# 情報工学実験3

## -Mini-MIPS のパイプライン化と分岐予測によるハザード対策-

055755C：真玉橋 朝明

レポート提出日：2007 年 7 月 25 日

# 1 Mini-MIPS の命令の実装

初期状態 (配布された状態には) andi, sll, srl, j の 4 つの命令は未実装である．その命令を実装したブロック図を下に示す．シミュレーション結果を用いて正しく動作していることを確認する．

**ブロック図**

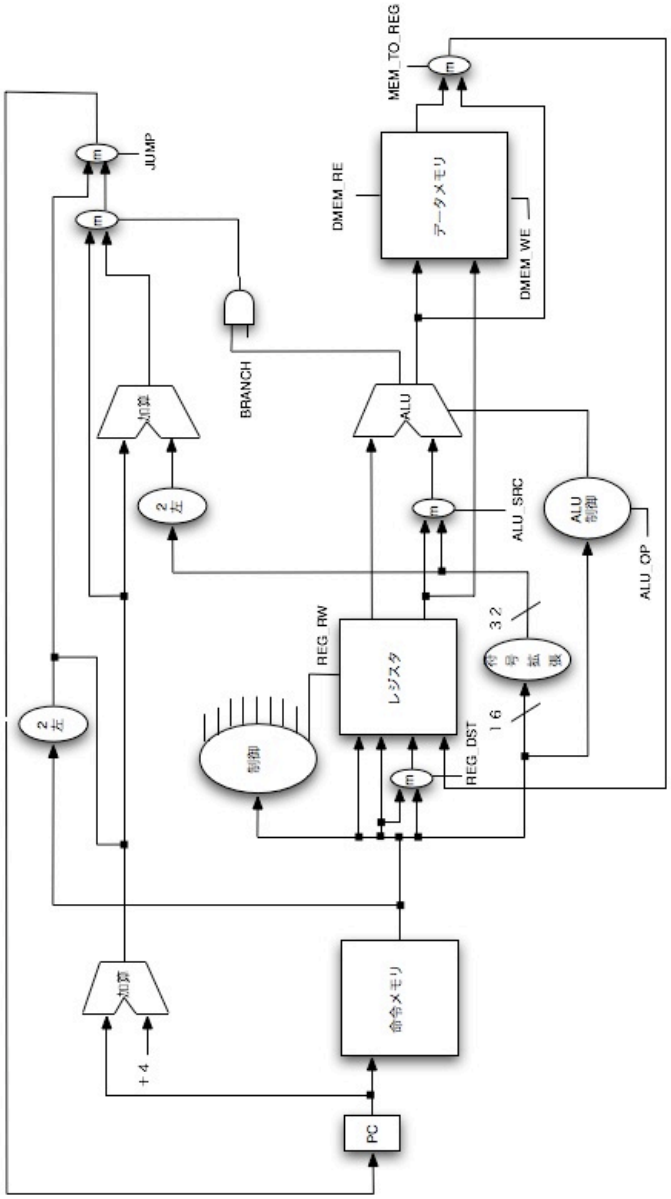


図 1: パイプライン化前のブロック図

## シミュレーション結果

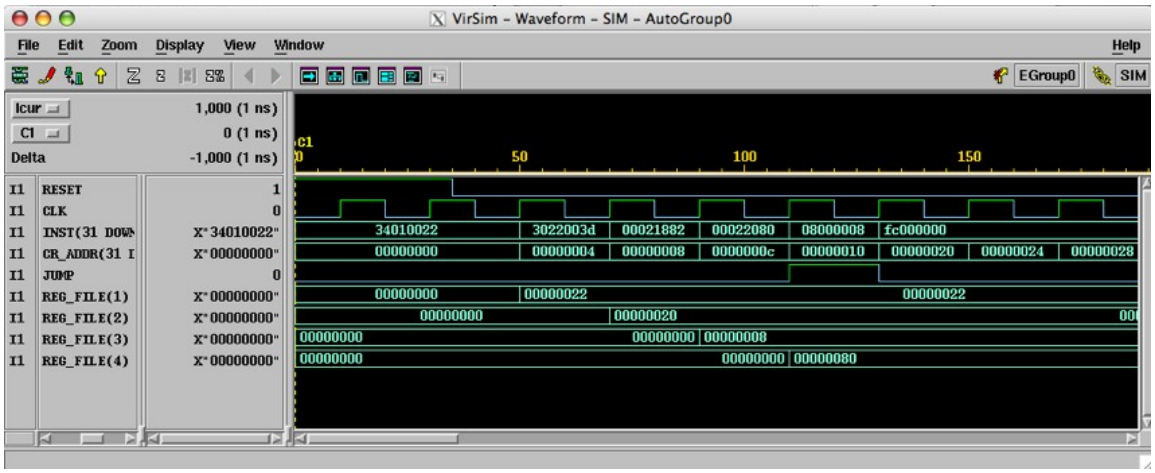


図 2: 4 つの命令の実装確認

### アセンブラプログラム

```
00：ori   R1   R0   34
04：andi  R2   R1   61
08：srl   R3   R2   2
0C：sll   R4   R2   2
10：j     8
```

**00** CR_ADDR は命令のアドレスを示している．CR_ADDR が $(00000000)_{16}$ のとき，プログラム [ORI R1 R0 34] を実行している．レジスタ REG_FILE(1) を見てみると 34 がせっていされているので命令が実現してる．

