製作CPUの説明

lay.sakura

2010年11月13日

目次

0.1 前書き

3 年前期課程の「コンピュータアーキテクチャ」の講義で，CPU やその他のコンピュータの構成要素 (メモリ・プログラムカウンタなど) の動作原理を学んだ．本実験には，そこで学んだものを実際に自分の手で設計・実装する役割がある．

また，CPU を自ら設計・実装することにより，既製のプロセッサ (x86 系など) についての理解を深めることもできる．それにより，OS のような基盤的なソフトウェアを開発する際も，CPU を正しく意識することができる．

本実験では，IA-32 アーキテクチャのサブセットを命令セットに持つシンプルな CPU を制作し，その動作テストまでを行った．ただし，FPGA を使用した実機テストまでには至らず，シミュレータ上のテストまでを実施した．

0.2 製作した CPU の全体像

この節では，CPU の各パーツの詳細には立ち入らずに，大まかな特徴を述べる．また，実装した命令セットについてもこの節で説明する．

0.2.1 制作した CPU の特徴

- パイプライン化は行なっていない．
- IA-32 アーキテクチャのサブセットを命令セットに持つ (詳細は 0.2.3)
- フェッチ (F)，リード (R)，実行 (X)，データメモリアクセス (M)，ライトバック (W) の 5 フェーズで動作．各フェーズには 1 クロックを要する．
- リトルエンディアン

0.2.2 制作した CPU の機構

[2] から拝借した図 1 が，本 CPU の機構を表している．ただし，図の中の IM,TR2 は設けておらず，また，新規にセレクタを 3 つ追加した (0.5.14 で説明).

0.2.3 実装した命令セット

本 CPU では，実験課題として「必須」扱いされている命令のみを全て実装した．

図 2 に，実装した命令とそれらのフェーズごとの動作を示した．表中で同じ色になっているセルは，フェーズ内で動作が同じため，1 つにまとめることが可能な動作である．

各命令の意図している動作や，対応する機械語は，[2] の `http://www.mtl.t.u-tokyo.ac.jp/~jikken/cpu/wiki/index.php?ISA%2FSubset` の「命令セット」のセクションに記述してある．(もちろん丸写しすることもできたが，徒にページ数を増やすだけだと判断しました．)

ただし，以下の命令に関しては，上記ページにないものを独自に判断して実装したので，それに関する詳細を記す．

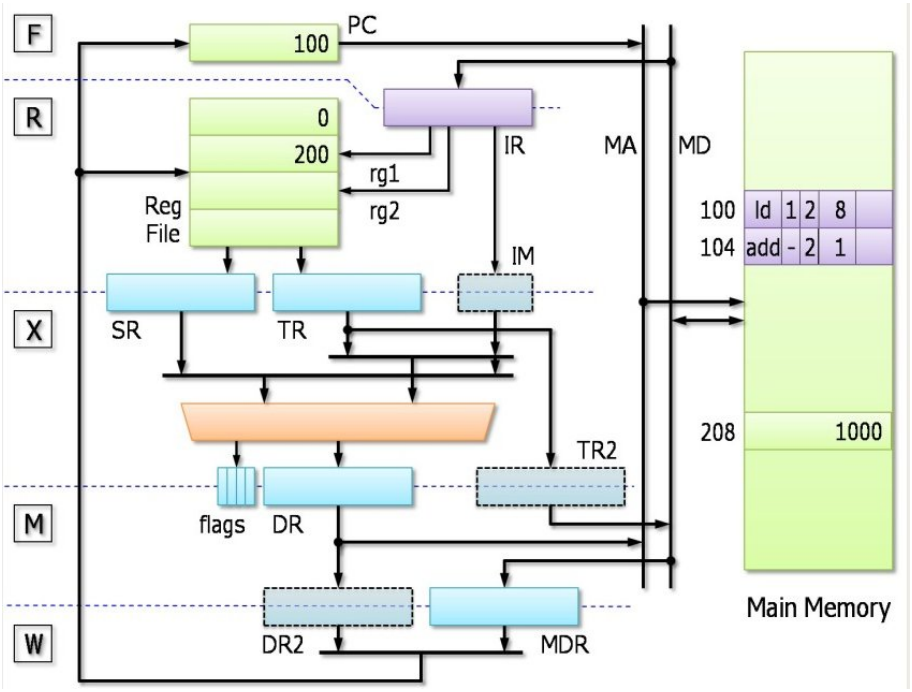

図 1　CPU の機構

	F	R	X	M	W
zLD	PC++→MA MD→IR	RF(rg1)→SR RF(rg2)→TR	SR + IR(sim8)→DR (flagは更新しない)	DR→MA MD→MDR	MDR→RF(rg2)
zST				DR→MA TR→MD	-
zLIL			IR(im16)->DR	DR->DR2	DR2->RF(rg2)
zMOV			SR->DR		
二項演算 reg-reg zADD zSUB zCMP zAND zOR zXOR			TR op SR→DR フラグのセット		
zBcc			PC + sim8→DR	DR → PC	-
zB					
zNOP			-	-	-
シフト zSLL zSLA zSRL zRSA			op[shift](TR, IR(sim8))->DR フラグのセット	DR->DR2	DR2->RF(rg2)
zHLT			PC--	-	-

図 2　命令セット

zHLT

zHLT 以外の命令では，X フェーズにおいてプログラムカウンタの値をインクリメントしているのだが，zHLT 命令ではそれをしない．そのことにより，何サイクルが経過しても常にプログラムカウンタは zHLT 命令列が入ったメモリ番地を指すことになるので，HLT で意図される動作となる．

zADD，zSUB，zCMP，zSLL

これら命令においては，フラグレジスタの値が以下の条件でセットされ，それが次のサイクルの X フェーズまで保持される．逆に，これら以外の命令が発行されたサイクルとその直後のサイクルでは，フラグレジスタの値は意味を持たない．

SF(**サインフラグ**)

zADD,zSUB 実行後に，オペランドに符号変化があったときに立つ．また，zCMP で $rg1 < rg2$ が成立したときにも立つ．

ZF(**ゼロフラグ**)

zADD,zSUB 実行後に，オペランドが 0 になったときに立つ．また，zCMP で $rg1 == rg2$ が成立したときにも立つ．

CF(**キャリーフラグ**)

zADD,zSUB,zSLL 実行後に，桁あふれが生じたときに立つ．また，zCMP で $rg1 < rg2$ が成立したときにも立つ．

OF(**オーバフローフラグ**)

(正の値) + (正の値) = (負の値)

または

(負の値) + (負の値) = (正の値)

が成立したときに立つ．

0.3 使用したソフトウェア

記述言語

Verilog HDL

エディタ

GNU Emacs

メモリモジュールの自動生成

Quartus II ウェブ・エディション v10.0

シミュレータ

ModelSim PE Student Edition 6.6c

0.4 動作結果

動作テストは，実装した各命令が正しく動作するかという観点で行った．命令が正しく動作するということは，各パーツ (レジスタファイルやプログラムカウンタなど) が正しく動作し，それらの協調がとれていることを意味すると考えたからである．

0.4.1 zLD,zST,zLIL,zHLT の動作テスト

ソース 1 をアセンブルしたコードをメモリに乗せてシミュレートした結果，図 3 を得た．意図通りの値をメモリに読み書きでき，かつプログラムカウンタの値も途中で止まっていることが分かる．

ソース 1　プログラムカウンタ

```
1         // Using operations:
2         // zLD, zST, zLIL, zHLT
3
4         .include "ia-32z.s"
5
6         //a number to store into mem.
7         zLIL 7 ax
8
9         //mem address to store '7'
10        zLIL 10 cx
11
12        //store '7' to mem.
13        zST ax 0 cx
14
15        //load '7' from mem to dx
16        zLD 0 cx dx
17
18        zHLT
```

0.4.2 zMOV,zB,zADD,zLIL の動作テスト

ソース 2 をアセンブルしたコードをメモリに乗せてシミュレートした結果，図 4 を得た．意図通りに，フィボナッチ数列の値が ALU から計算結果として出力されていることが分かる．

