# Design and Implementation of CELL SPU mini- ISA

ESE 545 Project Report

Atif Iqbal Ahangar (aahangar - 111416569)

Karthik Raj (karamachandr - 111675685)

## Table of Contents

## Tables

## Figures

## 1 Cell SPU Mini-ISA Instructions and Encodings

This section details different Instructions our implementation supports and different conventions and encoding we have used.

*Table 1 Pipe Index Mapping*

| Pipe | Index |
|---|---|
| Even | 0 |
| Odd | 1 |

*Table 2 Unit Index Mapping*

| Unit | Index |
|---|---|
| Simple Fixed | 1 |
| Shift | 2 |
| Single Precision | 3 |
| Byte | 4 |
| Permute | 5 |
| Local Store | 6 |
| Branch | 7 |

*Table 3 SPU mini instructions*

| No. | Name | Usage | Pipe | Pipe No. | Unit | Unit No. | Latency |
|---|---|---|---|---|---|---|---|
| **Memory-Load/Store Instructions** | | | | | | | |
| **1** | Load Quadword (d-form) | **`lqd rt, symbol(ra)`**<br>1. `LSA ← (RepLeftBit(I10 \|\| 0b0000,32) + RA0:3) & 0xFFFFFFF0`<br>2. `RT ← LocStor(LSA, 16)` | Odd | 1 | Load Store | 6 | 6 |
| **2** | Load Quadword (a-form) | **`lqa rt, symbol`**<br>1. `LSA ← RepLeftBit(I16 \|\| 0b00,32) & 0xFFFFFFF0`<br>2. `RT ← LocStor(LSA,16)` | Odd | 1 | Load Store | 6 | 6 |
| **3** | Store Quadword (d-form) | **`stqd rt, symbol(ra)`**<br>1. `LSA ← (RepLeftBit(I10 \|\| 0b0000,32) + RA0:3) & 0xFFFFFFF0`<br>2. `LocStor(LSA,16) ← RT` | Odd | 1 | Load Store | 6 | 6 |
| **4** | Store Quadword (a-form) | **`stqa rt, symbol`**<br>1. `LSA ← RepLeftBit(I16 \|\| 0b00,32) & LSLR & 0xFFFFFFF0`<br>2. `LocStor(LSA,16) ← RT` | Odd | 1 | Load Store | 6 | 6 |
| **5** | Immediate Load Halfword | **`ilh rt, symbol`**<br>1. `s ← I16`<br>2. `RT0:1 ← s`<br>3. `RT2:3 ← s`<br>4. `RT14:15 ← s` | Even | 0 | Simple Fixed | 1 | 2 |
| **6** | Immediate Load Word | **`Il rt, symbol`**<br>1. `t ← RepLeftBit(I16,32)`<br>2. `RT0:3 ← t`<br>3. `RT4:7 ← t`<br>4. `RT8:11 ← t`<br>5. `RT12:15 ← t` | Even | 0 | Simple Fixed | 1 | 2 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **7** | Immediate Load Address | **`ila rt, symbol`**<br>`1. t ← 140 \|\| I18`<br>`2. RT0:3 ← t`<br>`3. RT4:7 ← t`<br>`4. RT8:11 ← t`<br>`5. RT12:15 ← t` | Even | 0 | Simple Fixed | 1 | 2 |
| **Integer and Logical Instructions** | | | | | | | |
| **8** | Add Half Word | **`ah rt, ra, rb`**<br>`1. RT0:1 ← RA0:1 + RB0:1`<br>`2. RT2:3 ← RA2:3 + RB2:3`<br>`3. RT4:5 ← RA4:5 + RB4:5`<br>`4. RT6:7 ← RA6:7 + RB6:7`<br>`5. RT8:9 ← RA8:9 + RB8:9`<br>`6. RT10:11 ← RA10:11 + RB10:11`<br>`7. RT12:13 ← RA12:13 + RB12:13`<br>`8. RT14:15 ← RA14:15 + RB14:15` | Even | 0 | Simple Fixed | 1 | 2 |
| **9** | Add Half Word Immediate | **`ahi rt, ra, value`**<br>`1. s ← RepLeftBit(I10,16)`<br>`2. RT0:1 ← RA0:1 + s`<br>`3. RT2:3 ← RA2:3 + s`<br>`4. RT4:5 ← RA4:5 + s`<br>`5. RT6:7 ← RA6:7 + s`<br>`6. RT8:9 ← RA8:9 + s`<br>`7. RT10:11 ← RA10:11 + s`<br>`8. RT12:13 ← RA12:13+ s`<br>`9. RT14:15 ← RA14:15 + s` | Even | 0 | Simple Fixed | 1 | 2 |
| **10** | Add Word | **`a rt, ra, rb`**<br>`1. RT0:3 ← RA0:3 + RB0:3`<br>`2. RT4:7 ← RA4:7 + RB4:7`<br>`3. RT8:11 ← RA8:11 + RB8:11`<br>`4. RT12:15 ← RA12:15 + RB12:15` | Even | 0 | Simple Fixed | 1 | 2 |
| **11** | Add Word Immediate | **`ai rt, ra, value`**<br>`1. t ← RepLeftBit(I10,32)`<br>`2. RT0:3 ← RA0:3 + t`<br>`3. RT4:7 ← RA4:7 + t`<br>`4. RT8:11 ← RA8:11 + t`<br>`5. RT12:15 ← RA12:15 + t` | Even | 0 | Simple Fixed | 1 | 2 |
