情報工学実験 **IV**

第 **5** 回： **16** ビット **RISC** マシンのパイプライン化

2019 年 1 月 8 日

柴田 裕一郎（shibata@cis.nagasaki-u.ac.jp）

1　パイプライン処理の基本

現代プロセッサにおける高速化手法の基本はパイプライン処理です．これは工場におけるベルトコンベア式の流れ作業に似ています．命令の実行を複数のステップ（ステージ）に分け，各ステージごとに専門の担当者を配置します．各ステージでは前のステージから作業の結果を受取り，自分のステージの作業を行って結果を次のステージに渡します．

これまで設計してきた RISC16 では，LD/ST 系の命令では 4 状態（「命令フェッチ」「レジスタフェッチ」「実行」「書き戻し」），それ以外の命令では「書き戻し」以外の 3 状態で処理を行っていました．これは，命令を 1 個実行するのに 3 または 4 クロックかかることを意味しています．パイプライン処理では，これらの各状態をステージと考えて流れ作業を行います．例えば「命令フェッチ」のステージでは毎クロックひたすら命令をフェッチし続け，フェッチした命令を次の「レジスタフェッチ」のステージに渡します．「レジスタフェッチ」のステージでは，「命令フェッチ」のステージから渡された命令のレジスタフィールドを見て，必要なレジスタの値を読み出し，それらを次のステージに渡します．そして「実行」のステージでは毎クロックひたすら ALU で計算を行う…という具合です．

ポイントは各ステージの処理が命令間でオーバーラップしながら動作することです．1 つの命令がフェッチされてから実行が完了するまでの時間（レイテンシ: latency）は変らない（むしろステージ間での処理の受け渡しのために悪化する可能性もある）のですが，1 クロックごとに新しい命令がフェッチされて実行されるので，単位時間あたりの命令の実行数（スループット: throughput）は向上します．

2　効率のよいパイプライン処理のために

効率の良いパイプライン処理を設計するには，以下の点に注意が必要です．

（1）各ステージの処理時間をなるべく等しくする
例えば「レジスタフェッチ」のステージの処理だけ早く終わっても，「命令フェッチ」のステージの処理が終わらないことには，「レジスタフェッチ」では次にやるべき処理がないので，待ち状態となります．各ステージの処理時間がなるべく均等になるように分割することが重要になります．

（2）複数のステージで同じハードウェア資源を使用しないようにする
パイプライン処理では，各ステージは同時に異なる命令に対して処理を行います．したがって，例えば「命令フェッチ」のステージで ALU を使用し「実行」のステージでも ALU を使用するとなると，「命令フェッチ」と「実行」はオーバーラップして実行することができなくなります（図 1)．これは命令実行のスループットを低下させてしまいます．このように，資源の競合によってパイプラインの動作に問題が生じることを**構造ハザード（structual hazard）**と呼びます．また，ハザードのためにパイプラインが一時的に止まることをパイプラインの**ストール（stall）**と呼びます．

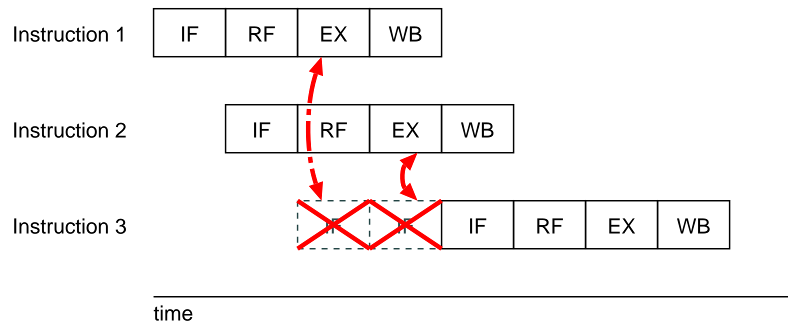

図 1 「命令フェッチ（IF)」と「実行（EX)」の両方で ALU を使う場合のパイプライン

3 RISC16p の設計

それでは RISC16 のパイプライン化版を設計してみましょう．以降，このパイプライン化された 16 ビット RISC を RISC16p と呼ぶことにします．命令セットについては RISC16 のものを引き継ぎますが，効率よくパイプライン処理を行うために，構成や動作にいくつかの変更を加えます．