ソース 2　プログラムカウンタ

```
1         //// Using operations:
2         //// zLIL, zMOV, zADD, zB
3
4         .include "ia-32z.s"
5
6         //set a_0
7         zLIL 1 ax
8
9         //set a_1
10        zLIL 1 cx
11
12 //以下をループで回す
13
14        //tmp = a_{i-1}
15        zMOV cx dx
16
17        //a_i = a_{i-1} + a{i-2}
```

```
18      zADD ax cx
19
20      //a_{i−2} = tmp
21      zMOV dx ax
22
23      //zB −3 //これはハンドアセンブルしました
```

0.4.3 zADD,zSUB,zAND,zOR,zXOR,zLIL の動作テスト

ソース 3 をアセンブルしたコードをメモリに乗せてシミュレートした結果，図 5 を得た．意図通りの計算結果が ALU から出力されていることが分かる．

ソース 3　プログラムカウンタ

```
1       //// Using operations:
2       //// zLIL, zADD, SUB, zAND, zOR, zXOR
3
4       .include ”ia−32z.s”
5
6       //0: ax = 1
7       zLIL 1 ax
8
9       //1: cx = 2 (fixed)
10      zLIL 2 cx
11
12      //2: ax += 2 (ax −> 3)
13      zADD cx ax
14
15      //3: ax −= 2 (ax −> 1)
16      zSUB cx ax
17
18      //4: ax = OR(ax,2) (ax−>3)
19      zOR cx ax
20
21      //5: ax = XOR(ax,2) (ax−>1)
22      zXOR cx ax
23
24      //6: ax = AND(ax,2) (ax−>0)
25      zAND cx ax
```

0.4.4 zSLL,zSRL,zSRA,zLIL の動作テスト

ソース 4 をアセンブルしたコードをメモリに乗せてシミュレートした結果，図 6 を得た．意図通りの計算結果が ALU から出力されていることが分かる．

ソース 4　プログラムカウンタ

```
1       //// Using operations:
2       //// zLIL, zSLL, zSRL, zSRA
3
```

```
4       .include "ia-32z.s"
5
6       //0: ax = 1
7       zLIL 1 ax
8
9       //1: ax << 2 (ax -> 4)
10      zSLL 2 ax
11
12      //2: cx = -4 (11111111_11111111_11111111_11111100)
13      zLIL -4 cx
14
15      //3: cx >> 2 (cx -> 1073741823)
16      zSRL 2 cx
17
18      //4: dx = -4
19      zLIL -4 dx
20
21      //5: dx >>> 2 (dx -> -1)
22      zSRA 2 dx
```

0.4.5 zCMP,zBcc,zLIL の動作テスト

ソース 5 をアセンブルしたコードをメモリに乗せてシミュレートした結果，図 7 を得た．意図通りに，比較の結果によってジャンプしていることが分かる．

ソース 5　プログラムカウンタ

```
1       //// Using operations:
2       //// zLIL, zCMP, zBcc
3
4       .include "ia-32z.s"
5
6       //0: ax = 5
7       zLIL 5 ax
8
9       //1: cx = 1
10      zLIL 1 cx
11
12      //2: if (cx < ax)
13      zCMP cx ax
14
15      //3: then goto -2
16 // zBcc 0010 -2 //これはハンドアセンブルしました.
```

0.5 各パーツの説明

ここでは，制作した CPU を構成する各パーツについて詳細な解説をする．それに先立ち，全てのソースファイルからインクルードされている，マクロ定義のファイルをソース 6 に示す．ここでは，以下のマクロを定義している．

- 命令コード
- フラグ
- zBcc 命令におけるジャンプ条件
- フェーズ

ソース 6　マクロ定義

```
1 // op codes
2 `define op_h 12
3 `define zLD 13'b1000_1011_01_xxx
4 `define zST 13'b1000_1001_01_xxx
5 `define zLIL 13'b0110_0110_10_111
6 `define zMOV 13'b1000_1001_11_xxx
7 `define zADD 13'b0000_0001_11_xxx
8 `define zSUB 13'b0010_1001_11_xxx
9 `define zCMP 13'b0011_1001_11_xxx
10 `define zAND 13'b0010_0001_11_xxx
11 `define zOR 13'b0000_1001_11_xxx
12 `define zXOR 13'b0011_0001_11_xxx
13 `define zNOT 13'b1111_0111_11_010
14 `define zSLL 13'b1100_0001_11_100
15 // zSLL and zSLA are totally the same.
16 `define zSRL 13'b1100_0001_11_101
17 `define zSRA 13'b1100_0001_11_111
18 `define zB 13'b1001_0000_11_101
19 `define zBcc 13'b1001_0000_01_11x
20 `define zHLT 13'b1111_0100_xx_xxx
21
22 // Flags
23 `define sf 0
24 `define zf 1
25 `define cf 2
26 `define of 3
27 `define flags_h 3
28
29 // cc (in zBcc)
30 `define o 4'b0000
31 `define no 4'b0001
32 `define b 4'b0010
33 `define nb 4'b0011
34 `define e 4'b0100
35 `define ne 4'b0101
36 `define be 4'b0110
37 `define nbe 4'b0111
38 `define s 4'b1000
39 `define ns 4'b1001
40 `define l 4'b1100
41 `define nl 4'b1101
42 `define le 4'b1110
43 `define nle 4'b1111
44
```

```
45 // Phases
46 `define f 0
47 `define r 1
48 `define x 2
49 `define m 3
50 `define w 4
51 `define phase_h 4
```

以下からパーツごとの説明に入るが，入出力や記憶内容について，ソースを一見すれば明らかなものの (クロック，リセット信号，フェーズを見分ける信号など) 説明は割愛することがあることを断っておく．

0.5.1 テストベンチ

テストベンチのソースコードをソース 7 に示す．ここで行っているのは，定期的にクロック信号をトップモジュールに送り出すことと，最初に一回リセット信号をトップモジュールに送り出すことである．,

ソース 7　テストベンチ

```
1  `timescale 1ns / 1ps
2
3  module test_cpu;
4    reg clk, rst;
5    initial
6      begin
7        clk = 0;
8        rst = 1;
9        forever #10 clk = ~clk;
10     end
11
12   initial
13     begin
14       #5 rst = 0;
15       #15 rst = 1;
16     end
17
18   cpu cpu_bench(clk, rst);
19
20 endmodule
```

0.5.2 トップモジュール

トップモジュールのソースコードをソース 8 に示す．ここで行っていることは，図 1 を元に各パーツへ入出力線をつなぐことと，フェーズをクロック毎に回すことである．