| **12** | Subtract from Halfword | **`sfh rt, ra, rb`**<br>`1. RT0:1 ← RB0:1 + (¬RA0:1) + 1`<br>`2. RT2:3 ← RB2:3 + (¬RA2:3) + 1`<br>`3. RT4:5 ← RB4:5 + (¬RA4:5) + 1`<br>`4. RT6:7 ← RB6:7 + (¬RA6:7) + 1`<br>`5. RT8:9 ← RB8:9 + (¬RA8:9) + 1`<br>`6. RT10:11 ← RB10:11 + (¬RA10:11) +1`<br>`7. RT12:13 ← RB12:13 + (¬RA12:13) +1`<br>`8. RT14:15 ← RB14:15 + (¬RA14:15) +1` | Even | 0 | Simple Fixed | 1 | 2 |
| **13** | Subtract from Halfword Immediate | **`sfhi rt, ra, value`**<br>`1. t ← RepLeftBit(I10,16)`<br>`2. RT0:1 ← t + (¬RA0:1) + 1`<br>`3. RT2:3 ← t + (¬RA2:3) + 1`<br>`4. RT4:5 ← t + (¬RA4:5) + 1`<br>`5. RT6:7 ← t + (¬RA6:7) + 1`<br>`6. RT8:9 ← t + (¬RA8:9) + 1`<br>`7. RT10:11 ← t + (¬RA10:11) + 1`<br>`8. RT12:13 ← t + (¬RA12:13) + 1`<br>`9. RT14:15 ← t + (¬RA14:15) + 1` | Even | 0 | Simple Fixed | 1 | 2 |
| **14** | Subtract from Word | **`sf rt, ra, rb`**<br>`1. RT0:3 ← RB0:3 + (¬RA0:3) + 1`<br>`2. RT4:7 ← RB4:7 + (¬RA4:7) + 1`<br>`3. RT8:11 ← RB8:11 + (¬RA8:11) + 1`<br>`4. RT12:15 ← RB12:15 + (¬RA12:15) + 1` | Even | 0 | Simple Fixed | 1 | 2 |
| **15** | Subtract from Word Immediate | **`sfi rt, ra, value`**<br>`1. t ← RepLeftBit(I10,32)`<br>`2. RT0:3 ← t + (¬RA0:3) + 1` | Even | 0 | Simple Fixed | 1 | 2 |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | 3. RT4:7 ← t + (¬RA4:7) + 1<br>4. RT8:11 ← t + (¬RA8:11) + 1<br>5. RT12:15 ← t + (¬RA12:15) + 1 | | | | | |
| **16** | Add Extended | **addx rt, ra, rb**<br>1. RT0:3 ← RA0:3 + RB0:3 + RT31<br>2. RT4:7 ← RA4:7 + RB4:7 + RT63<br>3. RT8:11 ← RA8:11 + RB8:11 + RT95<br>4. RT12:15 ← RA12:15 + RB12:15 + RT127 | Even | 0 | Simple Fixed | 1 | 2 |
| **17** | Subtract from Extended | **sfx rt, ra, rb**<br>1. RT0:3 ← RB0:3 + (¬RA0:3) + RT31<br>2. RT4:7 ← RB4:7 + (¬RA4:7) + RT63<br>3. RT8:11 ← RB8:11 + (¬RA8:11) + RT95<br>4. RT12:15 ← RB12:15 + (¬RA12:15) + RT127 | Even | 0 | Simple Fixed | 1 | 2 |
| **18** | Carry Generate | **cg rt, ra, rb**<br>1. for j = 0 to 15 by 4<br>    t0:32 = ((0 \|\| RAj::4) + (0\|\|RBj::4))<br>    RTj::4 ← 310 \|\| t0<br>  End | Even | 0 | Simple Fixed | 1 | 2 |
| **19** | Borrow Generate | **bg rt, ra, rb**<br>1. for j = 0 to 15 by 4<br>    if (RBj::4 ≥u RAj::4) then RTj::4 ← 1<br>    else RTj::4 ← 0<br>  end | Even | 0 | Simple Fixed | 1 | 2 |
| **20** | Multiply | **mpy rt,ra,rb**<br>1. RT0:3 ← RA2:3 * RB2:3<br>2. RT4:7 ← RA6:7 * RB6:7<br>3. RT8:11 ← RA10:11 * RB10:11<br>4. RT12:15 ← RA14:15 * RB14:15 | Even | 0 | Single Precision | 3 | 7 |
| **21** | Multiply Immediate | **mpyi rt,ra,value**<br>1. t ← RepLeftBit(I10,16)<br>2. RT0:3 ← RA2:3 * t<br>3. RT4:7 ← RA6:7 * t<br>4. RT8:11 ← RA10:11 * t<br>5. RT12:15 ← RA14:15 * t | Even | 0 | Single Precision | 3 | 7 |
| **22** | Multiply and Add | **mpya rt,ra,rb,rc**<br>1. t0 ← RA2:3 * RB2:3<br>2. t1 ← RA6:7 * RB6:7<br>3. t2 ← RA10:11 * RB10:11<br>4. t3 ← RA14:15 * RB14:15<br>5. RT0:3 ← t0 + RC0:3<br>6. RT4:7 ← t1 + RC4:7<br>7. RT8:11 ← t2 + RC8:11<br>8 .RT12:15 ← t3 + RC12:15 | Even | 0 | Single Precision | 3 | 7 |
| **23** | Count Leading Zeros | **clz rt,ra**<br>1. for j = 0 to 15 by 4<br>    t ← 0<br>    u ← RAj::4<br>    for m = 0 to 31<br>     If um = 1 then leave<br>       t ← t + 1<br>    end<br>    RTj::4 ← t<br>   end | Even | 0 | Simple Fixed | 1 | 2 |
