コンピュータアーキテクチャ（7）

坂井　修一
東京大学大学院 情報理工学系研究科 電子情報学専攻
東京大学　工学部　電子情報工学科／電気電子工学科

- はじめに
- 基本CPUの設計

はじめに

- 本講義の目的
  - コンピュータアーキテクチャの基本を学ぶ
- 時間・場所
  - 火曜日 8:40 – 10:10　工学部2号館241
- ホームページ（ダウンロード可能）
  - url:　http://www.mtl.t.u-tokyo.ac.jp/~sakai/hard/
- 教科書
  - **坂井修一『コンピュータアーキテクチャ』（コロナ社、電子情報レクチャーシリーズC-9）**
  - **坂井修一『実践　コンピュータアーキテクチャ』（コロナ社）**

  教科書通りやります
- 参考書
  - D. Patterson and J. Hennessy, Computer Organization & Design、3rd Ed.（邦訳『コンピュータの構成と設計』（第3版)上下（日系ＢＰ））
  - 馬場敬信『コンピュータアーキテクチャ』（改訂２版）、オーム社
  - 富田眞治『コンピュータアーキテクチャⅠ』、丸善
- 予備知識：　論理回路
  - 坂井修一『論理回路入門』、培風館
- 成績
  - 試験＋レポート＋出席

講義の概要と予定（１／２）

１．コンピュータアーキテクチャ入門

ディジタルな表現、負の数、実数、加算器、ＡＬＵ，フリップフロップ、レジスタ、計算のサイクル

２．　データの流れと制御の流れ

主記憶装置、メモリの構成と分類、レジスタファイル、命令、命令実行の仕組み、実行サイクル、算術論理演算命令、シーケンサ、条件分岐命令

３．命令セットアーキテクチャ

操作とオペランド、命令の表現形式、アセンブリ言語、命令セット、
算術論理演算命令、データ移動命令、分岐命令、アドレシング、
サブルーチン、RISCとCISC

４．パイプライン処理（１）

パイプラインの原理、命令パイプライン、オーバヘッド、構造ハザード、データハザード、制御ハザード

５．パイプライン処理（２）

フォワーディング、遅延分岐、分岐予測、命令スケジューリング

６．キャッシュ

記憶階層と局所性、透過性、キャッシュ、ライトスルーとライトバック、
ダイレクトマップ型、フルアソシアティブ型、セットアソシアティブ型、キャッシュミス

7. 仮想記憶

仮想記憶、ページフォールト、TLB、物理アドレスキャッシュ、仮想アドレスキャッシュ、メモリアクセス機構

講義の概要と予定（２／２）

８． 基本CPUの設計

ディジタル回路の入力、Verilog HDL、シミュレーションによる動作検証、アセンブラ、基本プロセッサの設計、基本プロセッサのシミュレーションによる検証

９．命令レベル並列処理（１）

並列処理、並列処理パイプライン、VLIW、スーパスカラ、並列処理とハザード

１０．命令レベル並列処理（２）

静的最適化、ループアンローリング、ソフトウェアパイプライニング、トレーススケジューリング

１１．アウトオブオーダ処理

インオーダーとアウトオブオーダー、フロー依存、逆依存、出力依存、
命令ウィンドウ、リザベーションステーション、レジスタリネーミング、
マッピングテーブル、リオーダバッファ、プロセッサの性能