ソース 8　トップモジュール

```
1  module cpu(
2                clk, rst
3                );
4
```

```
5  `include "defines.vh"
6
7    input clk, rst;
8    reg [`phase_h:0] phase;
9
10   // wire naming rule:
11   // defines only output wires.
12   // must determine which port to input the output wires.
13   wire [31:0] pc_out;
14
15   wire [31:0] memory_out;
16   wire alu_memory_wren;
17
18   wire [7:0] ir_op1_out;
19   wire [1:0] ir_op2_out;
20   wire [2:0] ir_op3_out;
21   wire [2:0] ir_rg1_out;
22   wire [2:0] ir_rg2_out;
23   wire [7:0] ir_sim8_out;
24   wire [15:0] ir_im16_out;
25   wire [3:0] ir_tttn_out;
26
27   wire [31:0] regfile_sr_out;
28   wire [31:0] regfile_tr_out;
29
30   wire [31:0] sr_out;
31
32   wire [31:0] tr_out;
33
34   wire [31:0] alu_dr_out;
35   wire [`flags_h:0] alu_flags_out;
36
37   wire [`flags_h:0] flags_out;
38
39   wire [31:0] dr_out;
40
41   wire [31:0] dr2_out;
42
43   wire [31:0] mdr_out;
44
45   wire [31:0] selecter_regfile;
46   wire [31:0] selecter_pc;
47   wire [4:0] selecter_op_data;
48
49   selecter_regfile selecter_regfile_cpu
50     (
51       ir_op1_out, ir_op2_out, ir_op3_out,
52       dr2_out, mdr_out,
53       selecter_regfile);
54
55   selecter_pc_inc_jmp selecter_pc_cpu
56     (
```

```
57      ir_op1_out, ir_op2_out, ir_op3_out,
58      pc_out, dr_out,
59      selecter_pc);
60
61  selecter_op_data selecter_op_data_cpu
62    (
63      clk, rst, phase,
64      pc_out[4:0], dr_out[4:0],
65      selecter_op_data);
66
67  pc pc_cpu(clk, rst, phase,
68                  selecter_pc, pc_out);
69
70  ir ir_cpu(clk, rst, phase,
71                  memory_out,
72                  ir_op1_out, ir_op2_out, ir_op3_out,
73                  ir_rg1_out, ir_rg2_out,
74                  ir_sim8_out, ir_im16_out, ir_tttn_out);
75
76  regfile regfile_cpu(clk, rst, phase,
77                              ir_rg1_out, ir_rg2_out,
78                              selecter_regfile,
79                              regfile_sr_out, regfile_tr_out);
80
81  sr sr_cpu(clk, rst, phase, regfile_sr_out, sr_out);
82
83  tr tr_cpu(clk, rst, phase, regfile_tr_out, tr_out);
84
85  alu alu_cpu(pc_out,
86                  ir_op1_out, ir_op2_out, ir_op3_out,
87                  ir_sim8_out, ir_im16_out, ir_tttn_out,
88                  sr_out, tr_out,
89                  alu_dr_out,
90                  flags_out, alu_flags_out,
91                  alu_memory_wren);
92
93  flags flags_cpu(clk, rst, phase,
94                          ir_op1_out, ir_op2_out, ir_op3_out,
95                          alu_flags_out, flags_out);
96
97  dr dr_cpu(clk, rst, phase, alu_dr_out, dr_out);
98
99  dr2 dr2_cpu(clk, rst, phase, dr_out, dr2_out);
100
101 mdr mdr_cpu(clk, rst, phase, memory_out, mdr_out);
102
103 memory memory_cpu
104   (clk, tr_out, selecter_op_data, dr_out[4:0], alu_memory_wren, memory_out);
105
106 always @(posedge clk or negedge rst) begin
107   if (rst == 0) begin
108     phase <= 5'b00001; // turn to F phase
```

```
109     end
110     else if(phase[‘f] == 1) begin
111       phase <= 5’b00010;
112     end
113     else if(phase[‘r] == 1) begin
114       phase <= 5’b00100;
115     end
116     else if(phase[‘x] == 1) begin
117       if ({ir_op1_out,ir_op2_out,ir_op3_out} == ‘zHLT)
118         phase <= 5’b00100; // HLT!
119       else phase <= 5’b01000;
120     end
121     else if(phase[‘m] == 1) begin
122       phase <= 5’b10000;
123     end
124     else if(phase[‘w] == 1) begin
125       phase <= 5’b00001;
126     end
127   end // always @(...)
128
129 endmodule
```

0.5.3 プログラムカウンタ

プログラムカウンタのソースコードをソース 9 に示す．主な仕様は以下の通りである．

入力

- 新しいプログラムカウンタの番号 (32bits)．W フェーズで入力される．

出力

- 現在のプログラムカウンタの番号 (32bits).

記憶内容

- 現在のプログラムカウンタの番号 (32bits).

動作説明

メモリから命令列 (32bits) を取り出す際の，対象となるメモリ番地を保持・書き換えする．
W フェーズでプログラムカウンタの新しい値 (32bits) を受け取る．新しい値は，通常「現在の値 +1」だが，ジャンプ命令を受けたサイクルではジャンプ先アドレスになる．ジャンプ命令かを見て適切な入力を送り込むのは，21 にて示すセレクタである．

ソース 9　プログラムカウンタ

```
1 module pc(
2               clk, rst,
3               phase,
4               pc_in,
```

```
5               pc_reg
6               );
7
8  `include "defines.vh"
9
10   input clk, rst;
11   input [`phase_h:0] phase;
12   input [31:0] pc_in;
13
14   output reg [31:0] pc_reg;
15
16   always @(posedge clk or negedge rst) begin
17     if (rst == 0) begin
18       pc_reg <= 32'h00000000;
19     end
20
21     if (phase[`w] == 1) begin
22       pc_reg <= pc_in;
23     end
24
25   end // always @(...)
26
27 endmodule
```

0.5.4 メモリ

メモリは，[2] 中の http://www.mtl.t.u-tokyo.ac.jp/~jikken/cpu/wiki/index.php?Design%2FTips%2FRAM に従い自動生成した．分量が多い上に，オリジナルな点もないので，ソース掲載は割愛する．

主な動作は以下の通りである．

入力

- 新たに格納するデータ (32bits)
- 読み取り番地 (5bits)
- 書き込み番地 (5bits)
- ライトイネーブル信号 (1bit)．これが 1 のときにのみ，データを書き込める．

出力

- 読み取ったデータ (32bits)

記憶内容

- 命令列 (32bits) または任意のデータ (32bits).

動作説明

F フェーズで，プログラムカウンタの指す番地から命令列を出力し，M フェーズでデータを入出力する．

0.5.5 IR

IR のソースコードをソース 10 に示す．主な仕様は以下の通りである．

入力

- メモリからフェッチした命令列 (32bits)

出力

- 各命令を識別するコード (13bits)
- 命令の対象のレジスタ ID(3bits ×2)．いわゆる rg1, rg2.
- sim8(8bits)
- 即値 (16bits)．いわゆる im16．リトルエンディアンを採用している．
- zBcc 命令におけるジャンプ条件 (4bits)．いわゆる tttn.

記憶内容

以下のものを，1 サイクルで R フェーズ以降保持する．

- 命令コード
- sim8
- im16
- tttn

動作説明

F フェーズでメモリから命令列を読み取る．その命令列から，まず rg1, rg2 抽出し，F フェーズ中にレジスタファイルに伝える．その他抽出する，命令コード，sim8，im16，tttn は，X フェーズや W フェーズで使用することになるため，R フェーズ以降も保持・出力する．

ソース 10　IR

```
1  module ir(
2              clk, rst, phase,
3              ir_in,
4              op1, op2, op3,
5              rg1, rg2, sim8, im16,
6              tttn
7              );
8
9  `include "defines.vh"
10
11   input clk, rst;
12   input [`phase_h:0] phase;
13   input [31:0] ir_in;
14   output reg [7:0] op1;
15   output reg [1:0] op2;
16   output reg [2:0] op3;
17   output [2:0] rg1;
18   output [2:0] rg2;
```

```
19  output reg [7:0] sim8;
20  output reg [15:0] im16;
21  output reg [3:0] tttn; // used only in zBcc
22
23  assign {rg1,rg2}
24    = ir_out(ir_in);
25
26  function [5:0] ir_out;
27    input [31:0] ir_in;
28
29    begin
30      ir_out
31        = {ir_in[21:19], ir_in[18:16]};
32
33      casex (ir_in[31:19])
34        `zLD:
35          // in these ops, rg1 and rg2 are swaped
36          ir_out
37          = {ir_in[18:16], ir_in[21:19]};
38        `zST:
39          // in these ops, rg1 and rg2 are swaped
40          ir_out
41          = {ir_in[18:16], ir_in[21:19]};
42      endcase
43    end
44
45  endfunction
46
47  always @(posedge clk or negedge rst) begin
48    if (rst == 0) begin
49      op1 <= 8'b00000000;
50      op2 <= 2'b00;
51      op3 <= 3'b000;
52      sim8 <= 8'b00000000;
53      im16 <= 16'b00000000_00000000;
54      tttn <= 4'b0000;
55    end
56
57    else if (phase[`r] == 1) begin
58      op1 <= ir_in[31:24];
59      op2 <= ir_in[23:22];
60      op3 <= ir_in[21:19];
61      sim8 <= ir_in[15:8];
62      im16 <= {ir_in[7:0],ir_in[15:8]};
63      tttn <= ir_in[19:16];
64    end
65  end
66
67 endmodule
```

0.5.6 レジスタファイル

レジスタファイルのソースコードをソース 11 に示す．主な仕様は以下の通りである．

入力

- 使用するレジスタの ID(3bits ×2)．いわゆる rg1,rg2
- 指定されたレジスタに書き込むデータ (32bits)

出力

- rg1,rg2 で指定されたレジスタの値 (32bits)

記憶内容

- 32bits ×8 のレジスタの値．

動作説明

レジスタには，ax,cx,dx,bx,sp,bp,si,di の 8 種類を用意している．本 CPU は PUSH/POP 系の命令を実装していないため，sp や bp を含め，全てのレジスタを対等に使用できる．