3.1 パイプラインステージの設計

前述のように，効率の良いパイプライン処理のためには各ステージの処理時間をなるべく等しくする必要があります．RISC16 では LD/ST 系の命令以外では「命令フェッチ」「レジスタフェッチ」「実行」の 3 段階で処理を行っていました．最近ではキャッシュをチップ内部に持たせることが可能なため，キャッシュにヒットすればメモリへのアクセスはかなり高速です[*1]．したがって，この中では命令のオペコードに従って ALU を制御し結果をレジスタへ書込む「実行」のステージの処理時間が一番長くなります．そこで，「実行」のステージを ALU における計算のステージとレジスタの書込みへのステージに分離することにします．この結果，RISC16p のパイプラインは以下の 4 ステージとします．

（1）命令フェッチ（IF: Instruction Fetch）
　メモリから命令を取ってくる．同時に PC（プログラムカウンタ）に 2 を加える．
（2）レジスタフェッチ（RF: Register Fetch）
　レジスタファイルから必要なデータを読み出す．
（3）実行（EX: Execution）
　ALU を使って計算処理を行う．LD/ST 系の命令ではメモリへのアクセスを行う．
（4）書き戻し（WB: Write Back）
　計算結果を指定されたレジスタに書き込む．

LD 系の命令を含めてすべての命令が上記の 4 つのステージで実行されることになるので，より単純にパイプラインを構成できます．ちなみに，この 4 段パイプラインというのは，必ずしも一般的な構成というわけでは

[*1] 実際にキャッシュを装備するためにはそれなりの制御回路を持たせる必要がありますが，やや複雑なので本実験では扱いません．興味のある方は参考書等をご覧下さい．

ありません．例えば，典型的な RISC プロセッサアーキテクチャの MIPS32 は 5 段のパイプライン構成です．先進的なプロセッサの中には 20 段以上のパイプライン構造をもつものもあります．

3.2 構造ハザードの除去

次に，これらのステージ間で競合するハードウェア資源を考えてみます．以下の 3 つのハードウェアが競合を起こしてしまうことが分かります．

- ALU
  前回設計した RISC16 では，「実行（EX）」状態での計算処理に ALU を使ったうえ「命令フェッチ（IF）」の状態でも PC に 2 を足すために ALU を用いていました．この資源競合を解消するためには，IF ステージに PC に 2 を足すための専用の加算器が必要になります．
- メモリ
  IF ステージでは命令をフェッチするために毎クロックメモリにアクセスします．一方，EX ステージでは LD/ST 系の命令のときには，データを読み書きするためにメモリにアクセスするため競合が起きます．命令メモリとデータメモリを分離して同時にアクセスできるようにすれば競合は解消されますが，主記憶全体でこれをやろうとするとコストの面から実装が難しくなります．そこで一般には，チップ内部のキャッシュを命令キャッシュとデータキャッシュに分離し，外部の 2 次キャッシュや主記憶は共有する方法がとられます．このように命令とデータのメモリ階層を分離して同時にアクセスできる構成はハーバードアーキテクチャ（Harvard architecture）と呼ばれています．
  具体的には，アドレスバスとデータバスを命令用とデータ用の 2 系統用意して，それぞれ別々のキャッシュに接続します．また，メモリ読み出し用の制御信号（oe）も命令用とデータ用に分離してやります．なお，命令メモリ（キャッシュ）には書き込みをしませんので，書き込みの制御は必要ありません．
- レジスタファイル
  RF ステージではレジスタファイルから読み出しを行いますが，同時に WB ステージでは書き込みを行います．今までは 2 つのデータの読み出しは同時に行えるようになっていましたが，2 つのアドレス入力のうち片方は書き込みのアドレス入力を兼ねていました．そこでこれらを分離して，2 つのレジスタの読み出しと 1 つのレジスタへの書き込みを同時に行える 3 ポートのレジスタファイルに変更してやります．レジスタファイルのように比較的小規模なメモリであれば，この程度のポート数の増加は致命的なコストの増加にはつながりません．