１２．入出力と周辺装置

周辺装置、ディスプレイ、二次記憶装置、ハードウェアインタフェース、割り込みとポーリング、アービタ、DMA、例外処理

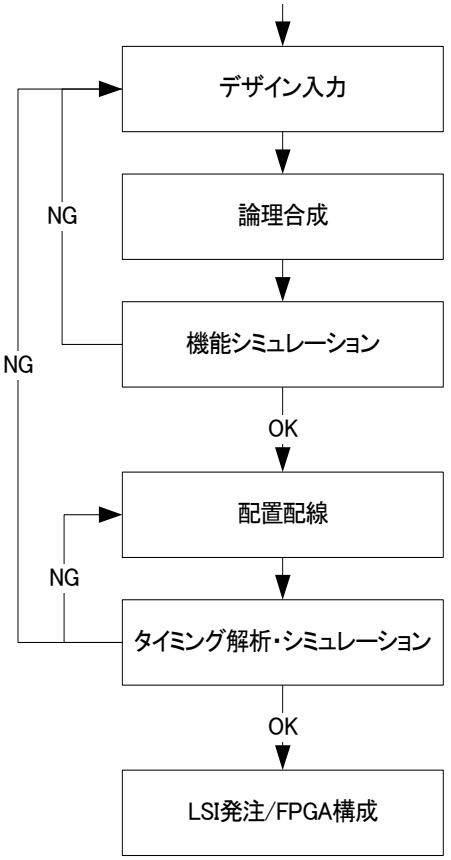
試験： ７月後半

4. 基本CPUの設計

- 内容
  - ディジタル回路の設計
  - 本講義での設計
  - 命令セット～アセンブラ
  - 基本プロセッサの設計
    - Verilog HDL
    - CPUの構成と設計の基本方針
    - トップモジュールの構成
    - 構成要素の設計
    - トップモジュールの設計
  - 基本プロセッサのシミュレーションによる検証
    - Modelsim
    - シミュレーションの方法と手順
    - 構成要素のシミュレーション
    - トップモジュールのシミュレーション

ディジタル回路の設計

- ディジタル回路の設計
  - 入力： 回路図、HDL
  - HDL = ハードウェア記述言語
    - hardware description language
- CAD
  - コンピュータ支援設計(computer aided design)
- Quartus II （Altera）
  - CADツールの一つ
  - http://www.altera.co.jp/support/software/download/altera_design/quartus_we/dnl-quartus_we.jsp
  - 「実践コンピュータアーキテクチャ」にダウンロード手順が書かれている



本講義での設計

- 基本アーキテクチャ
  - 「コンピュータアーキテクチャ」2章、3章
- アセンブラ
  - Perlで作成:「実践　コンピュータアーキテクチャ」6章
- ディジタル回路入力
  - Verilog HDL:「実践　コンピュータアーキテクチャ」7章

  ⇒　池田先生の講義「半導体デバイス工学」で学習する
- シミュレーション
  - Modelsim:「実践　コンピュータアーキテクチャ」8章

  ⇒　4年生の講義「VLSIアーキテクチャ」でも実習する
- 全体設計
  - Quartus II

命令セットとニューモニック

ニューモニック	命令形式	OP	AUX等	動作	補注
add	R	0	0	rd <- rs + rt	
addi	I	1	imm	rt <- rs + imm	
sub	R	0	2	rd <- rs - rt	
lui	I	3	imm	rt <- imm << 16	上位16ビットをセット
and	R	0	8	rd <- rs AND rt	ビットごとのAND
andi	I	4	imm	rt <- rs AND imm	即値とのAND
or	R	0	9	rd <- rs OR rt	
ori	I	5	imm	rt <- rs OR imm	
xor	R	0	10	rd <- rs XOR rt	
xori	I	6	imm	rt <- rs XOR imm	
nor	R	0	11	rd <- rs NOR rt	
sll	R	0	16	rd <- rs llshift aux[10,6]	論理左シフト
srl	R	0	17	rd <- rs lrshift aux[10,6]	論理右シフト
sra	R	0	18	rd <-rs arshift aux[10,6]	算術右シフト
lw	I	16	dpl	rt <- mem(rs+dpl)	一語読み出し
lh	I	18	dpl	rt <- mem(rs+dpl)	半語読み出し、符号拡張
lb	I	20	dpl	rt <- mem(rs+dpl)	バイト読み出し、符号拡張
sw	I	24	dpl	mem(rs+dpl) <- rt	一語書き込み
sh	I	26	dpl	mem(rs+dpl) <- rt	半語書き込み、符号拡張
sb	I	28	dpl	mem(rs+dpl) <- rt	バイト書き込み、符号拡張
beq	I	32	dpl	if rs == rt then PC = PC+dpl	等しければジャンプ
bne	I	33	dpl	if rs != rt then PC = PC+dpl	等しくなければジャンプ
blt	I	34	dpl	if rs < rt then PC = PC+dpl	未満であればジャンプ
ble	I	35	dpl	if rs <= rt then PC = PC+dpl	以下であればジャンプ
j	A	40	addr	PC <- addr	絶対番地のジャンプ
jal	A	41	addr	R31 <- PC+4; PC <- addr	ジャンプアンドリンク
jr	R	42		PC <- rs	レジスタ間接ジャンプ