F フェーズでは，W フェーズからライトバックされた値を指定されたレジスタ書き込む．

R フェーズでは，rg1 で指定されたレジスタの値を SR に，rg2 で指定されたレジスタの値を TR に送る．

ソース 11　レジスタファイル

```
1  module regfile(
2                      clk, rst,
3                      phase,
4                      rg1, rg2,
5                      regfile_in,
6                      sr, tr
7                      );
8
9  `include "defines.vh"
10
11   input clk, rst;
12   input [`phase_h:0] phase;
13   input [2:0] rg1, rg2;
14   input [31:0] regfile_in;
15   output reg [31:0] sr, tr;
16
17   reg [31:0] regfile_reg [0:`regfile_h];
18
19   always @(posedge clk or negedge rst) begin
20     if (rst == 0) begin
21       // doesn't use for statement to explicit
22       // non-blocking assignment
23       regfile_reg[`ax] <= 32'h00000000;
24       regfile_reg[`cx] <= 32'h00000000;
```

```
25          regfile_reg[‘dx] <= 32’h00000000;
26          regfile_reg[‘bx] <= 32’h00000000;
27          regfile_reg[‘sp] <= 32’h00000000;
28          regfile_reg[‘bp] <= 32’h00000000;
29          regfile_reg[‘si] <= 32’h00000000;
30          regfile_reg[‘di] <= 32’h00000000;
31      end
32
33      else if (phase[‘f] == 1) begin
34          regfile_reg[rg2] <= regfile_in;
35      end
36
37      else if (phase[‘r] == 1) begin
38          sr <= regfile_reg[rg1];
39          tr <= regfile_reg[rg2];
40      end
41
42  end // always @(...)
43
44 endmodule
```

0.5.7 SR

SR のソースコードをソース 12 に示す．主な仕様は以下の通りである．

入力

- レジスタファイルからのレジスタ値 (32bits)

出力

- レジスタファイルからのレジスタ値 (32bits)

記憶内容

なし

動作説明

レジスタファイルからのレジスタ値を入力し，即座に出力するだけなので，実質ただのワイヤと変わらない．これを設けた理由は図 1 中に書いてあったからで，残しておいている理由は何となく名残惜しいからである．

ソース 12　SR

```
1 module sr(
2               sr_in,
3               sr_out
4               );
5
6 ‘include ”defines.vh”
7
```

```
8   input [31:0] sr_in;
9   output [31:0] sr_out;
10
11  assign sr_out = sr_in;
12
13 endmodule
```

0.5.8 TR

TR のソースコードをソース 13 に示す．説明は，SR と全く同様なので割愛する．

ソース 13　TR

```
1  module tr(
2              tr_in,
3              tr_out
4              );
5
6  ‘include ”defines.vh”
7
8   input [31:0] tr_in;
9   output [31:0] tr_out;
10
11  assign tr_out = tr_in;
12
13 endmodule
```

0.5.9 ALU

ALU のソースコードをソース 14 に示す．主な仕様は以下の通りである．

入力

- プログラムカウンタの値 (32bits)
- 命令コード (13bits)
- SR からのデータ (32bits)
- TR からのデータ (32bits)
- sim8(8bits)
- im16(16bits)
- tttn(4bits)
- 現在のフラグレジスタの値 (4bits)

出力

- SR/TR の値,sim8,im16 に，命令コードに応じた計算を施した結果 (32bits)
- 新しいフラグ値 (4bits)
- 命令によって，メモリのライトイネーブル (1bit)

記憶内容

なし

動作説明

命令コードに応じて，0.2.3 に示した仕様に基づき計算を実行する．

ソース 14　ALU

```
1  module alu(
2                  pc,
3                  op1, op2, op3,
4                  sim8, im16,
5                  tttn,
6                  sr, tr,
7                  dr,
8                  flags_in, flags_out,
9                  memory_wren
10                 );
11
12 `include "defines.vh"
13
14   input [31:0] pc;
15   input [7:0] op1;
16   input [1:0] op2;
17   input [2:0] op3;
18   input signed [7:0] sim8;
19   input [15:0] im16;
20   input [3:0] tttn; // used only in zBcc
21   input [31:0] sr, tr;
22   output [31:0] dr;
23   input [3:0] flags_in; // {of,cf,zf,sf}
24   output [3:0] flags_out; // {of,cf,zf,sf}
25   output memory_wren;
26
27   assign {memory_wren,flags_out,dr}
28     = dr_out(pc, op1, op2, op3,
29                  sim8, im16, tttn,
30                  sr, tr,
31                  flags_in
32                  );
33
34   // Function to be used in calc operations.
35   // Returns {of,cf,zf,sf}
36   function [`flags_h:0] bi_op_flags;
37     input [31:0] sr;
38     input [31:0] tr;
39     input [32:0] result; // result = sr op tr
40     reg [3:0] flags;
41
42     begin
43       flags[`sf] = result[31]; // Not so confident...
44       flags[`zf]
```

```
45          = (result[32:0] == {33{1'b0}} ? 1 : 0);
46        flags[`cf] = result[32];
47        // of is 1 when:
48        // 1. (neg num) + (neg num) == (pos num)
49        // 2. (pos num) + (pos num) == (neg num)
50        // so, when "C = A + B",
51        // of = MSB(A)&MSB(B)&(~MSB(C))
52        // |(~MSB(A))&(~MSB(B))&MSB(C)
53        flags[`of] = (sr[31] & tr[31] & ~result[32])
54        | (~sr[31] & ~tr[31] & result[32]);
55
56        bi_op_flags = flags;
57      end
58    endfunction
59
60    function [36:0] dr_out; // 36: wren 35-32: flags_out 31-0: dr
61      input [31:0] pc;
62      input [7:0] op1;
63      input [1:0] op2;
64      input [2:0] op3;
65      input signed [7:0] sim8;
66      input [15:0] im16;
67      input [3:0] tttn; // used only in zBcc
68      input [31:0] sr, tr;
69      input [3:0] flags_in;
70      reg [32:0] result; // has flags info
71      reg [3:0] flags; // temporary var
72
73      casex ({op1,op2,op3})
74        `zLD: begin
75          dr_out = {1'b0,4'b0000, sr + $signed(sim8)};
76        end
77
78        `zST: begin
79          dr_out = {1'b1,4'b0000, sr + $signed(sim8)};
80        end
81
82        `zLIL: begin
83          if (im16[15] == 0) // pos num
84            dr_out = {1'b0,4'b0000,
85                      16'b00000000_00000000,im16};
86          else // neg num
87            dr_out = {1'b0,4'b0000,
88                      16'b11111111_11111111,im16};
89        end
90
91        `zMOV: begin
92          dr_out = {1'b0,4'b0000, sr};
93        end
94
95        `zADD: begin
96          result = tr + sr;
```

```
97          flags = bi_op_flags(sr, tr, result);
98          dr_out = {1'b0,flags, result[31:0]};
99        end
100
101       `zSUB: begin
102         result = tr - sr;
103         flags = bi_op_flags(sr, tr, result);
104         dr_out = {1'b0,flags, result[31:0]};
105       end
106
107       `zCMP: begin
108         flags[`zf] = (sr == tr ? 1 : 0);
109         if (sr < tr) begin
110           flags[`cf] = 1;
111           flags[`sf] = 1;
112           flags[`of] = 0;
113         end
114         else begin
115           flags[`cf] = 0;
116           flags[`sf] = 0;
117           flags[`of] = 0;
118         end
119         dr_out = {1'b0,flags, 32'h00000000};
120
121       end
122
123       `zAND: begin
124         dr_out = {1'b0,4'b0000, tr & sr};
125       end
126
127       `zOR: begin
128         dr_out = {1'b0,4'b0000, tr | sr};
129       end
130
131       `zXOR: begin
132         dr_out = {1'b0,4'b0000, tr ^ sr};
133       end
134
135       `zNOT: begin
136         dr_out = {1'b0,4'b0000, ~sr};
137       end
138
139       `zSLL: begin
140         flags[`of] = (tr[31] == 1 ? 1 : 0);
141         flags[`zf] = 0;
142         flags[`cf] = 0;
143         flags[`sf] = 0;
144         dr_out = {1'b0,flags, tr << $signed(sim8)};
145       end
146
147       `zSRL: begin
148         dr_out = {1'b0,4'b0000, tr >> $signed(sim8)};
```

```
149         end
150
151         `zSRA: begin
152           dr_out = {1'b0,4'b0000, $signed(tr) >>> $signed(sim8)};
153         end
154
155         `zB: begin
156           dr_out = {1'b0,4'b0000,
157                     $signed(pc) + $signed(sim8)};
158         end
159
160         `zBcc: begin
161           case (tttn)
162             `o: begin
163               if (flags_in[`of] == 1)
164                 dr_out = {1'b0,4'b0000, $signed(pc) + $signed(sim8)};
165               else
166                 dr_out = {1'b0,4'b0000, pc + 1};
167             end
168             `no: begin
169               if (flags_in[`of] == 1)
170                 dr_out = {1'b0,4'b0000, $signed(pc) + $signed(sim8)};
171               else
172                 dr_out = {1'b0,4'b0000, pc + 1};
173             end
174             `b: begin
175               if (flags_in[`cf] == 1)
176                 dr_out = {1'b0,4'b0000, $signed(pc) + $signed(sim8)};
177               else
178                 dr_out = {1'b0,4'b0000, pc + 1};
179             end
180             `nb: begin
181               if (flags_in[`cf] == 1)
182                 dr_out = {1'b0,4'b0000, $signed(pc) + $signed(sim8)};
183               else
184                 dr_out = {1'b0,4'b0000, pc + 1};
185             end
186             `e: begin
187               if (flags_in[`zf] == 1)
188                 dr_out = {1'b0,4'b0000, $signed(pc) + $signed(sim8)};
189               else
190                 dr_out = {1'b0,4'b0000, pc + 1};
191             end
192             `ne: begin
193               if (flags_in[`zf] == 1)
194                 dr_out = {1'b0,4'b0000, $signed(pc) + $signed(sim8)};
195               else
196                 dr_out = {1'b0,4'b0000, pc + 1};
197             end
198             `be: begin
199               if (flags_in[`zf] == 1 || flags_in[`cf] == 1)
200                 dr_out = {1'b0,4'b0000, $signed(pc) + $signed(sim8)};
```

```
201                 else
202                     dr_out = {1'b0,4'b0000, pc + 1};
203             end
204             `nbe: begin
205                 if (flags_in[`zf] == 1 || flags_in[`cf] == 1)
206                     dr_out = {1'b0,4'b0000, $signed(pc) + $signed(sim8)};
207                 else
208                     dr_out = {1'b0,4'b0000, pc + 1};
209             end
210             `s: begin
211                 if (flags_in[`sf] == 1)
212                     dr_out = {1'b0,4'b0000, $signed(pc) + $signed(sim8)};
213                 else
214                     dr_out = {1'b0,4'b0000, pc + 1};
215             end
216             `ns: begin
217                 if (flags_in[`sf] == 1)
218                     dr_out = {1'b0,4'b0000, $signed(pc) + $signed(sim8)};
219                 else
220                     dr_out = {1'b0,4'b0000, pc + 1};
221             end
222             `l: begin
223                 if (flags_in[`sf] ^ flags_in[`of] == 1)
224                     dr_out = {1'b0,4'b0000, $signed(pc) + $signed(sim8)};
225                 else
226                     dr_out = {1'b0,4'b0000, pc + 1};
227             end
228             `nl: begin
229                 if (flags_in[`sf] ^ flags_in[`of] == 0)
230                     dr_out = {1'b0,4'b0000, $signed(pc) + $signed(sim8)};
231                 else
232                     dr_out = {1'b0,4'b0000, pc + 1};
233             end
234             `le: begin
235                 dr_out = {1'b0,4'b0000,
236                             (((flags_in[`sf] ^ flags_in[`of]) | flags_in[`zf]) == 1
237                               ? $signed(pc[7:0]) + $signed(sim8) : pc + 1)};
238             end
239             `nle: begin
240                 if (flags_in[`sf] ^ flags_in[`of] | flags_in[`zf] == 0)
241                     dr_out = {1'b0,4'b0000, $signed(pc) + $signed(sim8)};
242                 else
243                     dr_out = {1'b0,4'b0000, pc + 1};
244             end
245           endcase
246       end
247
248       `zHLT: begin
249       end
250
251     endcase // case ({op1,op2,op3})
252 endfunction
```

```
253
254 endmodule
```

