# 情報工学実験III
# MINI-MIPS 設計　最終レポート

035743A：比嘉雅樹

提出日：2005/07/21

# 1　４つの命令の実装

既存の Mini-MIPS に対し、andi、sll、srl、j の４つの命令を実装させた。図１に、命令実装後のブロック図を示す。

図 1: ４つの命令実装後のブロック図

## 1.1　VHDL 記述の変更点

- andi
  andi は、ALU での動作を変更する以外に記述は ori との違いは無い為記述は省く。

- sll、srl
  sll 命令と srl 命令の場合、ALU 内での記述を以下の様にする事で実装した。

vhdl 記述

```
      when "100"  =>
        ALU_ODATA <= SHL(ALU_IDATA2,INST( 10 downto 6 ));
      when "101"  =>
        ALU_ODATA <= SHR(ALU_IDATA2,INST( 10 downto 6 ));
```

  SHL( std_logic_vector, std_logic_vector ); は、第一引数で渡した値を第二引数で指定したビット分左にシフトを行う。SHR も同様の様にして右シフトを行う。

- j
  j 命令は、以下の様に記述する事で実装した。

vhdl 記述

```
signal JUMP  : std_logic;
signal J_ADDR  : std_logic_vector( 31 downto 0 );
signal J_SHIFTER_IN  : std_logic_vector( 25 downto 0 );
signal J_SHIFTER_OUT  : std_logic_vector( 27 downto 0 );


-- 2-Bits Left Shifter (jump)
  J_SHIFTER_IN <= INST( 25 downto 0 );
  J_SHIFTER_OUT <= J_SHIFTER_IN & "00";

-- Jump Address
  J_ADDR <= INC_ADDR(31 downto 28 ) & J_SHIFTER_OUT;
```

## 1.2 動作確認

４つの命令が動くかを確認するため、以下のようなプログラムを実行させた。そのシミュレーション結果を図 2 に示す。

アセンブラプログラム

```
andi R01, R01, 16
 ori R02, R01, 32
 sll R02, R02, 2
 srl R02, R02, 3
   j 1
```

VirSim – Waveform – SIM – AutoGroup0

le Edit Zoom Display View Window Help

EGroup0 SIM

ur 2,000 (1 ns)
1 121 (1 ns)
lta -1,879 (1 ns)

0 C1 100 200 300

| Signal | Value | Waveform |
|---|---|---|
| SCLK | 0 | |
| SRESET | 0 | |
| INST(31 DOWN | X"08000001" | 30210010 *020 *080 *0c2 *001 *020 *080 *0c2 *001 *020 *080 *0c2 *001 *020 *0 |
| REG_FILE(1) | 0 | 0 0 0 |
| REG_FILE(2) | 16 | 0 0 32 128 16 16 32 128 16 16 32 128 16 16 3 |