アセンブラ

- アセンブラ
  - アセンブリ言語のプログラムを機械語（2進数）に変換するプログラム
- Perl言語による作成例
  - 「実践コンピュータアーキテクチャ」 図6.16

```
open(FH, "@ARGV[0]");
 $i = 0;
 # ラベル、カンマ、空白、タブの処理
 while ($line = <FH>) {
 $line =~ s/,/ /g;
 $line =~ s/¥t/ /g;
 $line =~ s/¥:/ : /g;
 $line =~ s/^ +//g;
 chomp($line);
 @instruction = split(/ +/, $line);
 if (@instruction[1] eq ":") {
 $labels{@instruction[0]} = $i;
 }
 $i++;
 }
 close(FH);

 open(FH, "@ARGV[0]");
 $i = 0;
 while ($line = <FH>) {
 # print("$i : ");
 $line =~ s/,/ /g;
 $line =~ s/¥t+/ /g;
 $line =~ s/¥:/ : /g;
 $line =~ s/^ +//g;
 # print($line);
 chomp($line);
 @instruction = split(/ +/, $line);
 if (@instruction[1] eq ":") {#    ラベルのある行をフィールドに分割する
 $op = @instruction[2];
 $f2 = @instruction[3];
 $f3 = @instruction[4];
 $f4 = @instruction[5];
 $f5 = @instruction[6];
 }
```

```
else {# ラベルのない行をフィールド分割する
 $op = @instruction[0];
 $f2 = @instruction[1];
 $f3 = @instruction[2];
 $f4 = @instruction[3];
 $f5 = @instruction[4];
 }
# 機械語の出力
    if ($op eq "add"){p_b(6,0); p_r3($f2, $f3, $f4); p_b(11,0);print("¥n");}
    elsif ($op eq "addi") {p_b(6,1); p_r2i($f2, $f3); p_b(16, $f4);print("¥n");}
    elsif ($op eq "sub"){p_b(6,0); p_r3($f2, $f3, $f4); p_b(11, 2); print("¥n");}
    elsif ($op eq "lui") {p_b(6,3); p_r2i($f2, "r0"); p_b(16, $f3);print("¥n");}
    elsif ($op eq "and"){p_b(6,0); p_r3($f2, $f3, $f4); p_b(11, 8); print("¥n");}
    elsif ($op eq "andi") {p_b(6,4); p_r2i($f2, $f3); p_b(16, $f4);print("¥n");}
    elsif ($op eq "or"){p_b(6,0); p_r3($f2, $f3, $f4); p_b(11,9);print("¥n");}
    elsif ($op eq "ori"){p_b(6,5); p_r2i($f2, $f3); p_b(16, $f4);print("¥n");}
    elsif ($op eq "xor"){p_b(6,0); p_r3($f2, $f3, $f4); p_b(11,10);print("¥n");}
    elsif ($op eq "xori"){p_b(6,6); p_r2i($f2, $f3); p_b(16, $f4);print("¥n");}
    elsif ($op eq "nor"){p_b(6,0); p_r3($f2, $f3, $f4); p_b(11,11);print("¥n");}
    elsif ($op eq "sll"){p_b(6,0); p_r3($f2, $f3, "r0"); p_b(5, $f4); p_b(6,16);print("¥n");}
    elsif ($op eq "srl"){p_b(6,0); p_r3($f2, $f3, "r0"); p_b(5, $f4); p_b(6,17);print("¥n");}