0.5.10 フラグレジスタ

フラグレジスタのソースコードをソース 15 に示す．主な仕様は以下の通りである．

入力

- 新しいフラグ値 (4bits)

出力

- 現在のフラグ値 (4bits)

記憶内容

- 現在のフラグ値 (4bits)

動作説明

フラグは，サインフラグ (SF)，ゼロフラグ (ZF)，キャリーフラグ (CF)，オーバフローフラグ (OF) の 4 種類がある．0.2.3 で説明したように，zADD，zSUB，zCMP，zSLL でそれぞれのレジスタが適切に設定される．それ以外の命令が発行されるサイクルとその直後のサイクルでは，フラグ値は意味を持たない．

X フェーズで，ALU によりフラグ値が更新される．出力は，全てのフェーズにおいて行われる．

ソース 15　フラグレジスタ

```
1  module flags(
2                  clk, rst,
3                  phase,
4                  op1, op2, op3,
5                  flags_in,
6                  flags_out
7                  );
8
9  `include "defines.vh"
10
11   input clk, rst;
12   input [`phase_h:0] phase;
13   input [7:0] op1;
14   input [1:0] op2;
15   input [2:0] op3;
16   input [`flags_h:0] flags_in;
17   output [`flags_h:0] flags_out;
18
19   reg [3:0] flags_reg;
20
21   assign flags_out = flags_reg;
```

```
22
23   always @(posedge clk or negedge rst) begin
24     if (rst == 0) begin
25       flags_reg <= 4'b0000;
26     end
27
28     else if (phase[`m] == 1) begin
29       casex ({op1,op2,op3})
30         // the cases in which flags are renewed.
31         `zADD:
32           flags_reg <= flags_in;
33         `zSUB:
34           flags_reg <= flags_in;
35         `zCMP:
36           flags_reg <= flags_in;
37         `zSLL:
38           flags_reg <= flags_in;
39       endcase
40
41     end
42
43   end // always @(...)
44
45 endmodule
```

0.5.11 DR

DR のソースコードをソース 16 に示す．主な仕様は以下の通りである．

入力

- ALU の計算結果 (32bits)

出力

- DR の値 (32bits)

記憶内容

- DR の値 (32bits)

動作説明

ALU の結果を格納する．

M フェーズにおいて，命令により以下の 3 通りの動作をする．

1. (zLD,zST) 値を，アクセスするメモリ番地としてメモリに伝える．
2. (zB,zBcc) 値を，プログラムカウンタ値としてプログラムカウンタに送る．
3. (その他) 値を DR2 に送る．

ソース 16　DR

```
1  module dr(
2                  clk, rst,
3                  phase,
4                  dr_in,
5                  dr_out
6                  );
7
8  `include "defines.vh"
9
10   input clk, rst;
11   input [`phase_h:0] phase;
12   input [31:0] dr_in;
13   output reg [31:0] dr_out;
14
15   reg [31:0] dr_reg;
16
17   always @(posedge clk or negedge rst) begin
18      if (rst == 0) begin
19         dr_reg = 32'h00000000;
20      end
21
22      else if (phase[`m] == 1) begin
23         dr_reg = dr_in;
24         dr_out = dr_reg;
25      end
26   end // always @(...)
27
28 endmodule
```

0.5.12　DR2

DR2 のソースコードをソース 17 に示す．主な仕様は以下の通りである．

入力

- DR からの値 (32bits)

出力

- DR2 の値 (32bits)

記憶内容

- DR2 の値 (32bits)

動作説明

M フェーズで，DR から値を受け取る．

W フェーズで，値をレジスタファイルに渡す．

ソース 17　DR2

```
1  module dr2(
2                  clk, rst,
3                  phase,
4                  dr2_in,
5                  dr2_out
6                  );
7
8  `include "defines.vh"
9
10   input clk, rst;
11   input [`phase_h:0] phase;
12   input [31:0] dr2_in;
13   output reg [31:0] dr2_out;
14
15   reg [31:0] dr2_reg;
16
17   always @(posedge clk or negedge rst) begin
18      if (rst == 0) begin
19         dr2_reg = 32'h00000000;
20      end
21
22      else if (phase[`w] == 1) begin
23         dr2_reg = dr2_in;
24         dr2_out = dr2_reg;
25      end
26   end // always @(...)
27
28 endmodule
```