| **24** | Count Ones in Bytes | **cntb rt,ra**<br>1. for j = 0 to 15<br>    c = 0<br>    b ← RAj<br>    For m = 0 to 7<br>      If bm = 1 then c ← c + 1<br>    end<br>    RTj ← c<br>  end | Even | 0 | Byte | 4 | 4 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **25** | Average Bytes | **`avgb rt,ra,rb`**<br>`1. for j = 0 to 15`<br>`     RTj ← ((0x00 \|\| RAj) + (0x00 \|\| RBj) + 1)7:14`<br>`   end` | Even | 0 | Byte | 4 | 4 |
| **26** | Absolute Difference of Bytes | **`absdb rt,ra,rb`**<br>`1. for j = 0 to 15`<br>`    if (RBj >u RAj) then`<br>`        RTj ← RBj - RAj`<br>`    else RTj ← RAj - RBj`<br>`   end` | Even | 0 | Byte | 4 | 4 |
| **27** | Sum Bytes into Halfword | **`sumb rt,ra,rb`**<br>`1. RT0:1 ← RB0 + RB1+ RB2 + RB3`<br>`2. RT2:3 ← RA0 + RA1+ RA2 + RA3`<br>`3. RT4:5 ← RB4 + RB5+ RB6 + RB7`<br>`4. RT6:7 ← RA4 + RA5+ RA6 + RA7`<br>`5. RT8:9 ← RB8 + RB9+ RB10 + RB11`<br>`6. RT10:11 ← RA8 + RA9+ RA10 + RA11`<br>`7. RT12:13 ← RB12 + RB13+ RB14 +RB15`<br>`8. RT14:15 ← RA12 + RA13+ RA14 +RA15` | Even | 0 | Byte | 4 | 4 |
| **28** | And | **`and rt,ra,rb`**<br>`1. RT0:3 ← RA0:3 & RB0:3`<br>`2. RT4:7 ← RA4:7 & RB4:7`<br>`3. RT8:11 ← RA8:11 & RB8:11`<br>`4. RT12:15 ← RA12:15 & RB12:15` | Even | 0 | Simple Fixed | 1 | 2 |
| **29** | And Word Immediate | **`andi rt,ra,value`**<br>`1. b ← I10 & 0x00FF`<br>`2. bbbb ← b \|\| b \|\| b \|\| b`<br>`3. RT0:3 ← RA0:3 & bbbb`<br>`4. RT4:7 ← RA4:7 & bbbb`<br>`5. RT8:11 ← RA8:11 & bbbb`<br>`6. RT12:15 ← RA12:15 & bbbb` | Even | 0 | Simple Fixed | 1 | 2 |
| **30** | And Halfword Immediate | **`andhi rt,ra,value`**<br>`1. t ← RepLeftBit(I10,16)`<br>`2. RT0:1 ← RA0:1 & t`<br>`3. RT2:3 ← RA2:3 & t`<br>`4. RT4:5 ← RA4:5 & t`<br>`5. RT6:7 ← RA6:7 & t`<br>`6. RT8:9 ← RA8:9 & t`<br>`7. RT10:11 ← RA10:11 & t`<br>`8. RT12:13 ← RA12:13 & t`<br>`9. RT14:15 ← RA14:15 & t` | Even | 0 | Simple Fixed | 1 | 2 |
| **31** | And Byte Immediate | **`andbi rt,ra,value`**<br>`1. t ← RepLeftBit(I10,32)`<br>`2. RT0:3 ← RA0:3 & t`<br>`3. RT4:7 ← RA4:7 & t`<br>`4. RT8:11 ← RA8:11 & t`<br>`5. RT12:15 ← RA12:15 & t` | Even | 0 | Simple Fixed | 1 | 2 |
| **32** | Or | **`or rt,ra,rb`**<br>`1. RT0:3 ← RA0:3 \| RB0:3`<br>`2. RT4:7 ← RA4:7 \| RB4:7`<br>`3. RT8:11 ← RA8:11 \| RB8:11`<br>`4. RT12:15 ← RA12:15 \| RB12:15` | Even | 0 | Simple Fixed | 1 | 2 |
| **33** | Or Word Immediate | **`ori rt,ra,value`**<br>`1. t ← RepLeftBit(I10,32)`<br>`2. RT0:3 ← RA0:3 \| t`<br>`3. RT4:7 ← RA4:7 \| t`<br>`4. RT8:11 ← RA8:11 \| t`<br>`5. RT12:15 ← RA12:15 \| t` | Even | 0 | Simple Fixed | 1 | 2 |
| **34** | Or Halfword Immediate | **`orhi rt,ra,value`**<br>`1. t ← RepLeftBit(I10,16)`<br>`2. RT0:1 ← RA0:1 \| t`<br>`3. RT2:3 ← RA2:3 \| t`<br>`4. RT4:5 ← RA4:5 \| t`<br>`5. RT6:7 ← RA6:7 \| t`<br>`6. RT8:9 ← RA8:9 \| t` | Even | 0 | Simple Fixed | 1 | 2 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | 7. RT10:11 ← RA10:11 \| t<br>8. RT12:13 ← RA12:13 \| t<br>9. RT14:15 ← RA14:15 \| t | | | | | |
| **35** | Or Byte Immediate | **orbi rt,ra,value**<br>1. b ← I10 & 0x00FF<br>2. bbbb ← b \|\| b \|\| b \|\| b<br>3. RT0:3 ← RA0:3 \| bbbb<br>4. RT4:7 ← RA4:7 \| bbbb<br>5. RT8:11 ← RA8:11 \| bbbb<br>6. RT12:15 ← RA12:15 \| bbbb | Even | 0 | Simple Fixed | 1 | 2 |
| **36** | Exclusive Or | **xor rt,ra,rb**<br>1. RT0:3 ← RA0:3 ⊕ RB0:3<br>2. RT4:7 ← RA4:7 ⊕ RB4:7<br>3. RT8:11 ← RA8:11 ⊕ RB8:11<br>4. RT12:15 ← RA12:15 ⊕ RB12:15 | Even | 0 | Simple Fixed | 1 | 2 |