    elsif ($op eq "sra"){p_b(6,0); p_r3($f2, $f3, "r0"); p_b(5, $f4); p_b(6,18);print("¥n");}
    elsif ($op eq "lw"){p_b(6,16); p_r2i($f2, base($f3)); p_b(16, dpl($f3));print("¥n");}
    elsif ($op eq "lh"){p_b(6,18); p_r2i($f2, base($f3)); p_b(16, dpl($f3));print("¥n");}
    elsif ($op eq "lb"){p_b(6,20); p_r2i($f2, base($f3)); p_b(16, dpl($f3));print("¥n");}
    elsif ($op eq "sw"){p_b(6,24); p_r2i($f2, base($f3)); p_b(16, dpl($f3));print("¥n");}
    elsif ($op eq "sh"){p_b(6,26); p_r2i($f2, base($f3)); p_b(16, dpl($f3));print("¥n");}
    elsif ($op eq "sb"){p_b(6,28); p_r2i($f2, base($f3)); p_b(16, dpl($f3));print("¥n");}
    elsif ($op eq "beq") {p_b(6,32); p_r2b($f2, $f3); p_b(16, $labels{$f4}-$i-1);print("¥n");}
    elsif ($op eq "bne") {p_b(6,33); p_r2b($f2, $f3); p_b(16, $labels{$f4}-$i-1);print("¥n");}
    elsif ($op eq "blt") {p_b(6,34); p_r2b($f2, $f3); p_b(16, $labels{$f4}-$i-1);print("¥n");}
    elsif ($op eq "ble") {p_b(6,35); p_r2b($f2, $f3); p_b(16, $labels{$f4}-$i-1);print("¥n");}
    elsif ($op eq "j") {p_b(6,40); p_b(26, $labels{$f2});print("¥n");}
    elsif ($op eq "jal"){p_b(6, 41); p_b(26, $labels{$f2});print("¥n");}
    elsif ($op eq "jr"){p_b(6, 42); p_r3("r0", $f2, "r0"); p_b(11, 0);print("¥n");}
    else {print("ERROR: Illegal Instruction¥n");}
    $i++;
 }
 close(FH);
```

```
sub p_b{#  $numを2進数$digitsに変換して出力
 ($digits, $num) = @_;
 if ($num >= 0) {
 printf("%0".$digits."b_", $num);
 } else {
 print(substr(sprintf(“%b ”, $num), 32-$digits));                #    64ビットOSのときは、32 -> 64
 }
 }

 sub p_r3{#    R型のレジスタ番地を出力
 ($rd, $rs, $rt) = @_;
 $rs =~ s/r//; p_b(5, $rs);
 $rt =~ s/r//; p_b(5, $rt);
 $rd =~ s/r//; p_b(5, $rd);
 }

 sub p_r2i{#    I型のレジスタ番地を出力
 ($rt, $rs) = @_;
 $rs =~ s/r//; p_b(5, $rs);
 $rt =~ s/r//; p_b(5, $rt);
 }

 sub p_r2b{# 条件分岐で比較するレジスタ番地を出力
 ($rs, $rt) = @_;
 $rs =~ s/r//; p_b(5, $rs);
 $rt =~ s/r//; p_b(5, $rt);
 }

 sub base{#     ベースアドレスレジスタの番地を返す
 ($addr) = @_;
 $addr =~ s/.*¥(//;
 $addr =~ s/¥)//;
 return($addr);
 }

 sub dpl{# 変位を返す
 ($addr) = @_;
 $addr =~ s/¥(.*¥)//;
 return($addr);
 }
```

