1　作成したプロセッサ

1.1　できたこと

シングルサイクルプロセッサを作成。Uart は成功したが CoreMark でなにも出力されず性能評価まで至らず。

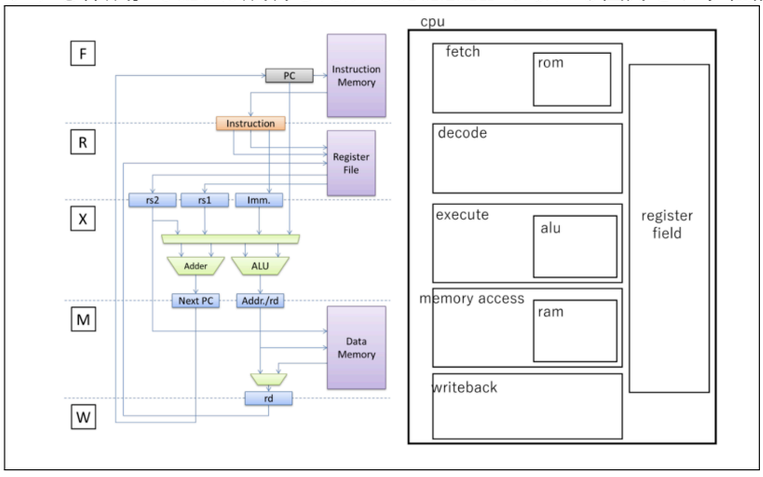

図 1　参考にした図

1.2　作成したモジュール

以下のように wire を作成しそれぞれのモジュールへと繋げた。

- fetch (rom は内蔵)
- decoder
- execute
  - alu
- data_mem
- reg_file
- write_back
- uart
- hardware_counter

1.3　cpu モジュールの実装

```
module cpu(
    input wire sysclk,
    input wire cpu_resetn,
    output wire uart_tx
    );
    wire rst;
    assign rst = cpu_resetn;

    wire [31:0] pc; //プログラムカウンタ
    wire [31:0] ir; //

    wire [4:0]  srcreg1_num;
    wire [4:0]  srcreg2_num;
    wire [4:0]  dstreg_num;
    wire [31:0] imm;
    wire [5:0]  alucode;
    wire [1:0]  aluop1_type;
    wire [1:0]  aluop2_type;
    wire            reg_we;
    wire            is_load;
    wire                is_store;
    wire        is_halt;

    wire [31:0] srcreg1_data;
    wire [31:0] srcreg2_data;
    wire [31:0] nextpc;
    wire [31:0] alu_result;

    wire [31:0] dstreg_data;

    wire [31:0] ram_addr;

    //    wire [31:0] r_addr;
    wire [31:0] w_data;
    wire [31:0] r_data;
```

```
    wire [31:0] hc_OUT_data;

    // uart
    wire [7:0] uart_IN_data;
    wire uart_we;
    wire uart_OUT_data;
```

1.3.1 モジュールの繋げ方

基本的に各モジュールを wire で繋いだだけ。load 命令と計算結果のどちらかを使用するかだけは cpu モジュール内で決めた。このようにした理由は is_load によってメモリ制御のモジュールの動作を変更するようにするより cpu モジュール内で is_load によって繋げる wire を変更するように実装する方が楽だったからです。

```
assign dstreg_data =  is_load   ?  r_data       : alu_result;
```

2 苦労した点

データのもろもろを理解できるのに二日間かかった。fetch, reg_file, data_file をどのようにして作るか考えるのに時間がかかった。

2.1 decoder 作成 (verilog 文法の理解)

2.1.1 wire の値の変更

srcreg1_num, srcreg2_num, dstreg_num, imm の三つは wire のため

```
  always @(ir) begin

  srcreg1_num <= ir[19:15];
```

のように値を変更するのは文法エラー always の外側の位置で assign 文で変更する必要がある。

そこで op_type というレジスタを用意してその値を always 文ないで変更し、レジスタの値によってそれぞれの wire が接続する値を変更することにした。

```
  reg [2:0] op_type;

  assign  srcreg1_num  = (op_type == `TYPE_U) ? 5'b0:
                         (op_type == `TYPE_J) ? 5'b0:
                         (op_type == `TYPE_I) ? ir[19:15]:
                         (op_type == `TYPE_B) ? ir[19:15]:
                         (op_type == `TYPE_S) ? ir[19:15]:
                         (op_type == `TYPE_R) ? ir[19:15]:
                         5'b0;

  always @(ir) begin
      // 略
      op_type <= 3'd3;
```