このように構造ハザードの除去には，競合する資源の多重化などハードウェアコストの増加が要求されます．したがって，設計者は常に性能とハードウェアコストを秤にかけながら，ハザードを除去するのかパイプラインをストールさせるのかを選択していく必要があります．今回の例では上記のように比較的小さなコストですべての構造ハザード除去することができます．

3.3 RISC16p の構成と動作

RISC16p のパイプラインの構成を図 2 に示します．また，表 1 に制御信号線の役割をまとめて示します．各ステージでは，前のステージからデータを受取り処理を行います．結果は，必ず FF（Flip-Flop）に書かれ次のステージに渡されます．もしステージ間のデータの受け渡しに FF を使用しないと，命令をオーバラップ

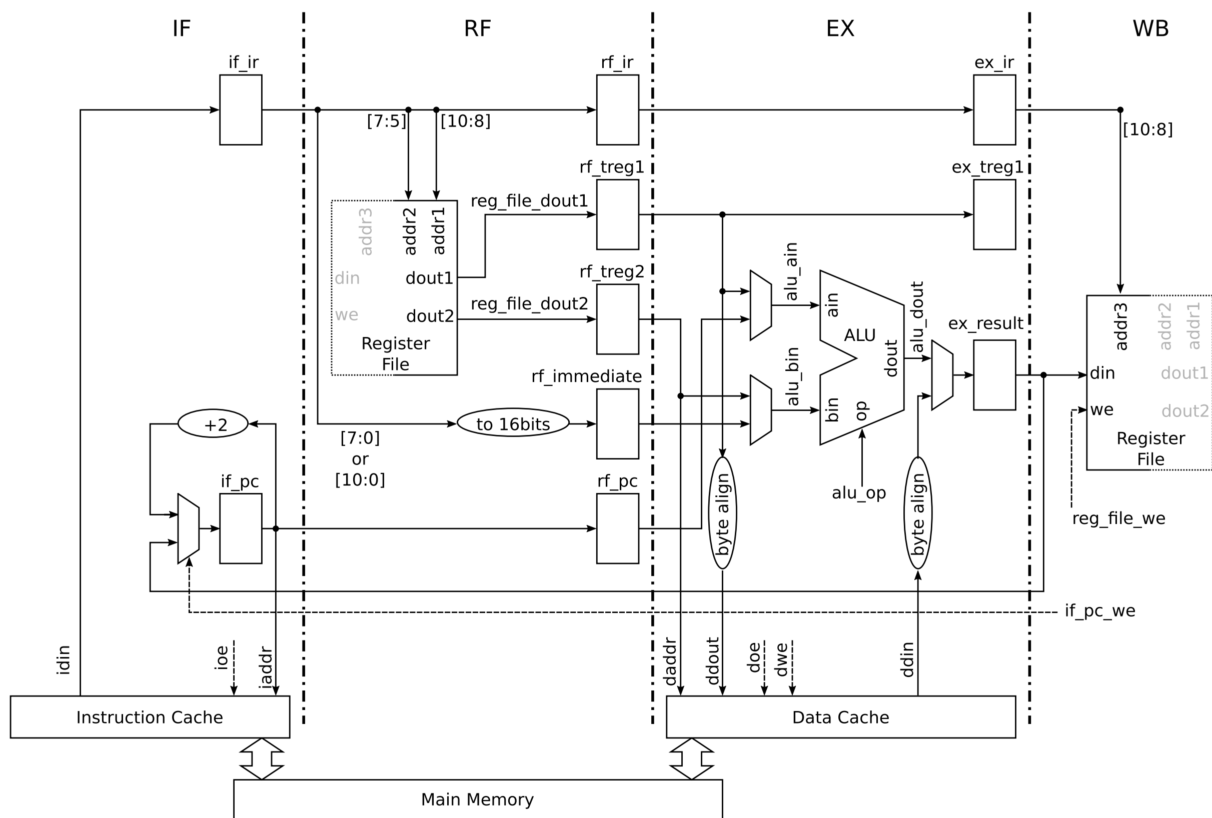

図 2　RISC16p のパイプラインの構成

しながら毎クロック処理を行うことができません．また，動作中のある瞬間を見ると，各ステージは別々の命令を処理することになるので，たとえば IR（命令レジスタ）も，IF ステージ用（`if_ir`），RF ステージ用（`rf_ir`）および EX ステージ用（`ex_ir`）ではそれぞれ別のものが用意されています．

以下に，各ステージの処理を簡単にまとめます．

（1）IF ステージ

プログラムカウンタ（`if_pc`）を命令メモリ用アドレスバス（`if_addr`）に繋いで読み出し制御信号（ioe）を ‘1’ にします．これで命令が命令レジスタ（`if_ir`）にフェッチされます．同時に専用の加算器を使って PC の値に 2 を足します．また，後述するように分岐命令の場合には EX ステージで飛び先のアドレスが計算されたのち，WB ステージで `if_pc_we` 信号（表 1 参照）が ‘1’ になります．この場合には，`ex_result` レジスタの値（計算された飛び先のアドレス）が `if_pc` にセットされます．

（2）RF ステージ

IF ステージから現在のプログラムカウンタとフェッチされた命令（`if_pc` と `if_ir`）を受取り，命令レジスタのレジスタ番号のフィールド（`if_ir[10:8]` と `if_ir[7:5]`）を使ってレジスタファイルからオペランドデータを読み出します．これらはレジスタ `rf_treg1` と `rf_treg2` に書き込みまれます．また，次の EX ステージの実行時間を短くするために，即値（イミディエイト）フィールド（`if_ir[7:0]`，ただし JUMP では `if_ir[10:0]`）も読み出して即値用のレジスタ `rf_immediate` に書き込みます．このとき，ADDI 命令や分岐命令，JUMP 命令の場合には符号拡張を行い，それ以外の命令ではゼロ拡