Verilog HDL

- Verilog HDL　＝　ハードウェア記述言語
  - １つないし複数のモジュールで回路を記述する
- モジュール構成

```
module　　モジュール識別子(ポートリスト);
    宣言部
    回路記述部
endmodule
```

- 宣言部

```
ポート宣言
ネット宣言
レジスタ宣言
パラメタ宣言
```

宣言

- ポート宣言

```
input   CLK, RD, DATA_IN;          // 1ビット入力
input  ［31:0］DBUS_IN;            // 32 ビット入力バス
output DATA_OUT;                   // 1 ビット出力
output［31:0］DBUS_OUT;            // 32 ビット出力バス
inout   CTL;                       // 1 ビット入出力
inout  ［31:0］DBUS_INOUT;         // 32 ビット入出力バス
```

- ネット宣言

```
wire   SIGNAL1;                    // 1 ビット内部信号
wire  ［31:0］DBUS;                // 32 ビットバス
```

- レジスタ宣言

```
reg   Q;                           // 1 ビットフリップフロップの出力
reg  ［31:0］R;                    // 32 ビットレジスタの出力
```

- パラメタ宣言

```
parameter   ONE = 8'b00000001;     // 8 ビットの定数
parameter   TEN = 10;              //10 進数の定数
```

値と型

値

論理値	電圧
0	低電圧　（0V など）
1	高電圧（1.1V など）
x	不定値
z	ハイ・インピーダンス

型

型分類	型（予約語）	意味	代入可能な場所
ネット型	wire	0, 1 を出力できる信号で値を保持しないもの	assign 文の中
	tri	0, 1, Z を出力できる信号で値を保持しないもの	
	wor	ワイヤード OR（値を保持しない）	
	wand	ワイヤード AND（値を保持しない）	
レジスタ型	reg	値を保持する信号	always 文の中 initial 文の中 function の中 task の中

定数

定数の表記

ビット幅 ' 基数　値

項目	表記 : 意味	省略時
ビット幅	10 進数 : 2 進数で表したときの桁数	32 ビット
' 基数	'b, 'B: 2 進数 'o, 'O: 8 進数 'd, 'D: 10 進数 'h, 'H: 16 進数	10 進数
値	基数によって定められた表記 :　値	

定数の例

2 進数	1'b0	1'b1	1'bx	1'bz	4'b1010	8'bxxx1_1111
8 進数	3'o5	5'o37	7'ozz3	'o35_165_072_424		
10 進数	0	4'd12	2047	32'd123456789		
16 進数	3'h5	7'h2e	12'hzzz	'hffff	32'h00_00_00_xx_00	

素子とライブラリ

- 基本素子

  and　or　not　nand　nor　xor　xnor　buf

- 3状態素子

  bufif0　bufif1　notif0　notif1

- 回路ライブラリ

  フリップフロップ、マルチプレクサ、デコーダなど

演算

演算子

算術演算子	+	加算	等号演算子	==	等しい
	-	減算		!=	等しくない
	*	乗算	関係演算子	>	大なり
	/	除算		>=	以上
	%	剰余		<	小なり
ビット演算子	~	NOT		<=	以下
（ビット単位）	&	AND	リダクション	&	リダクション AND
	|	OR	演算子	|	リダクション OR
	^	XOR		~&	リダクション NAND
	~^	XNOR		~|	リダクション NOR
論理演算子	!	論理否定		^	リダクション XOR
	&&	論理積		~^	リダクション XNOR
	||	論理和	条件演算子	? :	条件付き実行
シフト演算子	<<	左シフト	連接演算子	{ , }	連接
	>>	右シフト			