0.5.13　MDR

MDR のソースコードをソース 18 に示す．主な仕様は以下の通りである．

入力

- メモリからのデータ (32bits)

出力

- MDR の値 (32bits)

記憶内容

- MDR の値 (32bits)

動作説明

M フェーズで，メモリからデータを受け取る．

W フェーズで，値をレジスタファイルに渡す．

ソース 18　MDR

```
1  module mdr(
2                  clk, rst,
3                  phase,
4                  mdr_in,
5                  mdr_out
6                  );
7
8  `include "defines.vh"
9
10   input clk, rst;
11   input [`phase_h:0] phase;
12   input [31:0] mdr_in;
13   output reg [31:0] mdr_out;
14
15   reg [31:0] mdr_reg;
16
17   always @(posedge clk or negedge rst) begin
18      if (rst == 0) begin
19         mdr_reg = 32'h00000000;
20      end
21
22      else if (phase[`w] == 1) begin
23         mdr_reg = mdr_in;
24         mdr_out = mdr_reg;
25      end
26   end // always @(...)
27
28 endmodule
```

0.5.14　各種セレクタ

パーツの中には，命令やフェーズによって入力先を切り替える必要のあるものがある．そのために，ソース 19,20,21 で示された 3 種類のセレクタを用意した．それぞれの役割は，以下の通りである．

ソース 19

入力は DR2,MDR を取り，レジスタファイルに書き戻す値を選択する．zLD 命令では MDR の値を，それ以外の命令では DR2 の値を選択する．

ソース 20

入力はプログラムカウンタの値と DR を取り，アクセスするメモリ番地を選択する．W,F,R フェーズではプログラムカウンタの指す番地に読み取りアクセスさせ，X,M フェーズでは DR の指す番地に読み書きアクセスさせる．

ソース 21

入力はプログラムカウンタの値と DR を取り，新しいプログラムカウンタの値を選択する．命令が zB,zBcc のときは DR で指定された値を，zHLT のときは現在の値そのままを，それ以外の時は現在の値 +1 を選択する．

ソース 19　セレクタ 1

```
1  module selecter_regfile(
2                               op1, op2, op3,
3                               dr2, mdr,
4                               selected
5                               );
6
7  `include "defines.vh"
8
9    input [7:0] op1;
10   input [1:0] op2;
11   input [2:0] op3;
12   input [31:0] dr2, mdr;
13   output [31:0] selected;
14
15   assign selected
16     = selecter_regfile_out(op1, op2, op3, dr2, mdr);
17
18   function [31:0] selecter_regfile_out;
19     input [7:0] op1;
20     input [1:0] op2;
21     input [2:0] op3;
22     input [31:0] dr2, mdr;
23
24     begin
25       casex ({op1,op2,op3})
26         `zLD:
27           selecter_regfile_out = mdr;
28         default:
29           selecter_regfile_out = dr2;
30       endcase
31     end
32   endfunction
33
34 endmodule
```

ソース 20　セレクタ 2

```
1  module selecter_op_data(
2                               clk, rst, phase,
3                               pc, dr,
4                               selected
5                               );
6
7  `include "defines.vh"
8
9    input clk, rst;
10   input [`phase_h:0] phase;
11   input [4:0] pc, dr;
12   output [4:0] selected;
13
14   assign selected = select_out(phase, pc, dr);
```

```
15
16   function [4:0] select_out;
17     input [`phase_h:0] phase;
18     input [4:0] pc, dr;
19
20     if (phase[`w] == 1
21         || phase[`f] == 1
22         || phase[`r] == 1) begin
23       select_out = pc;
24     end
25
26     else if (phase[`x] == 1
27                 || phase[`m] == 1) begin
28       select_out = dr;
29     end
30
31   endfunction
32
33 endmodule
```

ソース 21　セレクタ 3

```
1  module selecter_pc_inc_jmp(
2                                 op1, op2, op3,
3                                 pc, dr2,
4                                 selected
5                                 );
6
7  `include "defines.vh"
8
9    input [7:0] op1;
10   input [1:0] op2;
11   input [2:0] op3;
12   input [31:0] pc, dr;
13   output [31:0] selected;
14
15   assign selected
16     = selecter_pc_inc_jmp_out(op1, op2, op3, pc, dr);
17
18   function [31:0] selecter_pc_inc_jmp_out;
19     input [7:0] op1;
20     input [1:0] op2;
21     input [2:0] op3;
22     input [31:0] pc, dr;
23
24     begin
25       casex ({op1,op2,op3})
26         `zB:
27           selecter_pc_inc_jmp_out = dr;
28         `zBcc:
29           selecter_pc_inc_jmp_out = dr;
30         `zHLT:
```

```
31              selecter_pc_inc_jmp_out = pc;
32            default:
33              selecter_pc_inc_jmp_out = pc + 1; // Main memory is
34                                                // dword(32bit) addressing.
35        endcase
36      end
37    endfunction
38
39 endmodule
```