**04** [ANDI R2 R1 61] を実行している R1 に格納されている値 $(010000010)_2$ と与えられた即値 $(01111101)_2$ の AND をとるとそのあたいは $(010000010)_2 = (20)_{16}$ であり，レジスタ REG_FILE(2) をみると等しい値が設定されている．

**08** レジスタ R3 の値を 2 ビット右シフトした値を R4 に設定する命令である．前の命令により R3 に格納されている値は $(010000010)_2$ になっているこれを 2 ビット右シフトすると $(0001000)_2 = (8)_{16}$ でシミュレーションよりレジスタ REG_FILE(4) を見るとはこの値と一致している

**0C** 次はレジスタ R3 の値を 2 ビット左シフトする．$(010000010)_2$ を 2 ビット左シフトすると $(10000000)_2 = (80)_{16}$ でレジスタ REG_FILE(4) を見るとはこの値と一致している

**10** 無条件分岐．[j 8] 分岐するアドレスは $4 \times 4 + 4 = (20)_{16}$ となっていて CR_ADDR をみるとアドレスが 10 から 20 にジャンプしていることがわかる

よって，4 つの命令を実装したのは正常に動作していることがわかる．

## 2　Mini-MIPS のパイプライン化

**パイプライン後のブロック図**

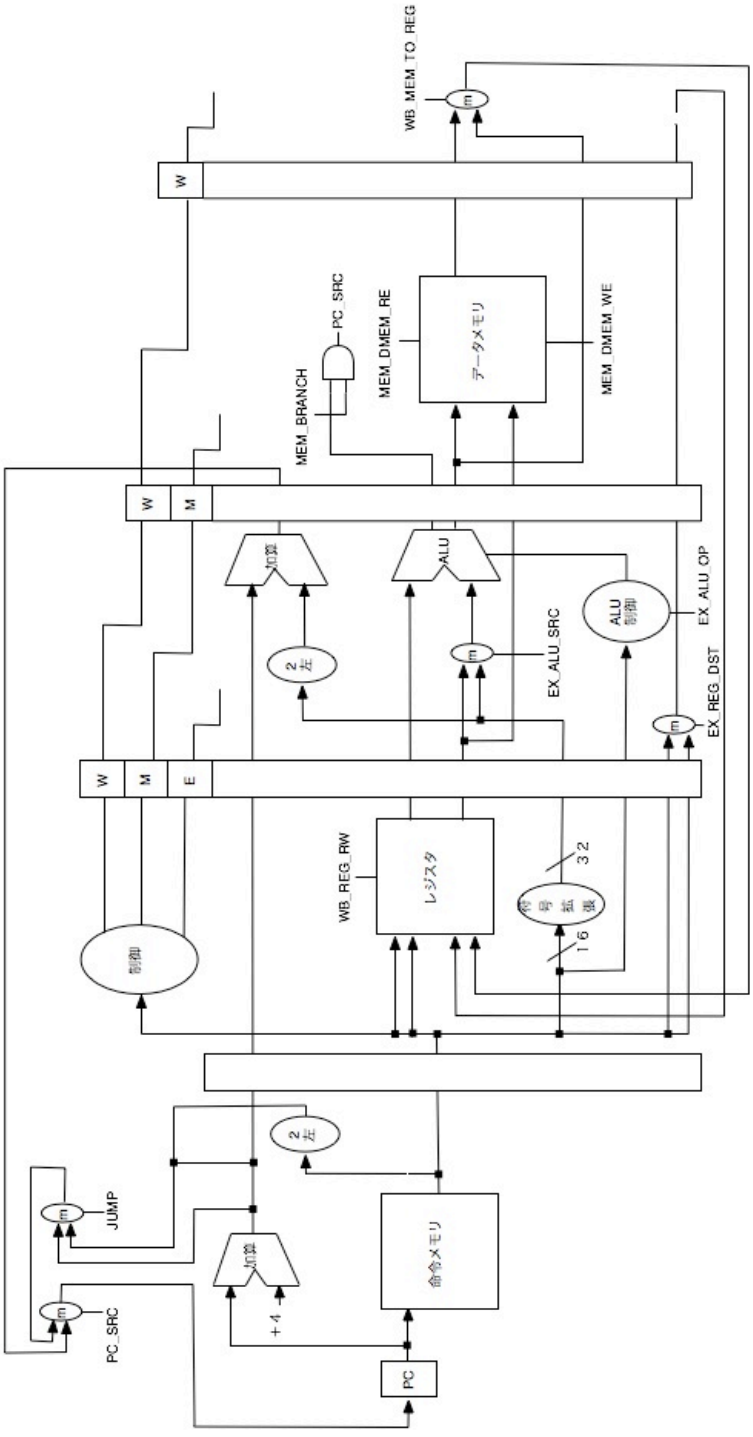


図 3: パイプライン化後

## パイプライン化における変更点

パイプライン化するにあたり処理を命令フェッチ (IF)，命令デコード (ID)，データメモリアクセス (MEM)，演算実行 (EX)，結果の格納 (WB) の 5 つのステージに分けた．そのために各ステージを分ける ID/IF レジスタ，IF/EX レジスタ，EX/MEM レジスタ，MEM/WB レジスタを追加した．クロック (CLK) が入るごとに各レジスタ内の値は更新される．以下のように書くステージは並列に処理される