| **37** | Exclusive Or Word Immediate | **xori rt,ra,value**<br>1. t ← RepLeftBit(I10,32)<br>2. RT0:3 ← RA0:3 ⊕ t<br>3. RT4:7 ← RA4:7 ⊕ t<br>4. RT8:11 ← RA8:11 ⊕ t<br>5. RT12:15 ← RA12:15 ⊕ t | Even | 0 | Simple Fixed | 1 | 2 |
| **38** | Exclusive Or Halfword Immediate | **xorhi rt,ra,value**<br>1. t ← RepLeftBit(I10,16)<br>2. RT0:1 ← RA0:1 ⊕ t<br>3. RT2:3 ← RA2:3 ⊕ t<br>4. RT4:5 ← RA4:5 ⊕ t<br>5. RT6:7 ← RA6:7 ⊕ t<br>6. RT8:9 ← RA8:9 ⊕ t<br>7. RT10:11 ← RA10:11 ⊕ t<br>8. RT12:13 ← RA12:13 ⊕ t<br>9. RT14:15 ← RA14:15 ⊕ t | Even | 0 | Simple Fixed | 1 | 2 |
| **39** | Exclusive Or Byte Immediate | **xorbi rt,ra,value**<br>1. b ← I10 & 0x00FF<br>2. bbbb ← b \|\| b \|\| b \|\| b<br>3. RT0:3 ← RA0:3 ⊕ bbbb<br>4. RT4:7 ← RA4:7 ⊕ bbbb<br>5. RT8:11 ← RA8:11 ⊕ bbbb<br>6. RT12:15 ← RA12:15 ⊕ bbbb | Even | 0 | Simple Fixed | 1 | 2 |
| **40** | Nand | **nand rt,ra,rb**<br>1. RT0:3 ← ¬(RA0:3 & RB0:3)<br>2. RT4:7 ← ¬(RA4:7 & RB4:7)<br>3. RT8:11 ← ¬(RA8:11 & RB8:11)<br>4. RT12:15 ← ¬(RA12:15 & RB12:15) | Even | 0 | Simple Fixed | 1 | 2 |
| **41** | Nor | **nor rt,ra,rb**<br>1. RT0:3 ← ¬(RA0:3 \| RB0:3)<br>2. RT4:7 ← ¬(RA4:7 \| RB4:7)<br>3. RT8:11 ← ¬(RA8:11 \| RB8:11)<br>4. RT12:15 ← ¬(RA12:15 \| RB12:15) | Even | 0 | Simple Fixed | 1 | 2 |
| **42** | Equivalent | **eqv rt,ra,rb**<br>1. RT0:3 ← RA0:3 ⊕ (¬RB0:3)<br>2. RT4:7 ← RA4:7 ⊕ (¬RB4:7)<br>3. RT8:11 ← RA8:11 ⊕ (¬RB8:11)<br>4. RT12:15 ← RA12:15 ⊕ (¬RB12:15) | Even | 0 | Simple Fixed | 1 | 2 |
| **Shift and Rotate Instructions** | | | | | | | |
| **43** | Shift left Halfword | **shlh rt,ra,rb**<br>1. for j = 0 to 15 by 2<br>      s ← RBj::2 & 0x001F<br>      t ← RAj::2<br>      for b = 0 to 15<br>               if b + s < 16 then<br>                      rb ← tb + s<br>               else rb ← 0<br>      end<br>      RTj::2 ← r<br>    end | Even | 0 | Shift | 2 | 4 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **44** | Shift left Halfword Immediate | **shlhi rt,ra,value**<br>1.s ← RepLeftBit(I7,16) & 0x001F<br>2.for j = 0 to 15 by 2<br>    t ← RAj::2<br>    for b = 0 to 15<br>      if b + s < 16 then rb ← t(b+s)<br>      else rb ← 0<br>    end<br>    RTj::2 ← r<br>  end | Even | 0 | Shift | 2 | 4 |
| **45** | Shift left Word | **shl rt,ra,rb**<br>1. for j = 0 to 15 by 4<br>       s ← RBj::4 & 0x0000003F<br>       t ← RAj::4<br>       for b = 0 to 31<br>           if b + s < 32 then<br>               rb ← tb + s<br>           else rb ← 0<br>       end<br>       RTj::4 ← r<br>   end | Even | 0 | Shift | 2 | 4 |
| **46** | Shift left Immediate | **shli rt,ra,value**<br>1. s ← RepLeftBit(I7,32) &<br>0x0000003F<br>2. for j = 0 to 15 by 4<br>       t ← RAj::4<br>       for b = 0 to 31<br>           if b + s < 32 then<br>                rb ← tb + s<br>           else rb ← 0<br>       end<br>       RTj::4 ← r<br>   end | Even | 0 | Shift | 2 | 4 |
| **47** | Shift left Quadword by Bits | **shlqbi rt,ra,rb**<br>1. s ← RB29:31<br>2. for b = 0 to 127<br>      if b + s < 128 then<br>            rb ← RAb + s<br>      else rb ← 0<br>    end<br>3. RT ← r | Odd | 1 | Permute | 5 | 4 |
| **48** | Shift left Quadword by Bits Immediate | **shlqbii rt,ra,value**<br>1. s ← I7 & 0x07<br>2. for b = 0 to 127<br>       if b + s < 128 then<br>             rb ← RAb + s<br>       else rb ← 0<br>   end<br>3. RT ← r | Odd | 1 | Permute | 5 | 4 |