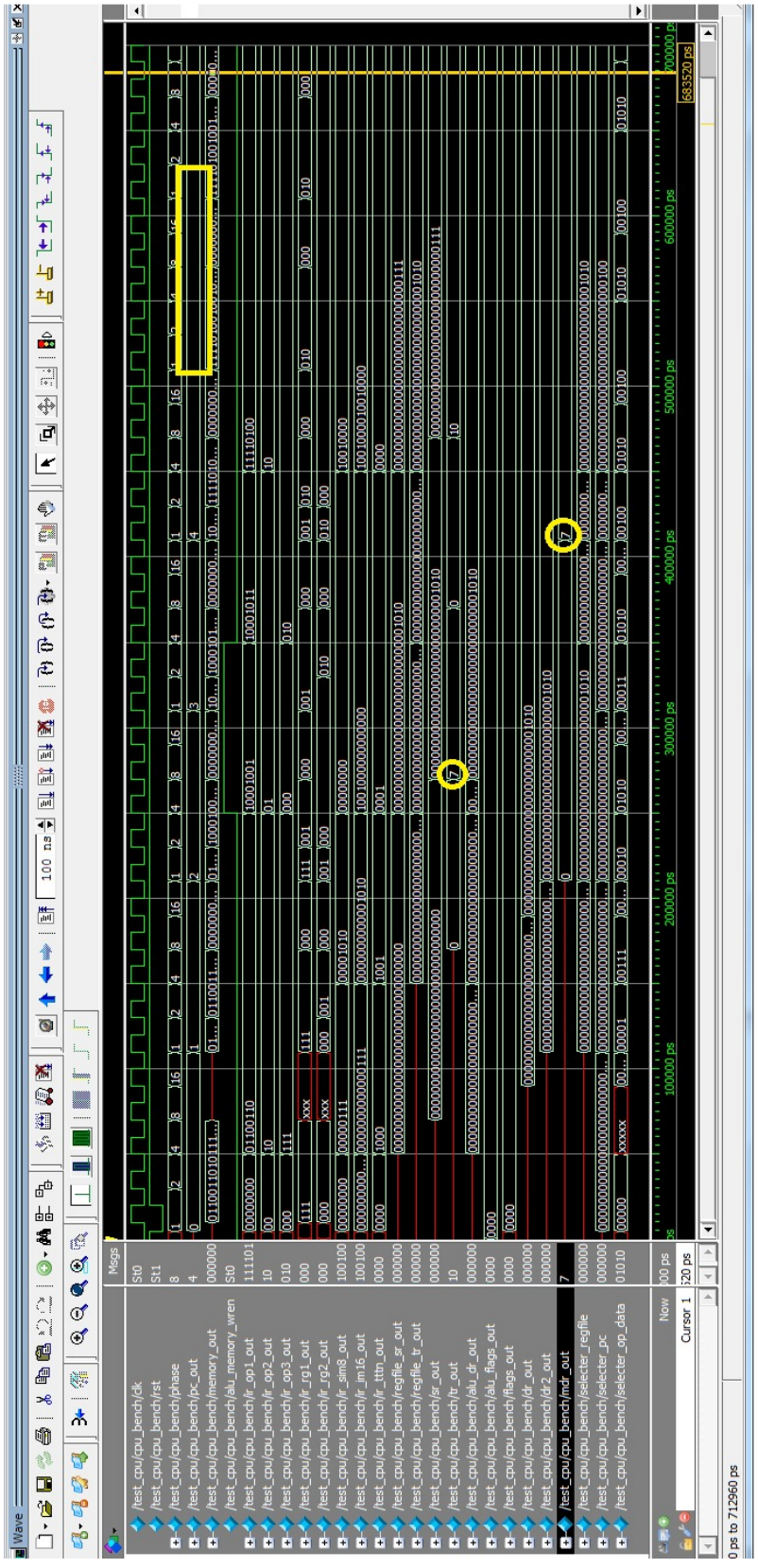

図 3　ソース 1 のシミュレート結果

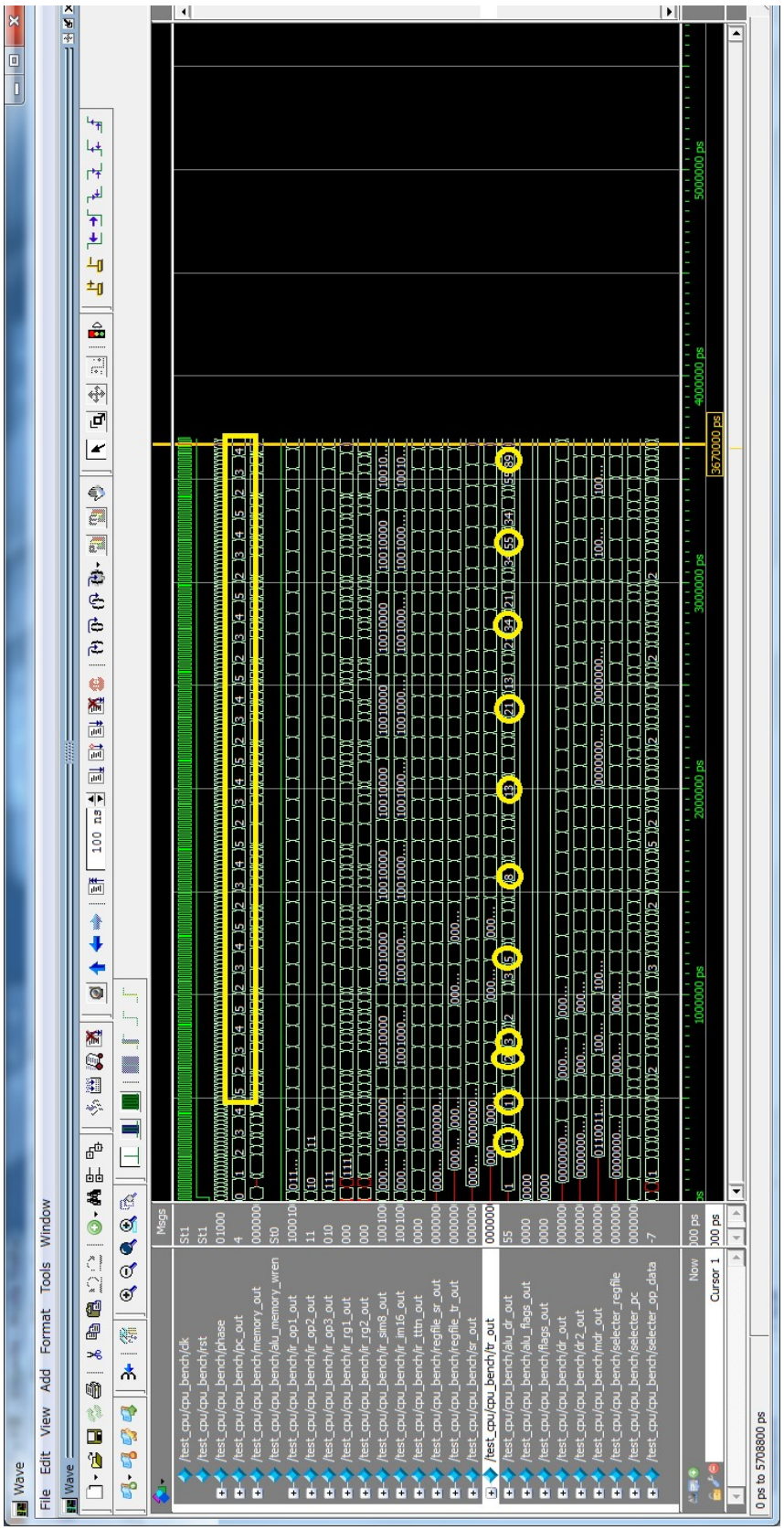

図 4　ソース 2 のシミュレート結果

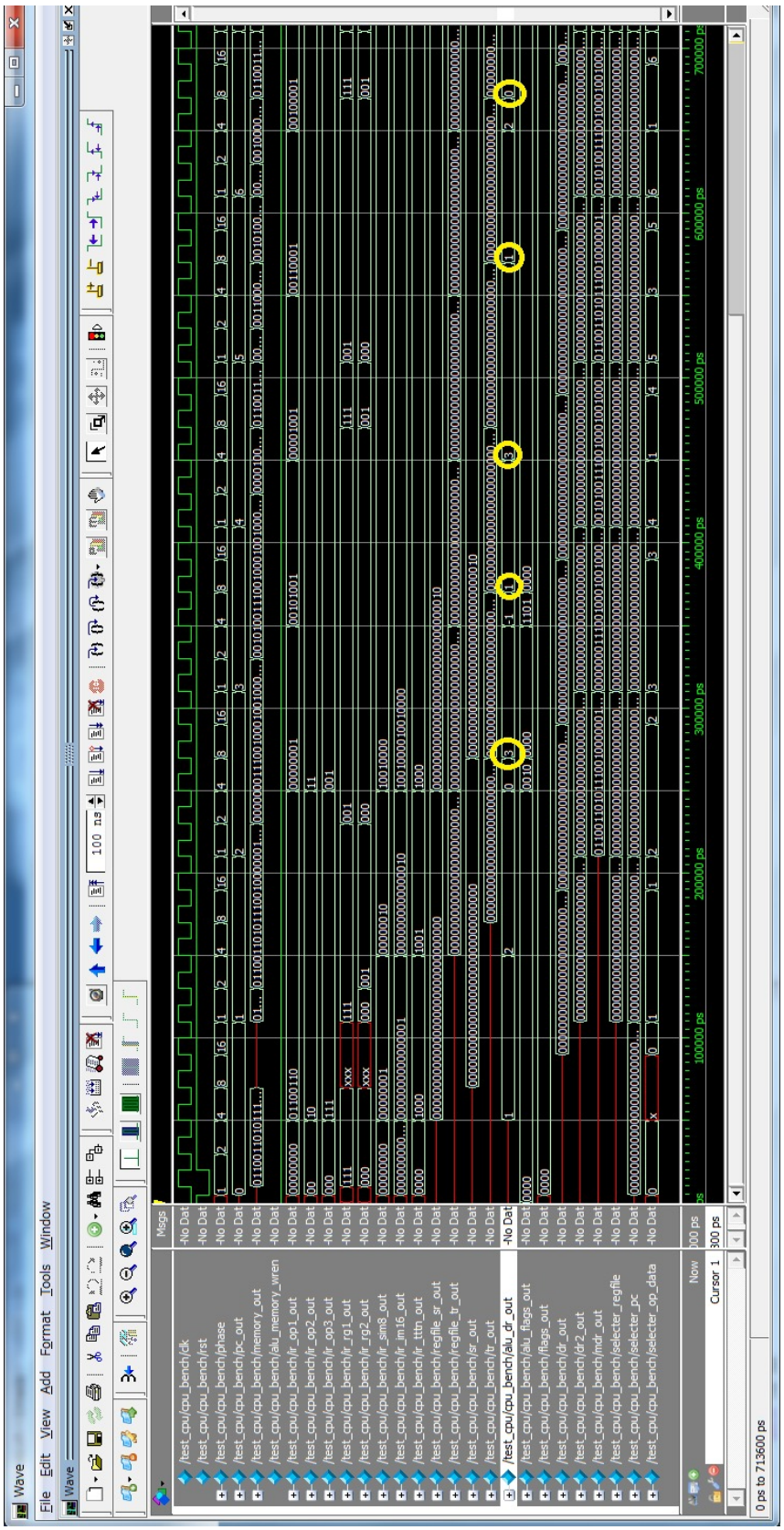

図 5　ソース 3 のシミュレート結果

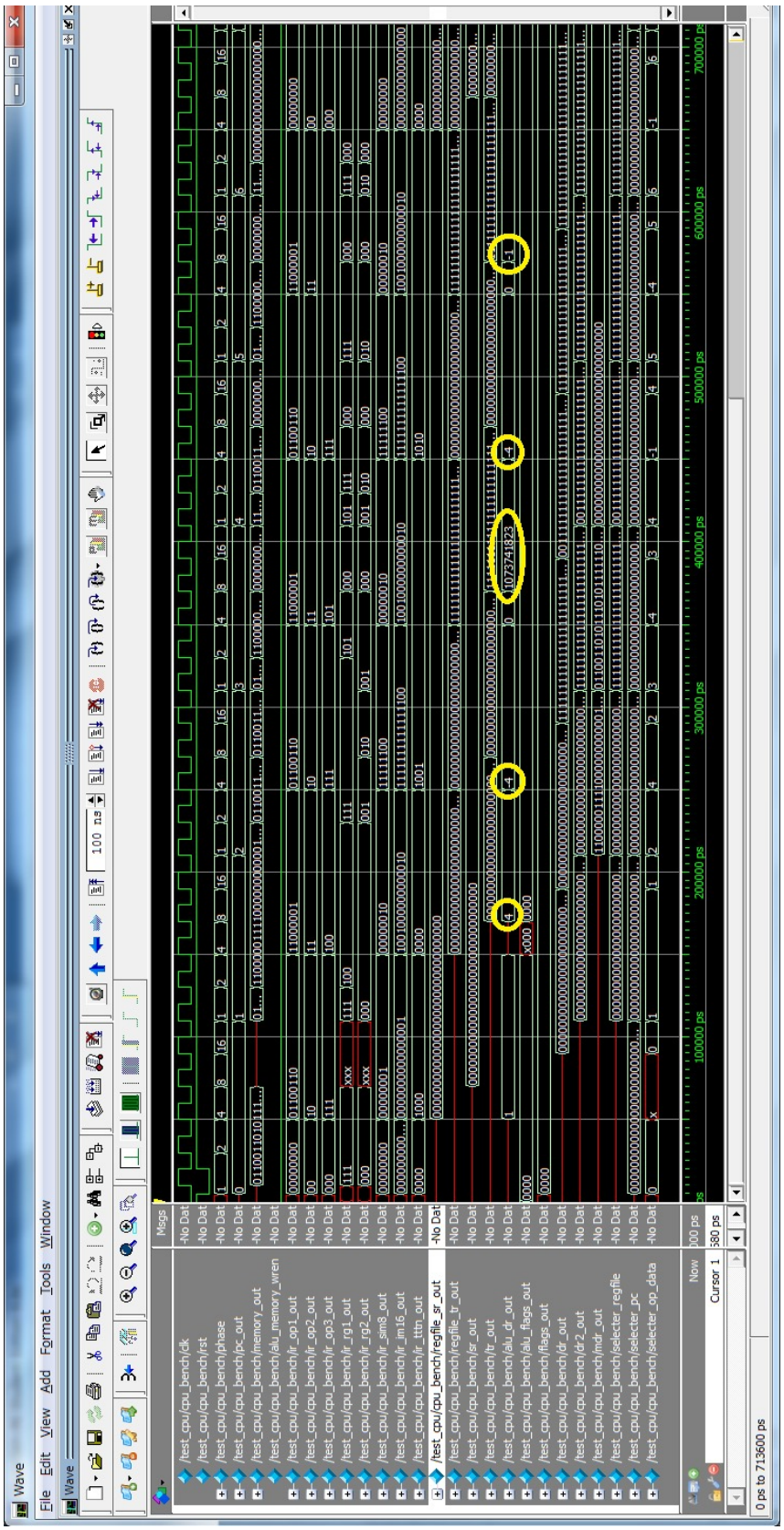