/TB_MM_SC/SRESET 1

# 2　パイプライン化

Mini-MIPS をパイプライン化した Mini-MIPS のブロック図を図 3 に示す。

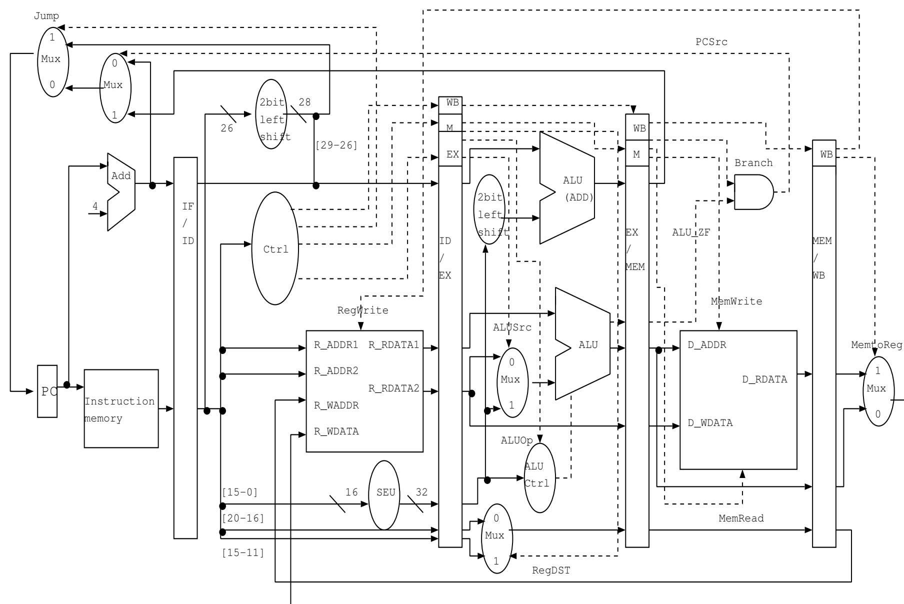



図 3: パイプライン化後のブロック図

パイプライン化前と比べ、パイプライン化後のブロック図には各ステージ間にパイプライン・レジスタが設けられ、次のステージに送るデータを経由させている。それにより各ステージで独立した処理をさせパイプライン化が可能になる。

## 2.1　動作確認

パイプライン化の動作を確認する為、1 から 10 の総和を求める下記のプログラムを用いてシミュレーションした。図 4 に、その実行結果を示す。ここでは、ハザード回避の為、各命令間に NOP 命令を挟んである。

アセンブラプログラム

```
andi R00, R00, 0
 ori R01, R01, 1
 ori R02, R02, 10
 slt R03, R00, R02
 beq R03, R00, 3
 add R04, R04, R02
 sub R02, R02, R01
   j 3
  sw R04, 0(R00)
```

X VirSim – Waveform – SIM – AutoGroup0

it Zoom Display View Window Help

EGroup0 SIM

20,000 (1 ns)
4,992 (1 ns)
-15,008 (1 ns)

0 1,500 3,000 4,500 C1

| Signal | Value | Waveform |
| --- | --- | --- |
| | 1 | |
| ET | 0 | |
| (31 DOWN | X"fc000000" | fc000000 |
| _1(31 DC | X"fc000000" | fc000000 |
| _2(31 DC | X"fc000000" | fc00000 |
| FILE(0) | 0 | 0 0 0 |
| FILE(1) | 1 | 0 1 1 1 |
| FILE(2) | 0 | 0 10 9 9 8 8 7 7 6 6 5 5 4 4 3 3 2 2 1 1 0 |
| FILE(3) | 0 | 0 0 1 1 1 0 |
| FILE(4) | 55 | 0 0 10 19 27 34 40 45 49 52 54 55 |
| M(0) | 55 | 0 0 0 55 |

## 2.2 VHDL 記述

パイプライン化は、図 3 のブロック図のように各ステージ間にパイプラインレジスタを設けることで実現した以下は、そのパイプラインレジスタの記述の一部である。

ID/EX ステージ間のパイプラインレジスタ

```
-- Pipeline2
process ( CLK, RESET )
begin
  if ( RESET = '1' ) then
    JUMP_2  <= '0';
J_ADDR_2  <= ( others => '0' );
ALU_OP_2  <= "00";
REG_DST_2  <= '0';
REG_RW_2  <= '0';
ALU_SRC_2  <= "00";
DMEM_RE_2  <= '0';
DMEM_WE_2  <= '0';
BRANCH_2  <= '0';
MEM_TO_REG_2  <= '0';
INST_2  <= (others => '0' );
INC_ADDR_2 <= ( others => '0' );
REG_RDATA1_2 <= ( others => '0' );
REG_RDATA2_2 <= ( others => '0' );
SEU_OUT_2 <= ( others => '0' );
  elsif ( CLK'event and CLK = '1'  ) then
    JUMP_2  <= JUMP;
J_ADDR_2  <= J_ADDR;
ALU_OP_2  <= ALU_OP;
REG_DST_2  <= REG_DST;
REG_RW_2  <= REG_RW;
ALU_SRC_2  <= ALU_SRC;
DMEM_RE_2  <= DMEM_RE;
DMEM_WE_2  <= DMEM_WE;
BRANCH_2  <= BRANCH;
MEM_TO_REG_2  <= MEM_TO_REG;
INST_2  <= INST_1;
INC_ADDR_2 <= INC_ADDR_1;
REG_RDATA1_2 <= NEW_DATA_1;
REG_RDATA2_2 <= NEW_DATA_2;
SEU_OUT_2 <= SEU_OUT;
  end if;
end process;
```