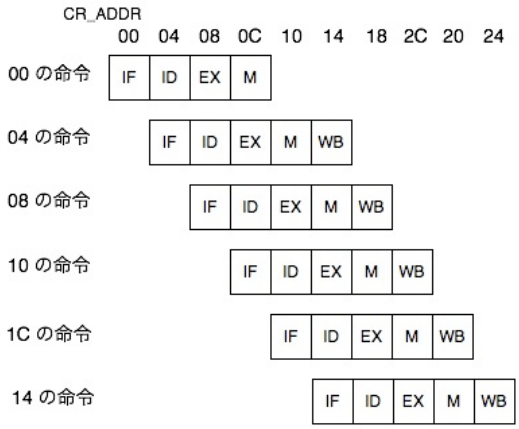


図 4: パイプライン処理

## シミュレーション結果

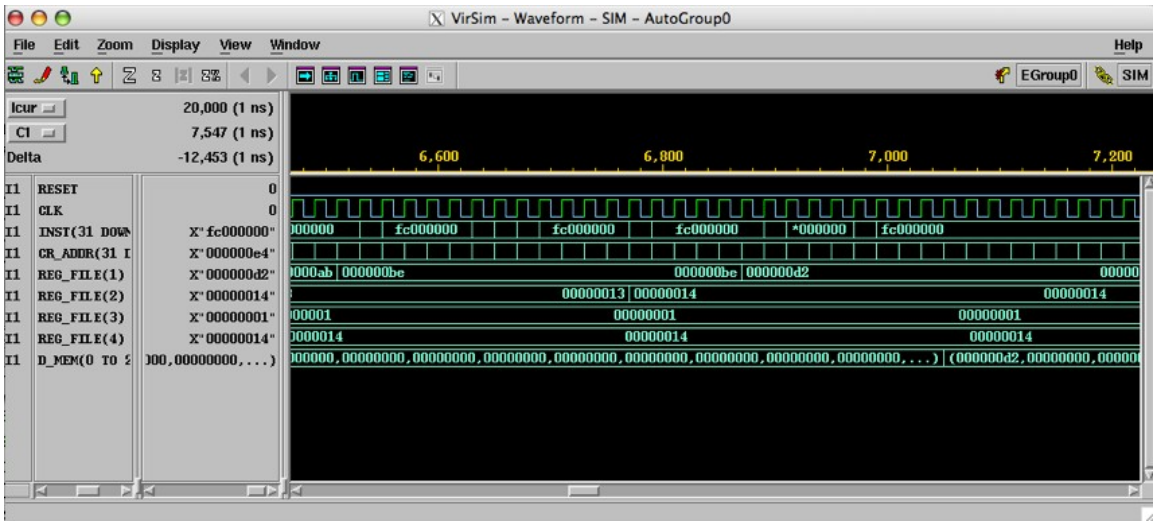


図 5: パイプライン化された回路で総和を求める

### アセンブラプログラム

n までの総和を求めるプログラム．n はレジスタ R4 に $(20)_{10} = (14)_{16}$ という即値を与えている．

```
00：ori   R1   R0   1
04：ori   R2   R0   1
08：ori   R3   R0   1
0C：ori   R4   R0   20
10：nop ×  4
20：add   R2   R2   R3
24：nop ×4
34：add   R1   R1   R2
38：nop ×4
48：beq   R2   R4   9
4C：nop ×4
5C：j     8
60：nop ×4
70：sw   R1   0(R0)
```

命令アドレスの「10〜16」,「24〜30」,「38〜44」ではデータハザード発生防止のため,「4C〜58」,「60〜6C」では制御ハザード発生防止のために NOP を 4 つずつ挟んでいる.「20：ori R2 R2 R3」で R2 に設定する値を大きくしている.「34：add R1 R1 R2」では大きくした数をどんどん足している．そして次の「48：beq R2 R4 9 」で大きくしていった数が「n 」と等しいか判断している．等しいなら「n」まで足し終わったことということでもあり条件に従い分岐する．そのとき R1 には「n」までの総和が格納されている．これを分岐先の命令「70：sw R1 0(R0)」より，メモリに格納する．実際にシミュレーションと比べてみると，REG_FILE(2) の値はどんどん大きくなり $(20)_{10} = (14)_{16}$ まで増加している．REG_FILE(1) ではそのときの REG_FILE(2) の値までの総和が加算されている．そして REG_FILE(2) の値が $(14)_{16}$ までで加算が止まり，REG_FILE(1) に格納されている総和がメモリ D_MEM の初めに格納されている．

