6'h0, 5'd4, 5'd4, 5'd4, 5'h0, 6'h20}; // add $4, $4, $4
  end
endmodule
```

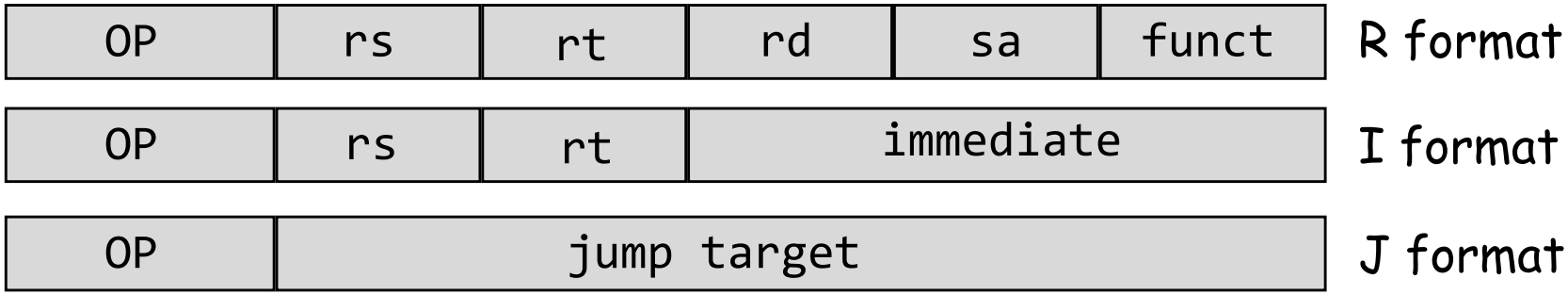


MIPS R3000 Instruction Set Architecture (ISA)

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special



3 Instruction Formats: all 32 bits wide



MIPS 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 $zero) for constants like 1
  - have special instructions that contain constants !

```
addi  $s0, $s1, 4   # $s0 = $s1 + 4
```

- Machine format (I format):

- The constant is kept inside the instruction itself
  - Immediate format limits values to the range $+2^{15}-1$ to -2^{15}

Integer（整数）Representation

- 2の補数 (two's complement) による符号付き数の表現
  - 正数のビットを反転させ，1を加えたものを負数とする．
  - 8ビットであれば，- 128 ～127 までの 255個の整数を表現できる

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

MIPS Arithmetic Instructions

- MIPS assembly language arithmetic statement

```
add   $t0, $s1, $s2
sub   $t0, $s1, $s2
```

- 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 ($t0,$s1,$s2) – indicated by $

MIPS Arithmetic Instructions

- add
- addi
- addiu
- addu
- sub
- subu

Add	add	R	R[rd]=R[rs]+R[rt]	(1)	0/20
Add Immediate	addi	I	R[rt]=R[rs]+SignExtImm	(1)(2)	8
Add Imm. Unsigned	addiu	I	R[rt]=R[rs]+SignExtImm	(2)	9
Add Unsigned	addu	R	R[rd]=R[rs]+R[rt]	(2)	0/21
Subtract	sub	R	R[rd]=R[rs]-R[rt]	(1)	0/22
Subtract Unsigned	subu	R	R[rd]=R[rs]-R[rt]		0/23

MIPS Logical Instructions

- and
- andi
- nor
- or
- ori
- xor
- xori

And	and	R	R[rd]=R[rs]&R[rt]		0/24
And Immediate	andi	I	R[rt]=R[rs]&ZeroExtImm	(3)	c
Nor	nor	R	R[rd]=~(R[rs]|R[rt])		0/27
Or	or	R	R[rd]=R[rs]|R[rt]		0/25
Or Immediate	ori	I	R[rt]=R[rs]|ZeroExtImm	(3)	d
Xor	xor	R	R[rd]=R[rs]^R[rt]		0/26
Xor Immediate	xori	I	R[rt]=R[rs]^ZeroExtImm		e

MIPS Shift Instructions

- sll (shift left logical)
- srl (shift right logical)
- sra (shift right arithmetic)
- sllv (shift left logical variable)
- srlv
- srav

Shift Left Logical	sll	R	R[rd]=R[rs]≪shamt	0/00
Shift Right Logical	srl	R	R[rd]=R[rs]≫shamt	0/02
Shift Right Arithmetic	sra	R	R[rd]=R[rs]≫>shamt	0/03
Shift Left Logical Var.	sllv	R	R[rd]=R[rs]≪R[rt]	0/04
Shift Right Logical Var.	srlv	R	R[rd]=R[rs]≫R[rt]	0/06
Shift Right Arithmetic Var.	srav	R	R[rd]=R[rs]≫>R[rt]	0/07

References

- Computer Logic Design support page
  - http://www.arch.cs.titech.ac.jp/lecture/CLD/
- 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 Nexys 4 DDR Atrix-7 FPGA
  - https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/start
  - https://store.digilentinc.com/nexys-4-ddr-artix-7-fpga-trainer-board-recommended-for-ece-curriculum/
- Verilog HDL
  - https://ja.wikipedia.org/wiki/Verilog
- Frix (Feasible and Reconfigurable IBM PC Compatible SoC)
  - http://www.arch.cs.titech.ac.jp/a/Frix/
- 7シリーズFPGAデータシート：概要
  - https://japan.xilinx.com/support/documentation/data_sheets/j_ds180_7Series_Overview.pdf
- 7シリーズFPGA コンフィギャラブルロジックブロック ユーザーガイド
  - https://japan.xilinx.com/support/documentation/user_guides/j_ug474_7Series_CLB.pdf

Exercise(4)

- Project_7
  - Adder の設計・実装.
  - VivadoとWNSを理解する.
- Project_8
  - Synthesis と Implementation を理解する.

Inside code078.v : n-bit Ripple Carry Adder

- 100MHzの動作周波数で制約を満たす n-bit Adder の最大の n を求める.
  - 求めた n の値をレポートに記入すること.
  - ヒント: D_N 48, 56 としてそれぞれの合成結果を確認してみる.
  - nを変化させてWNSを求めた結果を示す. Critical Path, Max Freq. は見積もり.
  - The carry out signal takes two gate delays per bit.

n	WNS	Critical Path (ns)	Max Freq.
8	6.858	3.142	
16	4.056	5.944	
24	2.266	7.734	129.30
32	0.982	9.018	110.89
40	0.997	9.003	111.07
48	0.479	9.521	105.03
56	−0.165	10.165	98.38
64	−2.262	12.262	81.55

```
/*****************************************************************************/
/* Report body(6). Find the maximum n of the n-bit Adder satisfying the constr. */

Write the value of n:

/*****************************************************************************/
/* Report body(7). Copy and paste the output of code079.v here.              */
```