リダクション演算、条件演算、連接演算
演算の優先順位　⇒　教科書を参照せよ

回路記述部

- assign文： 単純な組合せ回路

  assign　ネット型信号名　＝　論理式

- function： まとまった論理関数

  function　ビット幅　ファンクション名；

  　　宣言部

  　　文

  endfunction

- if文： 条件分岐

  if（式） 文1　else 文2

- case文：3方向以上の分岐

  case（式）

  　式、式…、式：文

  　式、式…、式：文

  　default: 文

  endcase

- always文： 主に順序回路の記述

  always @(イベント式） 文

- モジュール呼び出し

  モジュール名　インスタンス名（ポートリスト）

記述例

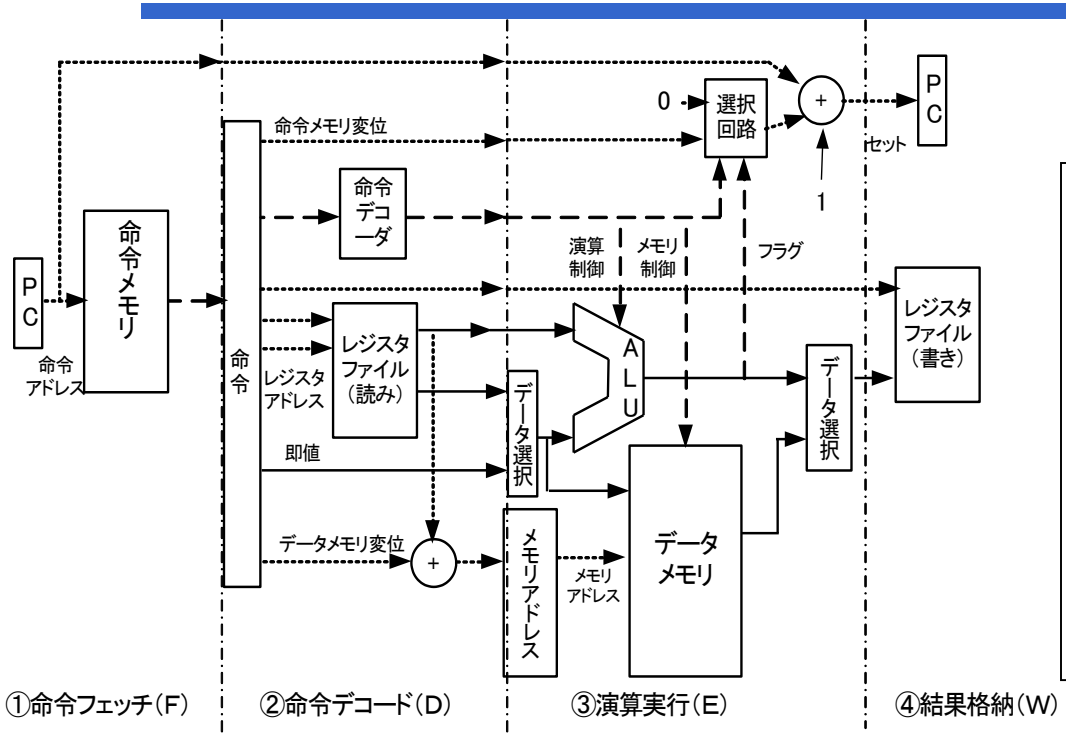
- ３２ビットレジスタ

```
module register32（CLK, RD, DI, DO）;
  input CLK, RD;
  input［31:0］ DI;
  output［31:0］ DO;
  reg［31:0］ DO;

  always @（negedge RD or posedge CLK）
     begin
          if（RD == 1'b0） DO <= 32'h00000000;
          else DO <= DI;
  end

endmodule
```

CPUの構成と設計の基本方針



アドレスの流れ　制御の流れ　データの流れ

動作に着目したプロセッサの内部構成

設計方針

(1) 上位モジュールは「動作」を基本とし、①命令フェッチ、②命令デコード、③実行、④結果の格納をそれぞれモジュールとして設計する。

(2) 下位モジュールは「ハードウェアの実体」に近いものとする。

トップモジュールの構成

```
module   computer();       //  コンピュータ全体
endmodule

module fetch();             // 命令フェッチ
endmodule

module decode();            // 命令デコード
endmodule;

module execute();           //  実行
endmodule;

module writeback();         //  結果格納
endmodule;


module register_file();     // レジスタファイル
endmodule;
```

- モジュール分割
  - 「動作」の単位
    - fetch
    - decode
    - execute
    - writeback
  - 複数のものから参照
    - register_file
- 入出力信号の設定
  - 教科書図7．5

構成要素の設計

- 命令フェッチ部
  - PCで指定される命令メモリの番地から命令を取り出し、デコード部に送る
- デコード部　→　消滅
- 実行部：　さらなる階層構造
  - opr_gen：　オペレーションの生成
  - alu：　ALU
  - wreg：書き込みレジスタ番地の計算
  - wrengen：　メモリアクセスのバイト指定
  - calc：　計算実行
  - npc：　次のPCの値の計算
- 書き戻し部
  - 結果データをレジスタに書き込み、PCを更新する