表 1　RISC16p の主な制御信号線

信号名	タイプ	ステージ	役割
ioe	出力	IF	データメモリ読み出し
doe	出力	EX	データメモリ読み出し
dwe	出力	EX	データメモリ書き込み
reg_file_we	内部信号	WB	レジスタファイル書き込み（アクティブ High）
if_pc_we	内部信号	WB	プログラムカウンタに飛び先番地を書き込み（アクティブ High）

張して 16 ビットにします．命令によっては rf_treg2 や rf_immediate に値を読み出す必要がない場合もありますが，読み出しても害はないので常に読み出します．if_pc と if_ir の内容はそのまま rf_pc と rf_ir に代入して EX ステージへ渡します．

（3）EX ステージ

RF ステージから受け取ったデータを ALU で処理します．また，LD/ST 系の命令の場合にはデータメモリにアクセスを行います．演算結果を ex_result に書き込む点以外は，パイプライン化する前と処理内容はあまり変りません．WB ステージには，rf_ir と rf_treg1 の内容をそのまま ex_ir と ex_treg1 に代入して渡します．

（4）WB ステージ

EX ステージから受け取った命令の内容（ex_ir）を見て，レジスタファイルに書き戻しを行う必要があるかを判断します．分岐命令，JUMP 命令，ST 命令，SBU 命令，NOP 命令では書き戻しの必要がありませんが，それ以外の命令では reg_file_we を ‘1’ にして ex_result の値を ex_ir[10:7] で指定されたレジスタへ書き込みます．レジスタファイルは RF ステージにありますが，図 2 では便宜上レジスタファイルの読み出し部分と書き込み部分を分離して示しています．また，分岐命令では ex_treg1 レジスタの値に基づき分岐が成立するかどうかを判断します．分岐が成立した場合には，if_pc_we を ‘1’ にします．こうすることで ex_result レジスタに入っている飛び先番地が if_pc にセットされます．もちろん JUMP 命令の場合には ex_treg1 レジスタの値に関わらず常に if_pc_we を ‘1’ にします．