| **49** | Shift left Quadword by Bytes | **shlqby rt,ra,rb**<br>1. s ← RB27:31<br>2. for b = 0 to 15<br>     if b + s < 16 then rb ← RAb + s<br>     else rb ← 0<br>   end<br>3. RT ← r | Odd | 1 | Permute | 5 | 4 |
| **50** | Shift left Quadword by Bytes Immediate | **shlqbyi rt,ra,value**<br>1. s ← I7 & 0x1F<br>2. for b = 0 to 15<br>     if b + s < 16 then rb ← RAb + s<br>     else rb ← 0<br>   end<br>3. RT ← r | Odd | 1 | Permute | 5 | 4 |
| **51** | Rotate Word | **rot rt,ra,rb**<br>1. for j = 0 to 15 by 4<br>       s ← RBj::4 & 0x0000001F | Even | 0 | Shift | 2 | 4 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | `t ← RAj::4` `for b = 0 to 31` `if b + s < 32 then` `rb ← tb + s` `else rb ← tb + s - 32` `end` `RTj::4 ← r` `end` | | | | | |
| **52** | Rotate Word Immediate | **`roti rt,ra,value`** `1. s ← RepLeftBit(I7,32) & 0x0000001F` `2. for j = 0 to 15 by 4` `t ← RAj::4` `for b = 0 to 31` `if b + s < 32 then` `rb ← tb + s` `else rb ← tb + s - 32` `end` `RTj::4 ← r` `end` | Even | 0 | Shift | 2 | 4 |
| **53** | Rotate Quadword by Bits | **`rotqbi rt,ra,rb`** `1. s ← RB29:31` `2. for b = 0 to 127` `if b + s < 128 then` `rb ← RAb + s` `else rb ← RAb + s - 128` `end` `3. RT ← r` | Odd | 1 | Permute | 5 | 4 |
| **54** | Rotate Quadword by Bits Immediate | **`rotqbii rt,ra,value`** `1. s ← I4:6` `2. for b = 0 to 127` `if b + s < 128 then` `rb ← RAb + s` `else rb ← RAb + s - 128` `end` `3. RT ← r` | Odd | 1 | Permute | 5 | 4 |
| **55** | Rotate Quadword by Bytes | **`rotqby rt,ra,rb`** `1. s ← RB28:31` `2. for b = 0 to 15` `if b + s < 16 then rb ← RAb + s` `else rb ← RAb + s - 16` `end` `3. RT ← r` | Odd | 1 | Permute | 5 | 4 |
| **56** | Rotate Quadword by Bytes Immediate | **`rotqbyi rt,ra,value`** `1. s ← I714:17` `2. for b = 0 to 15` `if b + s < 16 then` `rb ← RAb + s` `else rb ← RAb + s - 16` `end` `3. RT ← r` | Odd | 1 | Permute | 5 | 4 |
| **Compare, Branch, and Halt Instructions** | | | | | | | |
| **57** | Stop | **`stop`** `1. PC ← PC + 4` `2. Precise stop` | Even | 0 | Simple Fixed | 1 | 2 |
| **58** | Compare Equal Word | **`ceq rt,ra,rb`** `1. for i = 0 to 15 by 4` `If RAi::4 = RBi::4 then` `RTi::4 ← 0xFFFFFFFF` `else RTi::4 ← 0x00000000` `end` | Even | 0 | Simple Fixed | 1 | 2 |
| **59** | Compare Equal Word Immediate | **`ceqi rt,ra,value`** `1. for i = 0 to 15 by 4` `If RAi::4 = RepLeftBit(I10,32) then RTi::4 ← 0xFFFFFFFF` `else RTi::4 ← 0x00000000` | Even | 0 | Simple Fixed | 1 | 2 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | end | | | | | |
| **60** | Compare Equal Halfword | **ceqh rt,ra,rb**<br>1. for i = 0 to 15 by 2<br>    If RAi::2 = RBi::2 then<br>      RTi::2 ← 0xFFFF<br>    else RTi::2 ← 0x0000<br>  end | Even | 0 | Simple Fixed | 1 | 2 |
| **61** | Compare Equal Halfword Immediate | **ceqhi rt,ra,value**<br>1. for i = 0 to 15 by 2<br>    If RAi::2 = RepLeftBit(I10,16)<br>      then RTi::2 ← 0xFFFF<br>    else RTi::2 ← 0x0000<br>  end | Even | 0 | Simple Fixed | 1 | 2 |
| **62** | Compare Greater Than Word | **cgt rt,ra,rb**<br>1. for i = 0 to 15 by 4<br>    If RAi::4 > RBi::4 then<br>      RTi::4 ← 0xFFFFFFFF<br>    else RTi::4 ← 0x00000000<br>  end | Even | 0 | Simple Fixed | 1 | 2 |
| **63** | Compare Greater Than Word Immediate | **cgti rt,ra,value**<br>1. for i = 0 to 15 by 4<br>    If RAi::4 >RepLeftBit(I10,32)<br>      then RTi::4 ← 0xFFFFFFFF<br>    else RTi::4 ← 0x00000000<br>  end | Even | 0 | Simple Fixed | 1 | 2 |