# 3　分岐予測を用いたハザードフリー

## パイプラインハザードについてとその回避

パイプラインハザードに関して，主にデータハザード，構造ハザード，制御ハザードの 3 つがあり，Mini-MIPS で起こりうるハザードは，データハザードの一つの RAW ハザードと制御ハザードの 2 つになる．パイプライン化に伴い分岐命令では制御ハザードが起こってしまう．単一サイクルの場合，一命令ずつ処理を行うので分岐判断を行った後に次のアドレスの処理を行うが，パイプライン方式の場合，分岐が起こっても，実際に分岐するかどうか判断する前に並列で次の番地の命令の処理を進めてしまう．今回は解決策として、分岐予測を用いてこれを回避する．これを実現したブロック図は次のようになった．

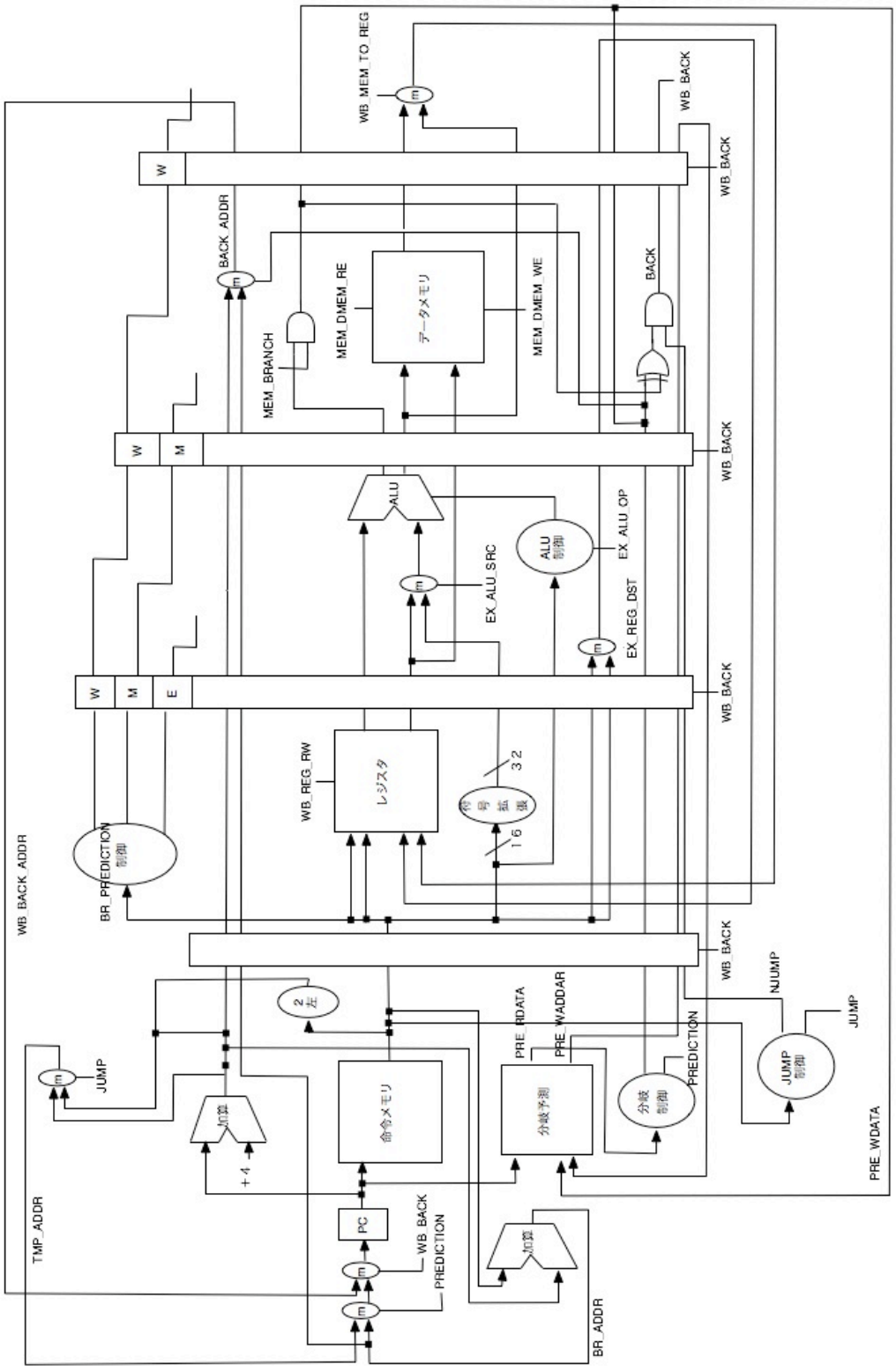


図 6: ハザードフリーのブロック図

## 分岐予測について

分岐が起こるかどうかをあらかじめ予測して処理を進める方法である．しかし，予測がはずれた場合は分岐命令以下の命令は正しくなかったことなり間違った処理でレジスタやメモリへの書き込みを阻止しなければならない．