4　パイプライン動作の例

それでは以下のような簡単なプログラムを実行して動作を確認してみます．

```
@00 00001001 00000001 // LLI  r1,   #1
@02 00001010 00000010 // LLI  r2,   #2
@04 00001011 00000011 // LLI  r3,   #3
@06 00100001 00000010 // ADDI r1,   #2
@08 00000010 01000100 // ADD  r2,   r2
```

結果は以下のようになります．レジスタファイルの値がいつどのように変化するかに注目してください．

```
==== clock: 0 ====
 if_pc:0000 if_ir:0000000000000000
 rf_pc:0000 rf_ir:0000000000000000 rf_treg1:0000 rf_treg2:0000 rf_immediate:0000
 ex_ir:0000000000000000 ex_result:0000
```

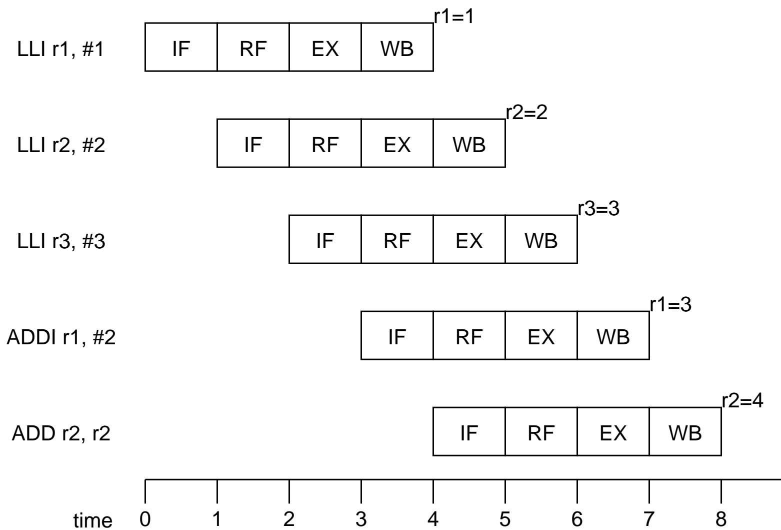

図 3　サンプルプログラムのパイプライン動作

```
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:0000 idin:0901 ioe:1
 alu_ain:0000 alu_bin:0000 alu_op:0000 reg_file_we:0 if_pc_we:0 led:000000
 regs: 0000 0000 0000 0000 0000 0000 0000 0000

==== clock: 1 ====
 if_pc:0002 if_ir:0000100100000001
 rf_pc:0000 rf_ir:0000000000000000 rf_treg1:0000 rf_treg2:0000 rf_immediate:0000
 ex_ir:0000000000000000 ex_result:0000
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:0002 idin:0a02 ioe:1
 alu_ain:0000 alu_bin:0000 alu_op:0000 reg_file_we:0 if_pc_we:0 led:000000
 regs: 0000 0000 0000 0000 0000 0000 0000 0000

==== clock: 2 ====
 if_pc:0004 if_ir:0000101000000010
 rf_pc:0002 rf_ir:0000100100000001 rf_treg1:0000 rf_treg2:0000 rf_immediate:0001
 ex_ir:0000000000000000 ex_result:0000
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:0004 idin:0b03 ioe:1
 alu_ain:0000 alu_bin:0001 alu_op:0001 reg_file_we:0 if_pc_we:0 led:000000
 regs: 0000 0000 0000 0000 0000 0000 0000 0000

==== clock: 3 ====
 if_pc:0006 if_ir:0000101100000011
 rf_pc:0004 rf_ir:0000101000000010 rf_treg1:0000 rf_treg2:0000 rf_immediate:0002
 ex_ir:0000100100000001 ex_result:0001
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:0006 idin:2102 ioe:1
 alu_ain:0000 alu_bin:0002 alu_op:0001 reg_file_we:1 if_pc_we:0 led:000000
```

```
 regs: 0000 0000 0000 0000 0000 0000 0000 0000

==== clock: 4 ====
 if_pc:0008 if_ir:0010000100000010
 rf_pc:0006 rf_ir:0000101100000011 rf_treg1:0000 rf_treg2:0000 rf_immediate:0003
 ex_ir:0000101000000010 ex_result:0002
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:0008 idin:0244 ioe:1