# 3 パイプライン・ハザードの対策

## 3.1 Mini-MIPS で発生しうるハザード

パイプライン・ハザードには大別してデータハザード、コントロールハザード、構造ハザードの３つがあるが、Mini-MIPS で発生しうるハザードはデータハザードの RAW ハザードとコントロールハザードの２つである。今回は、データハザードを回避する為に、データフォワーディングという手法をとる。これにより、レジスタに値を書き込む前にそのレジスタから古い値を読み込んで生じる矛盾をなくすことができる。例えば以下の様な命令があるとすると、２つ目の命令で求めたい R04 の値は R02+R03+R02=11 である。しかし２つ目の命令が読み込まれた時、まだ１つ目の命令で求めた結果はまだレジスタに書き込まれておらず、古い値を読み込んでしまう。そのため以下の命令の場合、最終的に R04 に書き込まれる値は R02+R03+R02=11 ではなく、R01+R02=3 となってしまう。

例

```
--R01=0,  R02=3, R03=5 とする。
add R01, R02, R03
add R04, R01, R02
```

データフォワーディングは既に演算結果が求められているステージからデータパスを繋ぐ事により、このハザードを解決する手法である。図５はデータフォワーディングを行っている様子である。

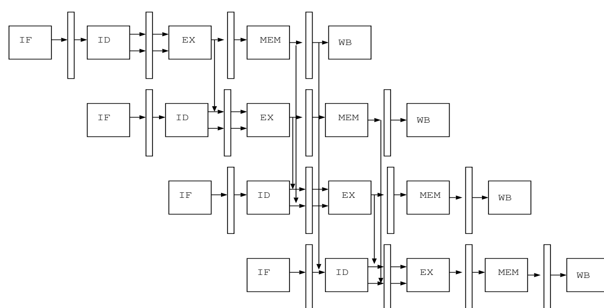



図 5: データフォワーディング

## 3.2　ブロック図

図 6 にデータハザードの対策を行った Mini-MIPS のブロック図を示す。太い線が新たに追加した部分である。

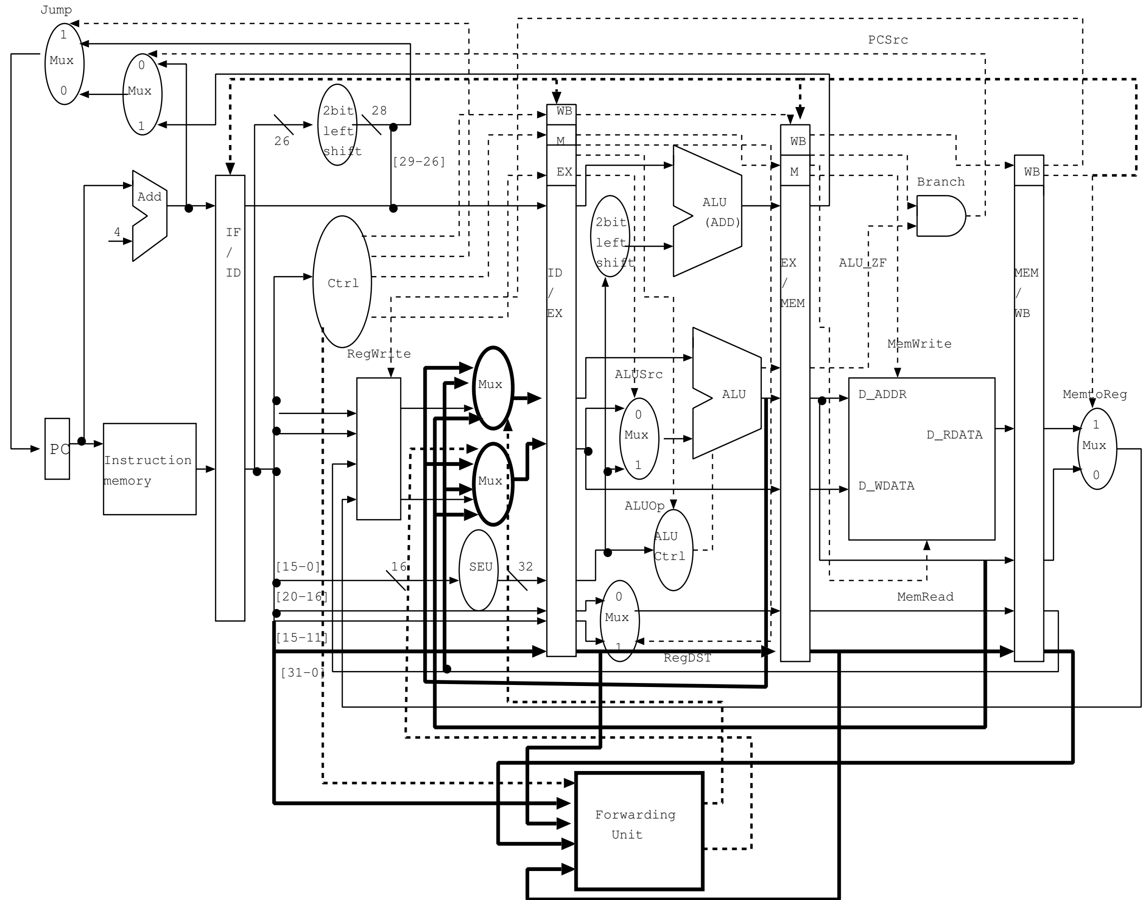



図 6: ハザード対策後のブロック図

ハザード対策前の回路に対し、フォワーディングが発生するか否かを判定する「Forwarding Unit」を新たに設けた。また、lw 命令については値を読み出すステージが他の命令と異なるため、lw 命令の場合は NOP を挿入している。（実際には挿入してないが、そう見せかけている）この際 lw 命令を判断するために制御線 `MEM_TO_REG` を利用した。これは今回の miniMIPS に含まれている命令の中では lw 命令しか `MEM_TO_REG=1` になる事が無いからである。

### 3.3 VHDL 記述

まず、Forwarding Unit の説明をする。Forwarding Unit は基準となる命令の 25〜21bit と 20〜16bit の２つに分けて判定するよう作成した。ここで、基準となる命令とは ID ステージで実行中の命令である。この ID ステージで実行中の命令と、EX ステージ、MEM ステージ、WB ステージで実行中の命令で使用している値と比較する。そしてそれぞれでフォワーディングが発生するかを判定し結果を FOR_FLAG として出力する。以下は 25〜21bit の判定をする Unit である。