## ハザードフリー化にともなう変更

- 分岐予測を行うための履歴メモリと予測の制御の追加
- 分岐アドレスを算出する加算器を IF ステージに移動
- 無条件分岐のアドレスの算出と分岐を IF ステージ で行う制御の追加
- 予測失敗時に分岐命令以下の処理を破棄し本来の次命令先を PC 設定する制御の追加

## 分岐メモリを用いた予測

分岐の履歴を保存するメモリを作り，実際に分岐したかどうかを履歴としてメモリに格納する．命令のアドレスから対応する履歴出力する．今回は，以下のような履歴を作った．

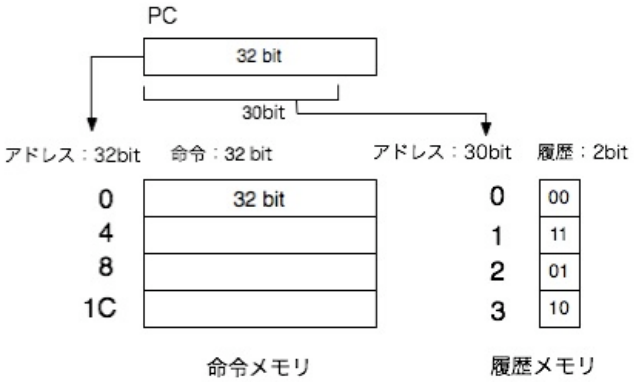


図 7: 履歴メモリと命令アドレス

全ての命令について対応している履歴メモリを作った．命令メモリが 32 bit でアドレスが 4 番地飛ばしであるのに対して履歴メモリのデータは 2 bit なのでアドレスは 1 番地ごとでよく 4 番飛ばしなので命令のアドレスの下位 2 bit を切り捨てることで，1 番地ごとに対応できるように変換できる．なので，メモリ自体は 4 分の 1 に軽量化できる．2 bit のデータなので前回と前々回の分の 2 回分の記録を保存している．左に 1 bit シフトしてデータを更新する．

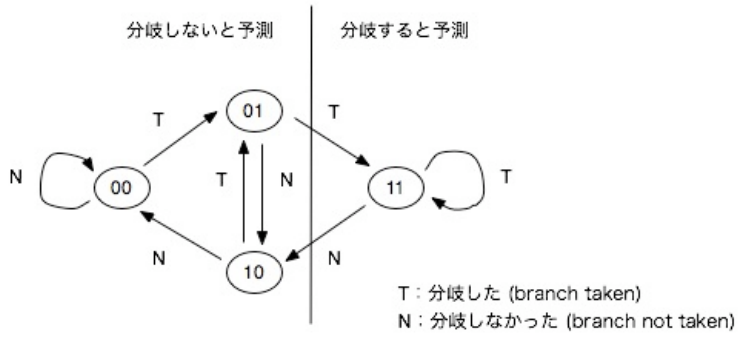


図 8: 履歴と制御

履歴メモリから出力された履歴から分岐の予測を行う．データが”11”のときのみ分岐すると予測する．これは，条件分岐の特性を考えた結果こうなった．まず，ループの場合条件を満たすまで JUMP など使い処理を繰り返す．条件を満たすまで分岐しないのだから前回が ’0’ ならいつ分岐するかわからないのに分岐するよりも今回も分岐しないと判断する．よって，”00”,”10” の場合は分岐しない．また，前回が分岐しても今回もそうとは限らない．例えば，条件を満たしてループを抜けても次に同じループに入るときは初期化されている可能性が高いと考える．よって，”01”でも分岐しないと予測する．”11”の場合ループであっても前回も前々回も分岐しているのでこれからも常に条件を満たしているだろうと考える．よって”11”のときは分岐すると予測する．基本的にはプログラムを見て総合的に分岐する方が少ない．よって，いつくるかわからない分岐を予測するよりは，失敗して始めて分岐の可能性を考えるような条件にした．

### VHDL による記述

```
--Reference Branch History

process (CR_ADDR) begin
  PRE_RADDR <= CR_ADDR(31 downto 2);
end process;

process (CLK, RESET)
begin  -- process
  if (RESET = '1') then
    for I in 0 to 2**6-1 loop
      P_MEM(I) <= ( others => '0');
    end loop;
  elsif (CLK'event and CLK = '1') then
    P_MEM(CONV_INTEGER(WB_PRE_WADDR)) <= PRE_WDATA;
  end if;
end process;
```

```
  PRE_RDATA <= P_MEM(CONV_INTEGER(PRE_RADDR));
  PRE_WADDR <= PRE_RADDR;
```

命令アドレスの下位2 bit を切り捨てて，29 bit のアドレスに変換．そのアドレスから履歴のデータを読み込んでいる．また，WB ステージでのデータを書き込んでいる．