 alu_ain:0000 alu_bin:0003 alu_op:0001 reg_file_we:1 if_pc_we:0 led:000000
 regs: 0000 0001 0000 0000 0000 0000 0000 0000

==== clock: 5 ====
 if_pc:000a if_ir:0000001001000100
 rf_pc:0008 rf_ir:0010000100000010 rf_treg1:0001 rf_treg2:0000 rf_immediate:0002
 ex_ir:0000101100000011 ex_result:0003
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:000a idin:0000 ioe:1
 alu_ain:0001 alu_bin:0002 alu_op:0100 reg_file_we:1 if_pc_we:0 led:000000
 regs: 0000 0001 0002 0000 0000 0000 0000 0000

==== clock: 6 ====
 if_pc:000c if_ir:0000000000000000
 rf_pc:000a rf_ir:0000001001000100 rf_treg1:0002 rf_treg2:0002 rf_immediate:0044
 ex_ir:0010000100000010 ex_result:0003
 daddr:0002 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:000c idin:0000 ioe:1
 alu_ain:0002 alu_bin:0002 alu_op:0100 reg_file_we:1 if_pc_we:0 led:000000
 regs: 0000 0001 0002 0003 0000 0000 0000 0000

==== clock: 7 ====
 if_pc:000e if_ir:0000000000000000
 rf_pc:000c rf_ir:0000000000000000 rf_treg1:0000 rf_treg2:0000 rf_immediate:0000
 ex_ir:0000001001000100 ex_result:0004
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:000e idin:0000 ioe:1
 alu_ain:0000 alu_bin:0000 alu_op:0000 reg_file_we:1 if_pc_we:0 led:000000
 regs: 0000 0003 0002 0003 0000 0000 0000 0000

==== clock: 8 ====
 if_pc:0010 if_ir:0000000000000000
 rf_pc:000e rf_ir:0000000000000000 rf_treg1:0000 rf_treg2:0000 rf_immediate:0000
 ex_ir:0000000000000000 ex_result:0000
 daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
 iaddr:0010 idin:0000 ioe:1
 alu_ain:0000 alu_bin:0000 alu_op:0000 reg_file_we:0 if_pc_we:0 led:000000
 regs: 0000 0003 0004 0003 0000 0000 0000 0000

==== clock: 9 ====
 if_pc:0012 if_ir:0000000000000000
```

```
rf_pc:0010 rf_ir:0000000000000000 rf_treg1:0000 rf_treg2:0000 rf_immediate:0000
ex_ir:0000000000000000 ex_result:0000
daddr:0000 ddin:xxxx ddout:xxxx doe:0 dwe:0
iaddr:0012 idin:xxxx ioe:1
alu_ain:0000 alu_bin:0000 alu_op:0000 reg_file_we:0 if_pc_we:0 led:000000
regs: 0000 0003 0004 0003 0000 0000 0000 0000
```

毎クロック新しい命令がフェッチされ実行されていく様子が分かると思います．クロック 4 の時に最初の命令（LLI r1, #1）の実行が完了して r1 に 1 がセットされています．その後は毎クロック新しい命令の実行が完了し，次々と各レジスタに結果が設定されています．この様子を図 3 に示します．