フォワーディングユニット１

```
 -- Forwarding_Unit_1
process ( INST_1, INST_2, INST_3, INST_4,  I_COM_2, I_COM_3 )
begin
  -- ジャンプ命令、または NOP
  if( (INST_1( 31 downto 26 ) = "000010" )
  or (INST_1( 31 downto 26 ) = "111111" ) ) then
    FOR_FLAG_1 <= "000";
  -- １つ前の命令が R 形式の時
  elsif( (INST_2( 31 downto 26 ) = "000000")
  and ( INST_2( 15 downto 11 ) = INST_1( 25 downto 21 ) ) ) then
    FOR_FLAG_1 <= "001";
  -- １つ前の命令が ori, andi, lui の時
  elsif(  ( I_COM_1 = '1' )
  and ( INST_2( 20 downto 16 ) = INST_1( 25 downto 21 ) ) ) then
FOR_FLAG_1 <= "001";
  -- ２つ前の命令が R 形式の時
  elsif( (INST_3( 31 downto 26 ) = "000000")
  and ( INST_3( 15 downto 11 ) = INST_1( 25 downto 21 ) ) ) then
    FOR_FLAG_1 <= "010";
  -- ２つ前の命令が ori, andi, lui の時
  elsif(( I_COM_2 = '1' )
  and ( INST_3( 20 downto 16 ) = INST_1( 25 downto 21 ) ) ) then
    FOR_FLAG_1 <= "010";
  -- ３つ前の命令が R 形式の時
  elsif( (INST_4( 31 downto 26 ) = "000000")
  and ( INST_4( 15 downto 11 ) = INST_1( 25 downto 21 ) ) ) then
    FOR_FLAG_1 <= "111";
  -- ３つ前の命令が ori, andi, lui の時
  elsif( ( I_COM_3 = '1' )
  and ( INST_4( 20 downto 16 ) = INST_1( 25 downto 21 ) ) ) then
    FOR_FLAG_1 <= "111";
  else
    FOR_FLAG_1 <= "000";
  end if;
end process;
```