2.1.2 wire の変更の時差

以下コードないにうまく動くコードと動かないコードを入れる。

```
  assign dstreg_num =   (op_type == `TYPE_U) ? ir[11:7]:
                        (op_type == `TYPE_J) ? ir[11:7]:
                        (op_type == `TYPE_I) ? ir[11:7]:
                        (op_type == `TYPE_B) ? 0:
                        (op_type == `TYPE_S) ? 0:
                        (op_type == `TYPE_R) ? ir[11:7]:
                        5'b0;
always @(ir) begin
// 略
`JAL    : begin
op_type <= `TYPE_J;

// うまくいかない
if (dstreg_num == 0) reg_we <= `REG_NONE;
else reg_we <= `REG_RD;

// うまくいく
if (ir[11:7] == 0) reg_we <= `DISABLE;
else reg_we <= `REG_RD;
```

op_type ¡= `TYPE_J; とした直後に dstreg_num=ir[11:7] となるわけではなく、時差があるため、レジスタではなく直接 ir[11:7] を呼び出すことが必要。

2.2 fetch の理解

decoder と alu、execute を書いた後一番困ったのが fetch の理解でした。与えられた benchmarks/tests/ControlTransfer/code.hex からどのように命令を読み取るのか、一番最初の初期値としてプログラムカウンタの値をどのように設定すればいいのか分かりませんでした。

2.2.1 fetch_tb.v の作成

fetch モジュールが機能しているのか確かめたかったため自前で fetch_tb.v を作成しました。これによりデータの取り出しが合っていることを確かめることができました。個人的に alu,docoder のテストベンチの他に fetch のテストベンチも用意してくだされば fetch の仕方をもう少し早く理解できたのではないかと思います。

```
`include "fetch.v"

`define assertf(name, signal, value) \
        if (signal !== value) begin \
            $display("ASSERTION FAILED in %s : signal is 'Ox%x' but expected 'Ox%x'!", name, signal, value); \
            $finish; \
        end else begin \
            $display("    signal == value"); \
        end

`define test_fetch(name, ex_ir) \
        $display("%s:", name); \
        `assertf("result", ir, ex_ir) \
        $display("%s test passed\n", name); \


module fetch_tb;
    reg [31:0] pc;
    wire [31:0] ir;


    fetch fetch(
        .pc(pc)
        ,.ir(ir));

    initial begin
      pc = 32'h8000;
      $display("%d",pc);
      $display("%d",pc[31:2]);
      #10

        `test_fetch("fetch", 32'h0080006f)
        $display("all tests passed!");
    end

endmodule
```

出力

```
[Running] fetch_tb.v
WARNING: ./fetch.v:6: $readmemh(/Users/momoka/git/microprocessor/benchmarks/tests/ControlTransfer/
    code.hex): Not enough words in the file for the requested range [0:32767].
ERROR: ./fetch.v:8: $readmemh: Unable to open /home/denjo/microprocessor/benchmarks/tests/Uart/code.
    hex for reading.
      32768
      8192
fetch:
    ir == 32'h0080006f
fetch test passed

all tests passed!
[Done] exit with code=0 in 0.192 seconds
```

ここではファイル benchmarks/tests/Uart/code.hex の 8193 行目の内容が 0080006f となっているかを確かめるテストケースである。結局 8193 行目の内容を得るためには pc=32' h8000 とすることが正解でした。

$pc = 32'\ h8000 = 2^{15} = 32768 = 4 \times 8192$

最初の命令は 8193 行目に出てくる。一命令 32 ビットだがメモリ空間はワードアドレッシングのため、$32 \div 8 = 4$ 倍した値を pc にする必要がある。よって pc の値は 4×8192 とするべき。

2.3 data_mem モジュールの作成

load,store の知識があやふやだったためとても難したかった。

2.3.1 load の作成

最初は store と同じようにクロック同期にしていたが、それだと欲しい値がえられるのが 1 クロック後からになってしまうため assign 文で書くことにした。シングルサイクルプロセッサなのでクロック同期は write_back モジュールの一箇所のみにした。

```
assign r_data = (alucode == `ALU_LB  &&  addr[1:0] == 2'd0) ? {{24{ram[addr[16:2]][7]}}, ram[addr[16:2]][7:0]}:
                (alucode == `ALU_LB  &&  addr[1:0] == 2'd1) ? {{24{ram[addr[16:2]][15]}}, ram[addr[16:2]][15:8]}:
                (alucode == `ALU_LB  &&  addr[1:0] == 2'd2) ? {{24{ram[addr[16:2]][23]}}, ram[addr[16:2]][23:16]}:
                (alucode == `ALU_LB  &&  addr[1:0] == 2'd3) ? {{24{ram[addr[16:2]][31]}}, ram[addr[16:2]][31:24]}:

                (alucode == `ALU_LH  &&  addr[1:0] == 2'd0) ? {{16{ram[addr[16:2]][15]}}, ram[addr[16:2]][15:0]}:
                (alucode == `ALU_LH  &&  addr[1:0] == 2'd1) ? {{16{ram[addr[16:2]][23]}}, ram[addr[16:2]][23:8]}:
                (alucode == `ALU_LH  &&  addr[1:0] == 2'd2) ? {{16{ram[addr[16:2]][31]}}, ram[addr[16:2]][31:16]}:

```

```
                (alucode == `ALU_LW  &&  addr[1:0] == 2'd3) ? ram[addr[16:2]][31:0]:

                (alucode == `ALU_LBU  &&  addr[1:0] == 2'd0) ? {{24{1'b0}}, ram[addr[16:2]][7:0]}:
                (alucode == `ALU_LBU  &&  addr[1:0] == 2'd1) ? {{24{1'b0}}, ram[addr[16:2]][15:8]}:
                (alucode == `ALU_LBU  &&  addr[1:0] == 2'd2) ? {{24{1'b0}}, ram[addr[16:2]][23:16]}:
                (alucode == `ALU_LBU  &&  addr[1:0] == 2'd3) ? {{24{1'b0}}, ram[addr[16:2]][31:24]}:

                (alucode == `ALU_LHU  &&  addr[1:0] == 2'd0) ? {{16{1'b0}}, ram[addr[16:2]][15:0]}:
                (alucode == `ALU_LHU  &&  addr[1:0] == 2'd1) ? {{16{1'b0}}, ram[addr[16:2]][23:8]}:
                (alucode == `ALU_LHU  &&  addr[1:0] == 2'd2) ? {{16{1'b0}}, ram[addr[16:2]][31:16]}:

                ram[addr[16:2]][31:0]; //どれにも当てはまらなかったら困る。とりあえずと同じにする。`ALU_LW
```

3　困ったこと

3.1　verilator の使用

使おうとしたがエラーによりコンパイルしてくれなかったため使えなかった。たくさんの warning で勉強にはなった。
よくわからなかった warning 達
＜＝ はタイミングずれるから＝にしろよ warning。そうなのか疑問に思いながら＜＝を＝に変更したが動かず。
メモリの大きさが違うと言われた。直した。メモリの大きさは間違ってても動いてしまうことがあることがわかった。ここら辺を直しても case 文の case 足りない warning が残りそれによりコンパイルしてくれなかった。(case 文 warning 残してもコンパイルしてくれるやろって思って別のところに原因があると思い諦めた。

4　工夫したこと

4.1　git の使用

学科 pc デコードを書くことになれていないため mac の VScode でコードを書いて学科 pc に git で送って Ubuntu で Vivado を使うという方法をとった。他の人と違い verilog でシミュレーションをしたいときは毎回 mac から学科 pc へ git を用いてコードを送ったため commit の回数は多くなった。git を多用したため uart が昨日まで動いていたのに今日なぜか動かない。みたいな時にコードを見返すことができた

4.1.1　iverilog の使用

微妙な差だが、個人的に vivado より iverilog の方が文法エラーがわかりやすかった。おもに iverilog で些細な文法ミスなどを発見してから git を使って学科 pc に送り vivado で全体のシミュレーションをして構造的なミスを見つける。といった使い分けをしていた。

4.1.2　iverilog の環境構築

```
brew install icarus-verilog
```

以下のコマンドでコンパイルできるらしい

```
iverilog -o 出力ファイル名[] -s トップモジュール名[] [.ファイルの羅列v]
```

VSCode 上では専用のエクステンションを使用することで実行した。https://github.com/leafvmaple/vscode-verilog
特に decoder_tb.v,alu_tb.v を実行するときは VSCode 上でテストした方が分かりやすかった。

5　やりたかったこと

5.1　完成

uart までしか完成させることができず、性能評価は周波数を下げても何も出力されませんでした。最後まで完成させたかったです。またパイプライン化やりたかった。

5.2　alu の変更

今回 alu は alu のテストベンチに剃って作成したが、個人的にテストベンチにそって作成した結果の alu モジュールと execute モジュールの仕事の分担に疑問があった。さらなる変更として現在の alu の仕事を execute に移動させて alu は計算、シフトのみを行う純粋な alu としてのモジュールに変えたいと思った。具体的には以下のように 5 ビットの制御信号を execute から受け取りそれにより決められた動作をするような alu を作成したいと思った。

制御信号					動作	説明
F4	F3	F2	F1	F0		
0	0	0	1	0	Y <= A + B	加算
0	0	0	1	1	Y <= A − B	減算、B の2の補数と A の加算で実現する
0	1	0	0	0	Y <= A and B	各桁ごとに and 演算を取る
0	1	1	0	0	Y <= A or B	各桁ごとに or 演算を取る
0	0	0	0	0	Y <= shl A	A を左へ（上位へ）1ビットシフト
1	0	0	0	0	Y <= shr A	A を右へ（下位へ）1ビットシフト

図 2　alu の制御信号と機能

参考文献

[1] http://exp.mtl.t.u-tokyo.ac.jp/2020/b3exp/wikis/home
[2] 実践 コンピュータアーキテクチャ (改訂版)