#### VHDL による記述

```
  process ( PRE_RDATA(0) ) begin
    if ( PRE_RDATA = "11") then
  PREDICTION <= '1';
  BR_PREDICTION <= '1' & PRE_RDATA(0);
else
  PREDICTION <= '0';
  BR_PREDICTION <= '0' & PRE_RDATA(0);
end if;
  end process;
```

出力された履歴データから予測をしている．「BR_PREDICTION」は次の命令アドレスが分岐先かどうかを予測する．「BR_PREDICTION」2 bit のデータで2 bit 目は分岐の予測，1 bit 目は実際の分岐の履歴を更新するための前回のデータとなっている．実際の分岐結果とあわして2 bit になり履歴を更新する．

### JUMP 制御

JUMP は無条件に分岐を行うので，制御ハザードがかなりの高確率で発生する．しかし，必ず分岐するのでデータによる予測を行う必要がなく．命令フェッチのステージで読み込んだ命令がJUMP かどうか判断し，JUMP なら，指定されたアドレスに即分岐するようにすればいい．

#### VHDL による記述

```
 --JUMP CONTROL

process (INST( 31 downto 26)) begin
 if (INST(31 downto 26) = "000010") then
   JUMP <= '1';
   NJUMP <= '0';
 else
   JUMP <= '0';
   NJUMP<= '1';
```

```
 end if;
end process;
```

命令の上位 6 bit から JUMP 命令かどうか判断している.「JUMP」は次の命令アドレスを命令で指定したアドレスかどうか判断する.「NJUMP」は「JUMP」の否定をとっていて，分岐予測の予測と結果の比較に無条件分岐による誤差をもたらさないために使用される.

## 予測失敗時の対処

実際に分岐予測と本来の分岐の組み合わせは 4 種類となり、

- 「分岐しない」と予測して実際にも「分岐しない」
- 「分岐しない」と予測したが実際は「分岐する」
- 「分岐する」と予測したが実際は「分岐しない」
- 「分岐する」と予測して実際にも「分岐する」

1 と 4 は予測と実際の処理が一致しているので問題はない．しかし，2 と 3 では予測の分岐と本来の分岐が一致しておらず問題が起こる．このとき分岐命令以下の処理は行うべきではなく，破棄しなければならない．また，プログラムカウンタの値も本来のアドレスにしなければならない.

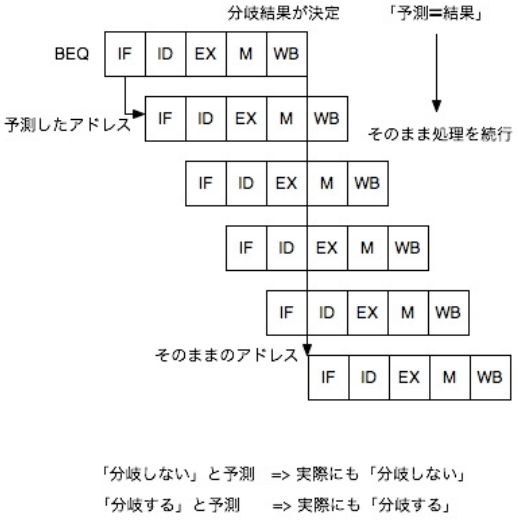


図 9: 予測成功のときの処理

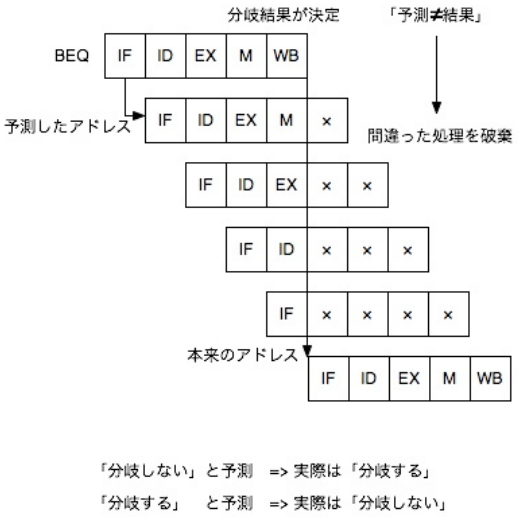


図 10: 予測失敗のときの処理

**VHDL による記述**

```
-- Next Address Selector
PC_SRC <= MEM_ALU_ZF and MEM_BRANCH;
-- 予測と結果の一致判断
RESULT <= PC_SRC xor MEM_BR_PREDICTION(1);
-- JUMP によるご認識がないよう見極める
BACK   <= RESULT and MEM_NJUMP;
```

「MEM_ALU_ZF」は ALU のゼロ判定,「MEM_BRANCH」は命令が条件分岐かどうかを示してる.「PC_SRC」はこれの論理積をとり，”1”ならつまり分岐命令が出ていて実際に分岐するということになる.

「RESLUT」予測と結果が等しいかどうかを示す.「MEM_BR_PREDICTION」は分岐の予測を示していて，先で求めた「PC_SRC」との XOR をとる．どちらかが”1”で”0”のとき”1”になる．つまり，結果と予測が一致しないときである.