20〜16bit の判定を行う Unit も上記と同様に行う。相違点として、 25〜21bit の際は現在の命令の形式に関係なく（ジャンプ除く）判定が行えたが 20〜16bit の場合、現在の命令の形式に左右される。

また、マルチプレクサから出力する信号を選択する記述を以下に示す。

マルチプレクサの記述

```
 -- Select_Data1
process ( FOR_FLAG_1, REG_RDATA1, ALU_ODATA, ALU_ODATA_3 )
begin
  case FOR_FLAG_1 is
    when "000" =>
  NEW_DATA_1 <= REG_RDATA1;
when "001" =>
  NEW_DATA_1 <= ALU_ODATA;
when "010" =>
  NEW_DATA_1 <= ALU_ODATA_3;
when "111" =>
  NEW_DATA_1 <= REG_WDATA;
when others =>
  NEW_DATA_1 <= ( others => '0' );
end case;
end process;

-- Select_Data2
process ( FOR_FLAG_2, REG_RDATA2, ALU_ODATA, ALU_ODATA_3 )
begin
  case FOR_FLAG_2 is
    when "000" =>
  NEW_DATA_2 <= REG_RDATA2;
when "001" =>
  NEW_DATA_2 <= ALU_ODATA;
when "010" =>
  NEW_DATA_2 <= ALU_ODATA_3;
when "111" =>
  NEW_DATA_2 <= REG_WDATA;
when others =>
  NEW_DATA_2 <= ( others => '0' );
end case;
end process;
```

先程の記述で求めた判定結果（FOR_FLAG）を元にマルチプレクサの出力を決定している。

次に、ID ステージで実行中の命令が lw 命令だった際の処理について述べる。ID ステージの命令が lw 命令である場合、制御線の MEM_TO_REG=1 となりその後 MEM_TO_REG はパイプラインレジスタを通過していく。この事を利用して、MEM_TO_REG がパイプラインレジスタを通過している間、次の命令を読み込まないようにした。これにより見かけ上 NOP が挿入されているように動作する。また、通過している間も IF ステージだった命令がパイプラインレジスタを通過していくが、それは MEM_TO_REG が最後のパイプラインレジスタを通過した際に 1 〜 3 つ目のパイプラインレジスタの値を 0 にする事で解決した。VHDL 記述の主な変更点を以下に示す。

lw 命令の際の動作の記述

```
   -- Instruction Memory ( ROM )
  process ( CR_ADDR , MEM_TO_REG_4) begin
    if ( (INST( 31 downto 26 ) /= "100011")
         or ( (MEM_TO_REG_4 = '0')
                and (MEM_TO_REG_3 = '0')
                and (MEM_TO_REG_2 = '0')
and (MEM_TO_REG = '0') ) ) then
case ( CR_ADDR ) is
  -- Put Machine Language Programme below
when ( H00 & H00 & H00 & H00 ) =>
INST <= "00110000000000000000000000000000";
when ( H00 & H00 & H00 & H04 ) =>
INST <= "00110100001000010000000000000010";

             (略)

-- Pipeline1
process ( CLK, RESET )
begin
  if ( RESET = '1' or MEM_TO_REG_4 /= '0' ) then
    INC_ADDR_1 <= ( others => '0' );
INST_1 <= ( others => '0' );
I_COM_1 <= '0';
MEM_TO_REG <= '0';
  elsif ( CLK'event and CLK = '1' ) then
    INC_ADDR_1 <= INC_ADDR;
INST_1 <= INST;
I_COM_1 <= I_COM;
  end if;
end process;

      (Pipeline2,3 も同様)
```