| **64** | Compare Greater Than Halfword | **cgth rt,ra,rb**<br>1. for i = 0 to 15 by 2<br>     If RAi::2 > RBi::2<br>       then RTi::2 ← 0xFFFF<br>     else RTi::2 ← 0x0000<br>  end | Even | 0 | Simple Fixed | 1 | 2 |
| **65** | Compare Greater Than Halfword Immediate | **cgthi rt,ra,value**<br>1. for i = 0 to 15 by 2<br>     If RAi::2 >RepLeftBit(I10,16)<br>then RTi::2 ← 0xFFFF<br>     else RTi::2 ← 0x0000<br>  end | Even | 0 | Simple Fixed | 1 | 2 |
| **66** | Branch Relative and Set Link | **brsl rt,symbol**<br>1. RT0:3 ← (PC + 4) & LSLR<br>2. RT4:15 ← 0<br>3. PC ← (PC + RepLeftBit(I16 \|\| 0b00,32)) & LSLR | Odd | 1 | Branch | 7 | 4 |
| **67** | Branch Absolute and Set Link | **brasl rt,symbol**<br>1. RT0:3 ← (PC + 4) & LSLR<br>2. RT4:15 ← 0<br>PC ← RepLeftBit(I16 \|\| 0b00,32) & LSLR | Odd | 1 | Branch | 7 | 4 |
| **68** | Branch Indirect | **bi ra**<br>1. PC ← RA0:3 & LSLR & 0xFFFFFFFC | Odd | 1 | Branch | 7 | 4 |
| **69** | Branch Relative | **br symbol**<br>1. PC ← (PC + RepLeftBit(I16 \|\| 0b00,32)) & LSLR | Odd | 1 | Branch | 7 | 4 |
| **70** | Branch Absolute | **bra symbol**<br>1. PC ← RepLeftBit(I16 \|\| 0b00,32) & LSLR | Odd | 1 | Branch | 7 | 4 |
| **71** | Branch if Not Zero Word | **brnz rt,symbol**<br>1. If RT0:3 ≠ 0 then<br>     PC ← (PC + RepLeftBit(I16 \|\| 0b00))& LSLR & 0xFFFFFFFC<br>  else<br>     PC ← (PC+4) & LSLR | Odd | 1 | Branch | 7 | 4 |
| **72** | Branch if Zero Word | **brz rt,symbol**<br>1. If RT0:3 = 0 then<br>    PC ← (PC + RepLeftBit(I16 \|\| 0b00))& LSLR & 0xFFFFFFFC<br>  else | Odd | 1 | Branch | 7 | 4 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | PC ← (PC + 4) & LSLR | | | | | |
| **73** | Branch if Not Zero Halfword | **`brhnz rt,symbol`**<br>1. If RT2:3 ≠ 0 then<br>PC ← (PC + RepLeftBit(I16 \|\| 0b00)) & LSLR & 0xFFFFFFFC<br>else<br>PC ← (PC + 4) & LSLR | Odd | 1 | Branch | 7 | 4 |
| **74** | Branch if Zero Halfword | **`brhz rt,symbol`**<br>1. If RT2:3 = 0 then<br>PC ← (PC + RepLeftBit(I16 \|\| 0b00)) & LSLR & 0xFFFFFFFC<br>else<br>PC ← (PC + 4) & LSLR | Odd | 1 | Branch | 7 | 4 |
| **Floating-Point Instructions** | | | | | | | |
| **75** | Floating Add | **`fa rt,ra,rb`** | Even | 0 | Single Precision | 3 | 6 |
| **76** | Floating Subtract | **`fs rt,ra,rb`** | Even | 0 | Single Precision | 3 | 6 |
| **77** | Floating Multiply | **`fm rt,ra,rb`** | Even | 0 | Single Precision | 3 | 6 |
| **78** | Floating Multiply and Add | **`fma rt,ra,rb,rc`** | Even | 0 | Single Precision | 3 | 6 |
| **79** | Floating Multiply and Subtract | **`fms rt,ra,rb,rc`** | Even | 0 | Single Precision | 3 | 6 |
| **80** | Floating Compare Equal | **`fceq rt,ra,rb`** | Even | 0 | Single Precision | 3 | 6 |
| **81** | Floating Compare Magnitude Equal | **`fcmeq rt,ra,rb`** | Even | 0 | Single Precision | 3 | 6 |
| **82** | Floating Compare Greater Than | **`fcgt rt,ra,rb`** | Even | 0 | Single Precision | 3 | 6 |
| **83** | Floating Compare Magnitude Greater Than | **`fcmgt rt,ra,rb`** | Even | 0 | Single Precision | 3 | 6 |
| **Control Instructions** | | | | | | | |
| **84** | No Operation (Load) | **`lnop`** | Even | 0 | - | - | - |
| **85** | No Operation (Execute) | **`nop`** | Odd | 1 | - | - | - |
| **86** | Cache Miss | **`cmiss`** | Odd | 1 | - | - | - |