図 6　ソース 4 のシミュレート結果

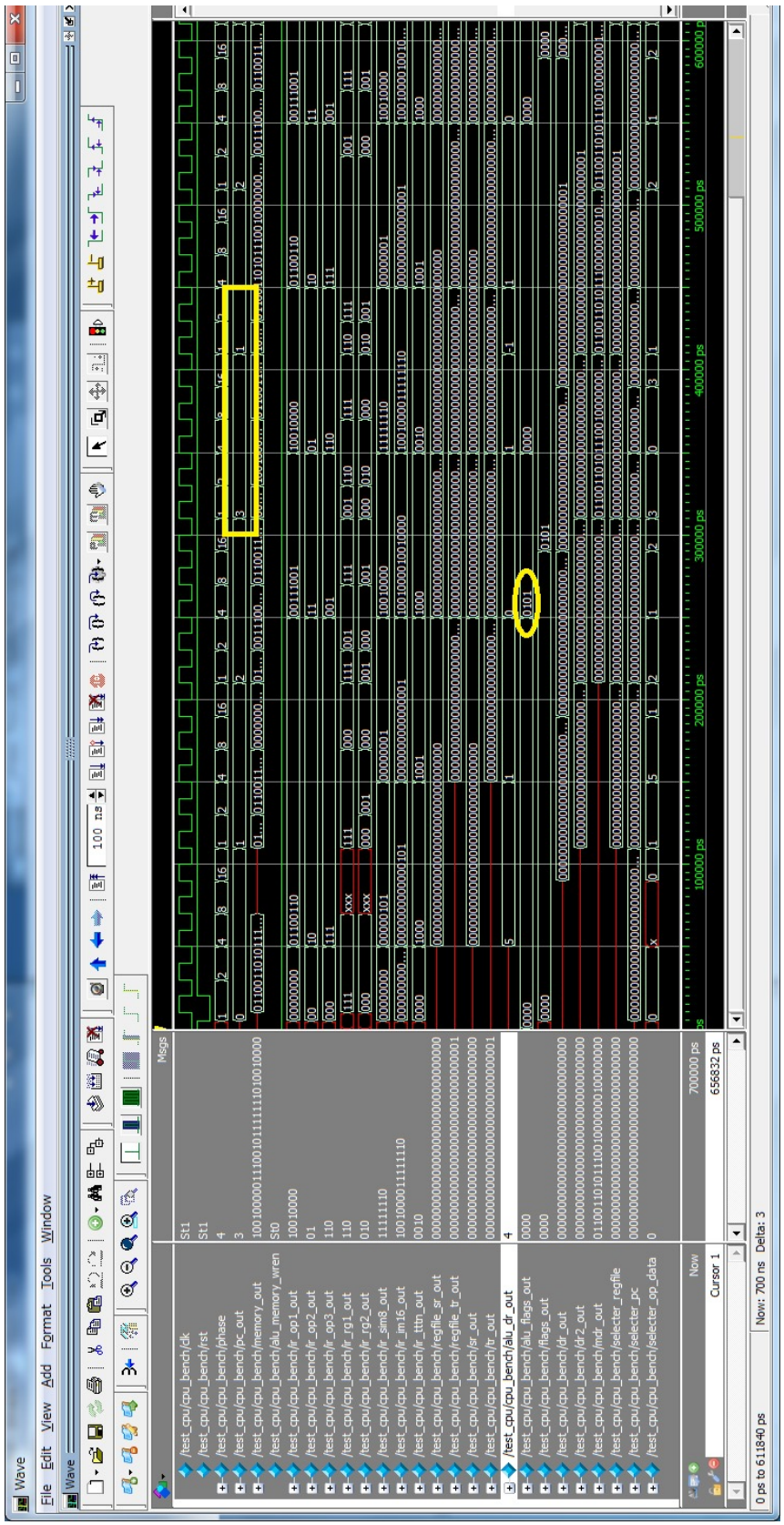

図 7　ソース 5 のシミュレート結果

参考

[1] Wikipedia. http://ja.wikipedia.org/wiki/.

[2] マイクロプロセッサの設計と実装. http://www.mtl.t.u-tokyo.ac.jp/~jikken/cpu/wiki/index.php?FrontPage.