「RESULT」で ”1”がでたなら予測失敗ということになるが，JUMP 命令による分岐の場合も考え JUMP 命令がでたなら”0”を出す「MEM_NJUMP」と AND をとる．JUMP 命令が出ていなくて，予測と結果が一致しないなら予測は失敗していたといえるので，失敗のため処理の破棄を命じる信号「BACK」を出す．WB ステージで WB_BACK として分岐命令以下の処理を破棄する.

**VHDL による記述**

```
--IF/ID
  process (CLK, RESET) begin
```

```
    if ( RESET = '1' ) then
      ID_INC_ADDR      <= (others => '0');
      ID_BR_ADDR       <= (others => '0');
      ID_INST          <= (others => '0');
      ID_NJUMP         <= '0';
      ID_BR_PREDICTION <= (others => '0');
      ID_PRE_WADDR     <= (others => '0');
    elsif ( CLK'event and CLK = '1' ) then
  if (WB_BACK = '1') then
        ID_INC_ADDR      <= ( others => '0');
        ID_BR_ADDR       <= ( others => '0');
        ID_INST          <= ( others => '0');
        ID_NJUMP         <= '0';
        ID_BR_PREDICTION <= ( others => '0');
        ID_PRE_WADDR     <= ( others => '0');
  else
        ID_INC_ADDR      <= INC_ADDR;
        ID_BR_ADDR       <= BR_ADDR;
        ID_INST          <= INST;
        ID_NJUMP         <= NJUMP;
        ID_BR_PREDICTION <= BR_PREDICTION;
        ID_PRE_WADDR     <= PRE_WADDR;
  end if;
    end if;
end process;
```

失敗時の命令の破棄は WB ステージで行う．よって予測失敗を示す「BACK」は　MEM/WB レジスタにより「WB_BACK」となる．これが '1' のとき RESET = '1' のようにレジスタで更新される全ての値が初期化される．

## VHDL による記述

```
  process ( MEM_BR_PREDICTION(1),MEM_INC_ADDR, MEM_BR_ADDR ) begin
    if ( MEM_BR_PREDICTION(1) = '1' ) then
      BACK_ADDR <= MEM_INC_ADDR;
    else
      BACK_ADDR <= MEM_BR_ADDR;
    end if;
  end process;

～～
```

```
--Next Address
process (WB_BACK_ADDR, NBACK_ADDR, WB_BACK) begin
  if (WB_BACK = '1') then
    NT_ADDR <= WB_BACK_ADDR;
  else
    NT_ADDR <= NBACK_ADDR;
  end if;
end process;
```

分岐に失敗したとき，次のアドレスは実行していなかった方になる．よって，もし，失敗したとき次のアドレスになるのは分岐していなかったときは分岐先の「MEM_BR_ADDR」分岐してなかったら次の番地であった「MEM_INC_ADDR」となる．予測した逆のアドレスを失敗時のときのために「BACK_ADDR」として設定しておく．そして次のステージで分岐の正否を決める「WB_BACK」によって，次のアドレス「NT_BACK」を決定する．