*Table 4 Instruction Format Mapping*

| **Format** | **Index** |
|---|---|
| RR | 1 |
| RRR | 2 |
| RI7 | 3 |
| RI10 | 4 |
| RI16 | 5 |
| RI18 | 6 |

*Table 5 Instruction Opcodes and Format*

| No. | Name | Opcode | Instruction Format No. |
|---|---|---|---|
| **SIMPLE FIXED** | | | |
| **1** | Immediate Load Halfword | 11’bXX010000011 | 5 |
| **2** | Immediate Load Word | 11’bXX010000001 | 5 |
| **3** | Immediate Load Address | 11’bXXXX0100001 | 6 |
| **4** | Add Half Word | 11’b00011001000 | 1 |
| **5** | Add Half Word Immediate | 11’bXXX00011101 | 4 |
| **6** | Add Word | 11’b00011000000 | 1 |
| **7** | Add Word Immediate | 11’bXXX00011100 | 4 |
| **8** | Subtract from Halfword | 11’b00001001000 | 1 |
| **9** | Subtract from Halfword Immediate | 11’bXXX00001101 | 4 |
| **10** | Subtract from Word | 11’b00001000000 | 1 |
| **11** | Subtract from Word Immediate | 11’bXXX00001100 | 4 |
| **12** | Add Extended | 11’b01101000000 | 1 |
| **13** | Subtract from Extended | 11’b01101000001 | 1 |
| **14** | Carry Generate | 11’b00011000010 | 1 |
| **15** | Borrow Generate | 11’b00001000010 | 1 |
| **16** | Count Leading Zeros | 11’b01010100101 | 1 |
| **17** | And | 11’b00011000001 | 1 |
| **18** | And Word Immediate | 11’bXXX00010100 | 4 |
| **19** | And Halfword Immediate | 11’bXXX00010101 | 4 |
| **20** | And Byte Immediate | 11’bXXX00010110 | 4 |
| **21** | Or | 11’b00001000001 | 1 |
| **22** | Or Word Immediate | 11’bXXX00000100 | 4 |
| **23** | Or Halfword Immediate | 11’bXXX00000101 | 4 |
| **24** | Or Byte Immediate | 11’bXXX00000110 | 4 |
| **25** | Exclusive Or | 11’b01001000001 | 1 |
| **26** | Exclusive Or Word Immediate | 11’bXXX01000100 | 4 |
| **27** | Exclusive Or Halfword Immediate | 11’bXXX01000101 | 4 |
| **28** | Exclusive Or Byte Immediate | 11’bXXX01000110 | 4 |
| **29** | Nand | 11’b00011001001 | 1 |
| **30** | Nor | 11’b00001001001 | 1 |
| **31** | Equivalent | 11’b01001001001 | 1 |
| **32** | Compare Equal Word | 11’b01111000000 | 1 |
| **33** | Compare Equal Word Immediate | 11’bXXX01111100 | 4 |
| **34** | Compare Equal Halfword | 11’b01111001000 | 1 |
| **35** | Compare Equal Halfword Immediate | 11’bXXX01111110 | 4 |
| **36** | Compare Greater Than Word | 11’b01001000000 | 1 |
| **37** | Compare Greater Than Word Immediate | 11’bXXX01001100 | 4 |
| **38** | Compare Greater Than Halfword | 11’b01001001000 | 1 |