### 3.4 実行結果

パイプライン化かつハザードフリー化した miniMIPS の動作結果を示す。実行したアセンブラプログラムは以下のとおりである。

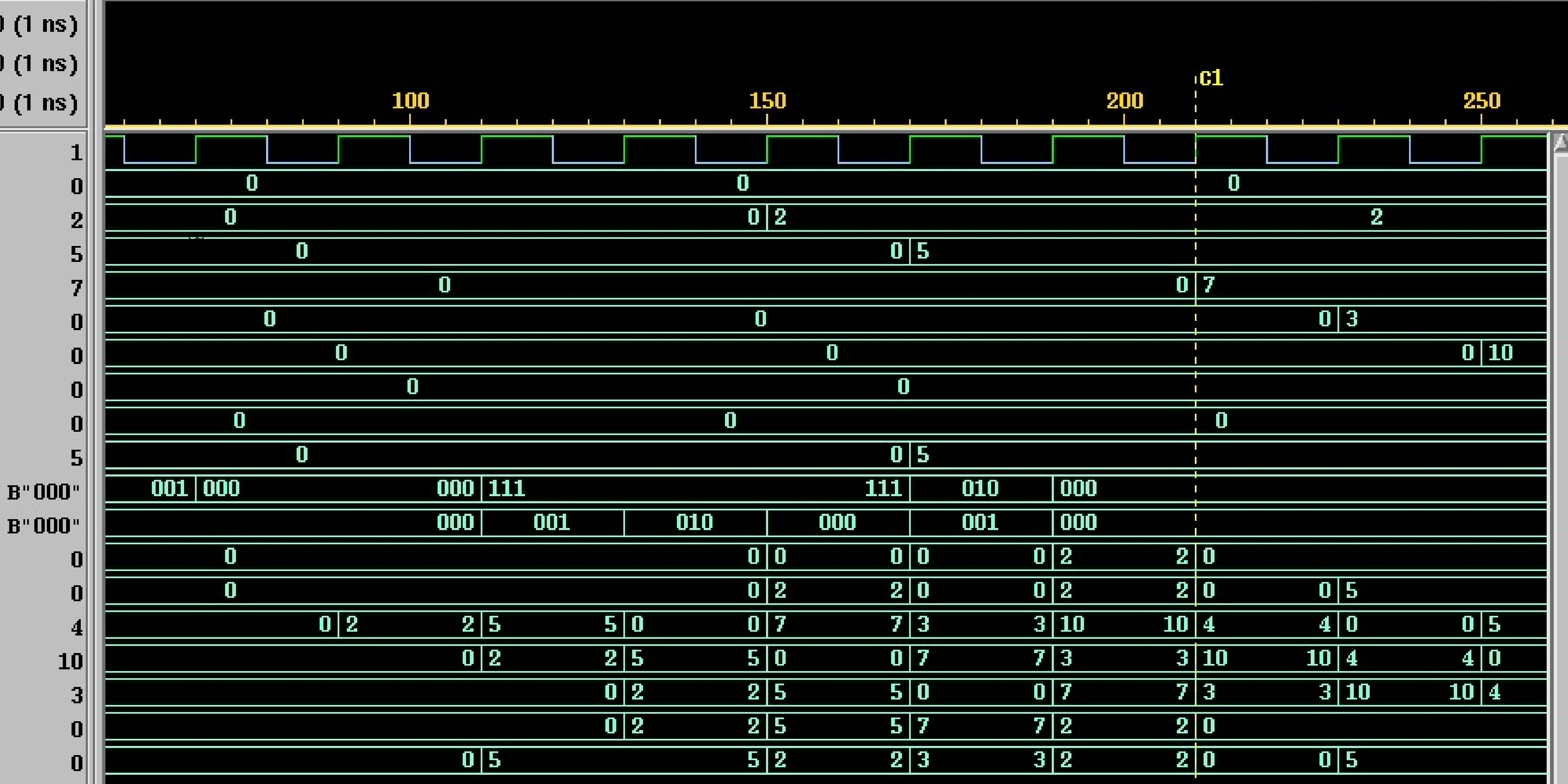
アセンブラプログラム

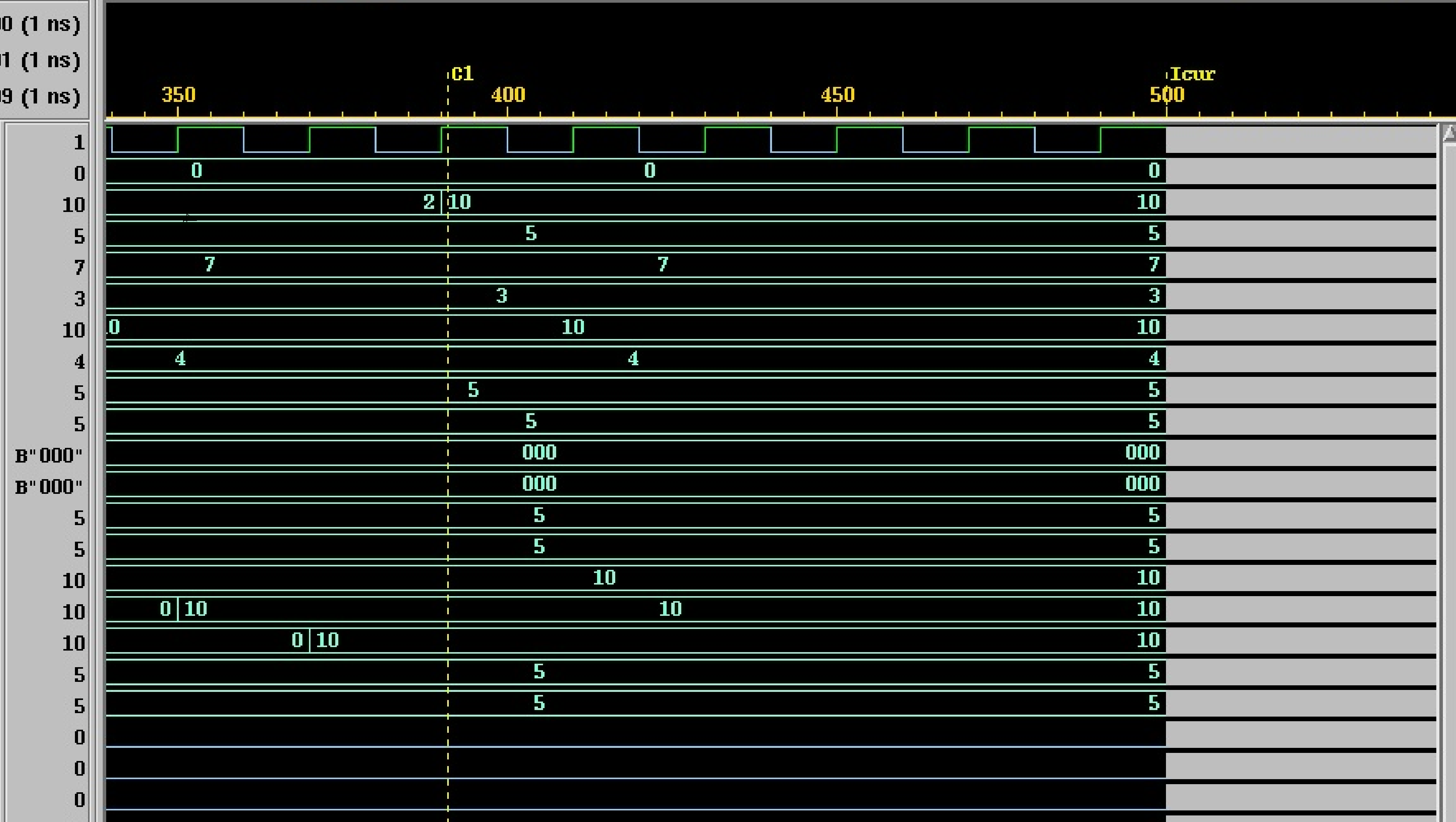
```
andi R00, R00, 0
ori R01, R01, 2
ori R02, R02, 5
sw R02, 0($0)         --ハザード発生
add R03, R01, R02     --ハザード発生
sub R04, R02, R01
add R05, R03, R04     --ハザード発生
add R06, R01, R01
lw R07, 0($0)
add R01, R07, R02     --ハザード発生
```