5 risc16p.sv のソースコード

```
1  ‘default_nettype none
2
3  ‘define ALU_THROUGH_AIN  4’b0000
4  ...
5
6  module risc16p
7    (
8     input wire          clk,
9     input wire          rst,
10    input wire [15:0]   ddin,
11    output logic [15:0] ddout,
12    output wire [15:0]  daddr,
13    output logic        doe,
14    output logic        dwe,
15    input wire [15:0]   idin,
16    output wire [15:0]  iaddr,
17    output wire         ioe
18    );
19
20    reg [15:0]   if_pc, if_ir;
21    reg [15:0]   rf_pc, rf_ir, rf_immediate, rf_treg1, rf_treg2, ex_treg1;
22    reg [15:0]   ex_ir, ex_result;
23    logic [15:0] alu_ain, alu_bin, alu_dout;
24    logic [3:0]  alu_op;
25    logic [15:0] reg_file_dout1, reg_file_dout2;
26    logic        reg_file_we, if_pc_we;
27
28    // IF (Instruction Fetch) stage
29    always_ff @(posedge clk) begin
30       if (rst)
31         if_ir <= 16’d0;
32       else
33         if_ir <= idin;
34    end
35
36    always_ff @(posedge clk) begin
37       if (rst)
38         if_pc <= 16’d0;
39       else begin
40          if (if_pc_we)
41            if_pc <= ex_result;
```

```
42          else
43            if_pc <= if_pc + 16'd2;
44       end
45    end
46
47    assign ioe = 1'b1;
48    assign iaddr = if_pc;
49
50    // RF (Register Fetch) stage
51    reg_file reg_file_inst
52      (
53       .clk(clk),
54       .rst(rst),
55       .addr1(if_ir[10:8]),
56       .addr2(if_ir[7:5]),
57       .addr3(ex_ir[10:8]),
58       .din(ex_result),
59       .dout1(reg_file_dout1),
60       .dout2(reg_file_dout2),
61       .we(reg_file_we)
62       );
63
64    always_ff @(posedge clk) begin
65       if (rst)
66         rf_ir <= 16'd0;
67       else
68         rf_ir <= if_ir;
69    end
70
71    always_ff @(posedge clk) begin
72       if (rst)
73         rf_treg1 <= 16'd0;
74       else
75         rf_treg1 <= reg_file_dout1;
76    end
77
78    always_ff @(posedge clk) begin
79       if (rst)
80         rf_treg2 <= 16'd0;
81       else
82         rf_treg2 <= reg_file_dout2;
83    end
84
85    always_ff @(posedge clk) begin
86       if (rst)
87         rf_immediate <= 16'd0;
88       else begin
89          if (if_ir[15] == 1'b0) begin
90         if (if_ir[14:11] == `ALU_ADD) begin
91               if (if_ir[7] == 1'b0)
92          rf_immediate <= {8'h00, if_ir[7:0]};
93               else
94          rf_immediate <= {8'hff, if_ir[7:0]};
95         end
96         else
97           rf_immediate <= {8'h00, if_ir[7:0]};
98          end
```

```
99        else begin
100          if (if_ir[14] == 1'b0) begin
101            if (if_ir[7] == 1'b0)
102              rf_immediate <= {8'h00, if_ir[7:0]};
103            else
104              rf_immediate <= {8'hff, if_ir[7:0]};
105          end
106          else begin
107            if (if_ir[10] == 1'b0)
108              rf_immediate <= {5'h00, if_ir[10:0]};
109            else
110              rf_immediate <= {5'h1f, if_ir[10:0]};
111          end
112        end
113      end
114    end
115
116    always_ff @(posedge clk) begin
117      if (rst)
118        rf_pc <= 16'd0;
119      else
120        rf_pc <= if_pc;
121    end
122
123    // EX (Execution) stage
124    alu16 alu16_inst
125      (
126       .ain(alu_ain),
127       .bin(alu_bin),
128       .op(alu_op),
129       .dout(alu_dout)
130       );
131
132    always_ff @(posedge clk) begin
133      if (rst)
134        ex_ir <= 16'd0;
135      else
136        ex_ir <= rf_ir;
137    end
138
139    always_ff @(posedge clk) begin