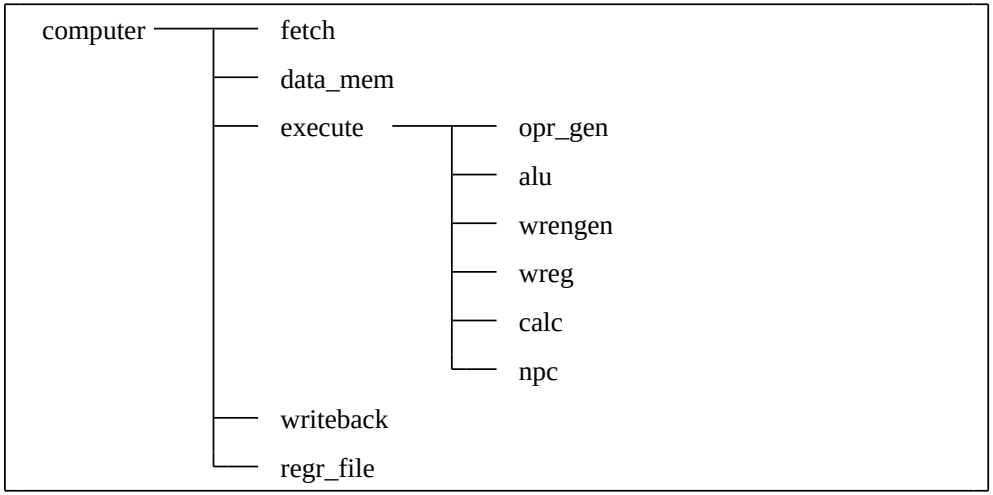


図 8.1　基本プロセッサの階層構造

命令フェッチ部

```
module fetch（pc, ins）;
  input［31:0］ pc;
  output［31:0］ ins;
  reg［31:0］ ins_mem［0:255］;

    assign ins = ins_mem［pc］ ;

endmodule
```

図 7.6 命令フェッチ部

実行部

```
module execute(clk, ins, pc, reg1, reg2, wra, result, nextpc);
    input clk; input [31:0] ins, pc, reg1, reg2;
    output[4:0] wra; output[31:0] result, nextpc;
    wire [5:0] op;   wire [4:0] shift, operation;   wire [25:0] addr;
    wire [31:0] dpl_imm, operand2, alu_result, nonbranch, branch, mem_address, dm_r_data;    wire [3:0] wren;

function [4:0] opr_gen;
function [31:0] alu;
function [31:0] calc;
function [31:0] npc;
function [4:0] wreg;
function [3:0] wrengen;

        assign op = ins[31:26]; assign shift = ins[10:6]; assign operation = ins[4:0]; assign dpl_imm = {{16{ins[15]}}, ins[15:0]};
        assign operand2 = (op == 6'd0) ? reg2: dpl_imm; assign alu_result = alu(opr_gen(op, operation), shift, reg1, operand2);
        assign mem_address = (reg1 + dpl_imm) >>> 2; assign wren = wrengen(op);
        data_mem data_mem_body0 (mem_address[7:0], clk, reg2[7:0], wren[0], dm_r_data[7:0]);
        data_mem data_mem_body1 (mem_address[7:0], clk, reg2[15:8], wren[1], dm_r_data[15:8]);
        data_mem data_mem_body2 (mem_address[7:0], clk, reg2[23:16], wren[2], dm_r_data[23:16]);
        data_mem data_mem_body3 (mem_address[7:0], clk, reg2[31:24], wren[3], dm_r_data[31:24]);
        assign wra = wreg(op, ins[20:16], ins[15:11]); assign result = calc(op, alu_result, dpl_imm, dm_r_data, pc);
        assign addr = ins[25:0]; assign nonbranch = pc+32'd1; assign branch = nonbranch + dpl_imm;
        assign nextpc = npc(op, reg1, reg2, branch, nonbranch, addr);
  endmodule    // end of execute
```


図 7.8　実行部

オペレーションの生成

```
function [4:0] opr_gen;
    input [5:0] op;
    input [4:0] operation;
     case (op)
          6'd0: opr_gen = operation;
          6'd1: opr_gen = 5'd0;
          6'd4: opr_gen = 5'd8;
          6'd5: opr_gen = 5'd9;
          6'd6: opr_gen = 5'd10;
          default: opr_gen = 5'h1f;
      endcase
     endfunction
```

alu

```
function [31:0] alu;
    input [4:0] opr, shift;
    input [31:0] operand1, operand2;
     case (opr)
         5'd0: alu = operand1 + operand2;
         5'd1: alu = operand1 - operand2;
         5'd8: alu = operand1 & operand2;
         5'd9: alu = operand1 | operand2;
         5'd10: alu = operand1 ^ operand2;
         5'd11: alu = ~ (operand1 & operand2);
         5'd16: alu = operand1 << shift;
         5'd17: alu = operand1 >> shift;
         5'd18: alu = operand1 >>> shift;
         default: alu = 32'hffffffff;
     endcase
   endfunction
```