上のプログラムを考えてみると、もしハザードの対策がとれているなら以下の様な結果になるであろう。

- R00 = 0
- R01 = 10 (途中まで 2 )
- R03 = 7
- R04 = 3
- R05 = 10
- R06 = 4
- R07 = 5
- M00 = 5

次に図 7 10 を見てみると図 7 では、Forwarding Unit での判定通りの出力が信号線に入っているのが分かる。図 8 での最終結果は上の考えと同じ値になっている事が分かり、ハザード対策がとれていることが分かる。図 9 では lw 命令時に、次の命令を読み込んでいない事が分かる。図 10 では MEM_TO_REG がパイプラインレジスタを通過後、その時のパイプラインレジスタの値が 0 にしている事が分かる（写真では一部の値しか載せていないが)。そしてその後に次の命令を実行してレジスタ 1 の値が 10 に変わっている。


VirSim – Waveform – SIM – AutoGroup0
Display
View
Window
Help
EGroup0
SIM
100
150
200
250
C1



VirSim – Waveform – SIM – AutoGroup0
Display
View
Window
Help
EGroup0
SIM
C1
Icur
350
400
450
500
B"000"
B"000"

VirSim – Waveform – SIM – AutoGroup0

m Display View Window Help

EGroup0 SIM

500 (1 ns)
500 (1 ns)
0 (1 ns)

260 280 300 320

1
0 — 0 0 0
10 — 2 2 2
5 — 5 5 5
7 — 7 7 7
3 — 3 3 3
10 — 0 10 10 10
4 — 0 4 4
5 — 0 0 5 5
5 — 5 5 5
0
_2 0
_3 0
_4 0
WN 00000" — 00000000111000100000100000100000
00 00000" — 0111000100000100000100000 | 00000000000000000000000000000000 | 00000000111000100000100000100000
00 00000" — 00 | 00000000111000100000100000100000 | 00000000000000000000000000000000 | 00000000
00 00000" — 00 | *00001110000000000000000 | *0000100000 | 00000000000000000000000000000000
00 00000" — 00 | *0100001001100000010000 | *00001110000000000000000 | 000000000000000000000000

VirSim – Waveform – SIM – AutoGroup0

m Display View Window Help

EGroup0 SIM

500 (1 ns)
398 (1 ns)
102 (1 ns)

300 350 C1 400 450

| | |
|---|---|
| | 1 |
| 0 | 0 |
| 0 | 10 |
| 0 | 5 |
| 0 | 7 |
| 0 | 3 |
| 0 | 10 |
| 0 | 4 |
| 0 | 5 |
| | 5 |
| | 0 |
| _2 | 0 |
| _3 | 0 |
| _4 | 0 |
| WN | 00000" |
| 00 | 00000" |
| 00 | 00000" |
| 00 | 00000" |
| 00 | 00000" |

0 0 0

2 2 10

5 5 5

7 7 7

3 3

10 10

0 4 4 4

0 5 5 5

5 5 5

00000 000000001

100000100000 *00000000000000 0000000011100010000010000010000

00000100000 *0000000000000000000000000 0000000011100010000010000010000

000000 *000 000000000000000000000000000000000 0000000011100010000010000010000

100000 *000000000 00000000000000000000000000000000 000000001110001000001000001000

# 4　評価

## 4.1　自己評価

８５点
毎回出席し、レポートも遅れる事無く提出したからです。１５点引いてあるのはハザード対策が思いつかず、フォワーディングという手法を用いたからです。

## 4.2　教員や TA の評価

９５点
最初は簡単な回路から初め、CAD を受けていなくても入りやすい授業で良かったです。また、講義での説明も分かりやすく、資料も充実していました。

# 参考文献

[1] コンピュータの構成と設計　”パターソン＆ヘネシー”著

[2] VHDL で学ぶ　ディジタル回路設計　”吉田 たけお　尾知 博”著