140      if (rst)
141        ex_treg1 <= 16'd0;
142      else
143        ex_treg1 <= rf_treg1;
144    end
145
146    always_ff @(posedge clk) begin
147      if (rst)
148        ex_result <= 16'd0;
149      else begin
150        if (rf_ir[15:11] == 5'd0 && rf_ir[4] == 1'b1)
151          ex_result <= ddin;
152        else
153          ex_result <= alu_dout;
154      end
155    end
```

```
156
157    assign daddr = rf_treg2;
158
159    always_comb begin
160       if (rf_ir[15] == 1'b0) begin
161          if (rf_ir[14:11] == 4'h0) begin // Register type
162             if (rf_ir[4] == 1'b0) begin
163                alu_ain <= rf_treg1;
164                alu_bin <= rf_treg2;
165                alu_op <= rf_ir[3:0];
166                ddout <= 16'dx;
167                doe  <= 1'b0;
168                dwe  <= 1'b0;
169             end
170             else begin
171                ...
172       end
173    end
174
175    // WB (Write Back) stage
176    always_comb begin
177       if (ex_ir[15] == 1'b0) begin // Register, Memory and Immediate−type instructions
178          if_pc_we <= 1'b0;
179          if (ex_ir[14:11] == 4'd0 && ex_ir[4] == 1'b1 && ex_ir[0] == 1'b0) // ST or SBU
180            reg_file_we <= 1'b0;
181          else if (ex_ir == 16'd0) // NOP
182            reg_file_we <= 1'b0;
183          else
184            reg_file_we <= 1'b1;
185       end
186       else begin // Branch and Jump−type instructions
187          ....
188    end
189 endmodule
190
191 module reg_file
192   (
193    input wire          clk, rst,
194    input wire [2:0]    addr1, addr2, addr3,
195    input wire [15:0]   din,
196    output logic [15:0] dout1, dout2,
197    input wire          we
198    );
199
200    reg [15:0]          register0, register1;
201    reg [15:0]          register2, register3;
202    reg [15:0]          register4, register5;
203    reg [15:0]          register6, register7;
204
205    always_comb begin
206       case (addr1)
207         3'h0: dout1 <= register0;
208         3'h1: dout1 <= register1;
209         3'h2: dout1 <= register2;
210         3'h3: dout1 <= register3;
211         3'h4: dout1 <= register4;
212         3'h5: dout1 <= register5;
```

```
213         3'h6: dout1 <= register6;
214         3'h7: dout1 <= register7;
215       endcase
216    end
217 
218    always_comb begin
219       case (addr2)
220         3'h0: dout2 <= register0;
221         3'h1: dout2 <= register1;
222         3'h2: dout2 <= register2;
223         3'h3: dout2 <= register3;
224         3'h4: dout2 <= register4;
225         3'h5: dout2 <= register5;
226         3'h6: dout2 <= register6;
227         3'h7: dout2 <= register7;
228       endcase
229    end
230 
231    always_ff @(posedge clk)
232      if (rst) begin
233         register0 <= 16'h0;
234         register1 <= 16'h0;
235         register2 <= 16'h0;
236         register3 <= 16'h0;
237         register4 <= 16'h0;
238         register5 <= 16'h0;
239         register6 <= 16'h0;
240         register7 <= 16'h0;
241      end
242      else if (we) begin
243         case (addr3)
244           3'h0: register0 <= din;
245           3'h1: register1 <= din;
246           3'h2: register2 <= din;
247           3'h3: register3 <= din;
248           3'h4: register4 <= din;
249           3'h5: register5 <= din;
250           3'h6: register6 <= din;
251           3'h7: register7 <= din;
252         endcase
253      end
254 endmodule
255 
256 ...
257 
258 `default_nettype wire
```