calc

```
function [31:0] calc;
    input [5:0] op;
    input [31:0] alu_result, dpl_imm, dm_r_data, pc;
      case (op)
        6'd0, 6'd1, 6'd4, 6'd5, 6'd6: calc = alu_result;
        6'd3: calc = dpl_imm << 16;
        6'd16: calc = dm_r_data;
        6'd18: calc = {{16{dm_r_data[15]}}, dm_r_data[15:0]};
        6'd20: calc = {{24{dm_r_data[7]}}, dm_r_data[7:0]};
        6'd41: calc = pc+32'd1;
        default: calc = 32'hffffffff;
      endcase
    endfunction
```

npc

```
function [31:0] npc;
    input [5:0] op;
    input [31:0] reg1, reg2, branch, nonbranch, addr;
  case (op)
     6'd32:  npc = (reg1 == reg2)? branch : nonbranch;
     6'd33:  npc = (reg1 != reg2)? branch : nonbranch;
     6'd34:  npc = (reg1 < reg2)? branch : nonbranch;
     6'd35:  npc = (reg1 <= reg2)? branch : nonbranch;
     6'd40, 6'd41:  npc = addr;
     6'd42: npc = reg1;
     default: npc = nonbranch;
  endcase
endfunction
```

wreg

```
function [4:0] wreg;
    input [5:0] op;
    input [4:0] rt, rd;
      case (op)
            6'd0: wreg = rd;
            6‘d1, 6’d3, 6‘d4, 6’d5, 6‘d6, 6'd16, 6'd18, 6'd20: wreg = rt;
            6'd41: wreg = 5'd31;
            default: wreg = 5'd0;
      endcase
    endfunction
```

wrengen

```
function [3:0] wrengen;
    input  [5:0] op;
     case (op)
      6'd24: wrengen = 4'b0000;
      6'd26: wrengen = 4'b1100;
      6'd28: wrengen = 4'b1110;
      default: wrengen = 4'b1111;
     endcase
    endfunction
```

データメモリ

```
module data_mem（address, clk, write_data, wren, read_data）;
    input［7:0］ address;
    input clk, wren;
    input［7:0］ write_data;
    output［7:0］ read_data;
    reg［7:0］ d_mem［0:255］;

    always @（posedge clk）
        if（wren == 0） d_mem［address］ <= write_data;
    assign read_data = d_mem［address］;
endmodule
```

図 7.9　データメモリ

書き戻し部

```
module writeback（clk, rstd, nextpc, pc）;
    input clk, rstd;
    input［31:0］ nextpc;
    output［31:0］ pc;
    reg［31:0］ pc;
    always @（negedge rstd　or posedge clk）
        begin
            if（rstd == 0） pc <= 32'h00000000;
            else if（clk == 1） pc <= nextpc;
        end
endmodule
```

図 7.10　書き戻し部

レジスタファイル

```
module reg_file（clk, rstd, wr, ra1, ra2, wa, wren, rr1, rr2）;
    input clk, rstd, wren;
    input ［31:0］ wr;
    input［4:0］ ra1, ra2, wa;
    output［31:0］ rr1, rr2;
    reg［31:0］ rf［0:31］;

    assign rr1 = rf［ra1］;
    assign rr2 = rf［ra2］;
    always @（negedge rstd or posedge clk）
        if（rstd == 0） rf［0］ <= 32'h00000000;
        else if（wren == 0） rf［wa］ <= wr;
endmodule
```

トップモジュールの設計

```
module computer（clk, rstd）;
  input clk, rstd;
  wire［31:0］ pc, ins, reg1,reg2, result, nextpc;
  wire［4:0］ wra;
  wire［3:0］ wren;
    fetch fetch_body（pc［7:0］, ins）;
    execute execute_body（clk, ins, pc, reg1, reg2, wra, result, nextpc）;
    writeback writeback_body（clk, rstd, nextpc, pc）;
    reg_file rf_body（clk, rstd, result, ins［25:21］, ins［20:16］, wra,（~| wra）, reg1, reg2）;
endmodule
```

シミュレーション（オプション）

- Intel/Altera Quartus Prime（CADシステム）
  - ダウンロードサイト
    - https://www.altera.co.jp/downloads/download-center.html
    - Quartus Prime Lite (License Free)
      - Quartus本体
      - Modelsim： シミュレータ
  - Tutorialをやってみる
- 本番： 『実践 コンピュータアーキテクチャ』参照
  - Verilog HDLで記述
  - パーツの検証： シミュレータはModelsimで
  - 全体の検証
  - 改良、 評価