## シミュレーション

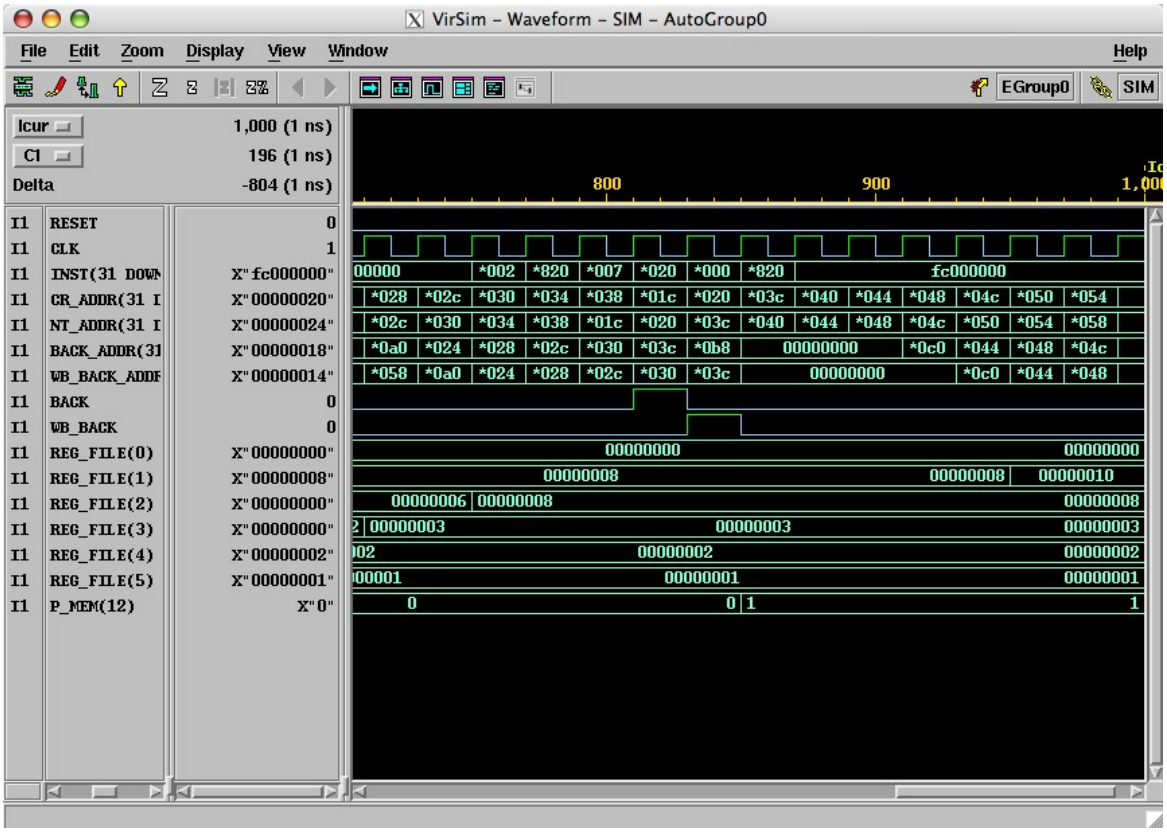


図 11: ハザードフリーの確認

**アセンブラプログラム**

```
00：ori  R1  R0  8
04：ori  R4  R0  2
08：ori  R5  R0  1
0C：nop ×4
1C：add  R2  R2 R4
20：nop ×4
30：beq  R1  R2  2
34：add  R3  R3 R5
38：JUMP  2
3C：add  R1  R1 R2
```

R1 に 8 設定し，R2 に 2 を繰り返し加算して R1 = R2 になったらループを抜けて R1 に R1 と R2 を加算した値を格納する簡単なプログラム．データハザードが発生されると思われる「0C〜10」,「20〜2C」には NOP を挟んで対応している.「00〜08」では ORI 命令によって値を設定.「1C〜38」で加算と条件分岐のループが出来ている．また,「34：add R3 R3 R5」と JUMP の数をカウントしている R1 = R2 となるのは，R2 に 4回 2 を加算したときで等しくなった時点で分岐するので分岐以下の命令 「34：add R3 R3 R5」と JUMP は実行されない．よって，R3 の値は 3 で止まり，R1 = R2 = 8 となる．最後の命令で R1 と R2 の値を加算して R1 に格納するので最終的に R1 = $(10)_{16}$ となる．これをシミュレーションと比較すると等価な値になっていることがわかる．

基本的に常に分岐しないので，分岐するときは値を初期化し，アドレスも分岐先にしなければならない．シミュレーションをみてみると，始めて分岐条件をみたし予測失敗のときに初期化及びアドレスを制御する BACK が '1' になっている．これにより次のクロックで WB_BACK = '1' となり，次のアドレスは失敗時の戻り値アドレス BACK_ADDR 次のクロックでは 各レジスタの値が初期化される． 800ns から 900ns あたりをみてみると，WB_BACK が'1' になったとき次のクロックで CR_ADDR のアドレスが修正され，BACK_ADDR や WB_BACK_ADDR の値が初期化され 「30：beq R1 R2 2」の分岐履歴を示す P_MEM(12) の値が '1' となっているのがわかる．これより，この回路が正常に動作しかつハザードを回避できていることがわかる．

# 4　作成したプログラムのソース

mM-SC-BC.vhd：４つの命令を実装時のファイル
ganmo.fts.ie.u-ryukyu.ac.jp:/home/vlsi04/Final/mini/
mM-SC-BC-P.vhd：パイプライン化したファイル
ganmo.fts.ie.u-ryukyu.ac.jp:/home/vlsi04/Final/pipe/
mM-SC-BC-BP.vhd：分岐予測を実装したファイル
ganmo.fts.ie.u-ryukyu.ac.jp:/home/vlsi04/Final/BP/

## 5　本実験に対する自分の総合成績

普段の実験でも話を集中してきていたし，出席状況も良かった．最終課題に関しては自分なりにがんばれたと思う．期限に間に合わせることが出来なかったのはやはりまだまだだと思う．プログラムを正常に動作させるのにかなり時間かかった．よくこんなにがんばれたなとも思うし，こんなに時間がかかって情けないとも思った．がんばったぶんとして 90 点くらいあると思う．そこから期限切れのぶんを減点して総合的な評価は 70 点とする．

## 6　教員や TA の評価

実験に関しての説明はわかり易かったし，基礎的なとこから丁寧に教えてくれたと思う．最終課題に関しても方向性がわかるように指導してくれた．TA の先輩もわからないところは教えてくれた．最終課題について最後のぎりぎりまで提出をまってもらったのは本当にありがたかった．特に文句はないので評価は 100 点 とする ．ただ，自主勉強について「コンピュータの構成と設計」は理解できてきたらいい本であると思うようになるけど基礎知識があまりない状態で章を 1 週間で読むのはきつかった．