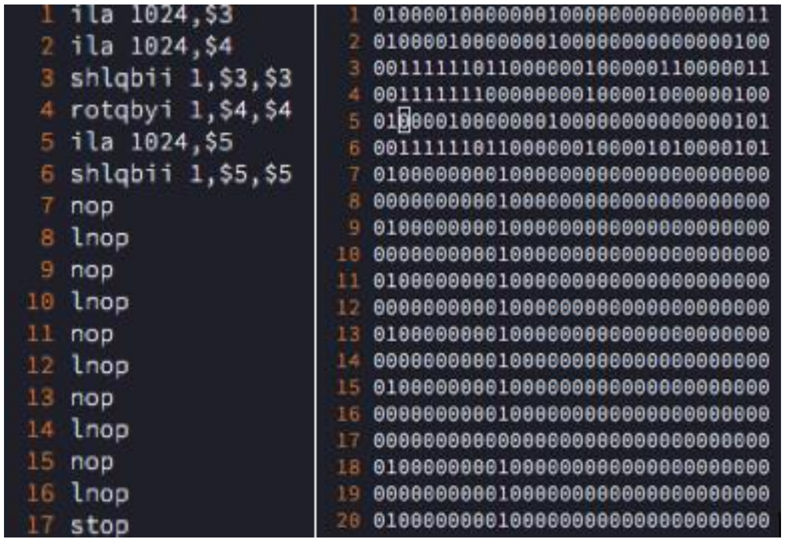
| | | | |
|---|---|---|---|
| **39** | Compare Greater Than Halfword Immediate | 11’bXXX01001101 | 4 |
| **Shift and Rotate** | | | |
| **40** | Shift left Halfword | 11’b00001011111 | 1 |
| **41** | Shift left Halfword Immediate | 11’b00001111111 | 3 |
| **42** | Shift left Word | 11’b00001011011 | 1 |
| **43** | Shift left Immediate | 11’b00001111011 | 3 |
| **44** | Shift left Quadword by Bits | 11’b00111011011 | 1 |
| **45** | Shift left Quadword by Bits Immediate | 11’b00111111011 | 3 |
| **46** | Shift left Quadword by Bytes | 11’b00111011111 | 1 |
| **47** | Shift left Quadword by Bytes Immediate | 11’b00111111111 | 3 |
| **48** | Rotate Word | 11’b00001011000 | 1 |
| **49** | Rotate Word Immediate | 11’b00001111100 | 3 |
| **50** | Rotate Quadword by Bits | 11’b00111011000 | 1 |
| **51** | Rotate Quadword by Bits Immediate | 11’b00111111000 | 3 |
| **52** | Rotate Quadword by Bytes | 11’b00111011100 | 1 |
| **53** | Rotate Quadword by Bytes Immediate | 11’b00111111100 | 3 |
| **Memory Load-Store Instructions** | | | |
| **54** | Load Quadword Dform | 11’bXXX00110100 | 4 |
| **55** | Load Quadword Aform | 11’bXX001100001 | 5 |
| **56** | Store Quadword Dform | 11’bXXX00100100 | 4 |
| **57** | Store Quadword Aform | 11’bXX001000001 | 5 |
| **Branch Halt Instructions** | | | |
| **58** | Branch Relative and Set Link | 11’bXX001100110 | 5 |
| **59** | Branch Absolute and Set link | 11’bXX001100010 | 5 |
| **60** | Branch Indirect | 11’b00110101000 | 1 |
| **61** | Branch Relative | 11’bXX001100100 | 5 |
| **62** | Branch Absolute | 11’bXX001100000 | 5 |
| **63** | Branch If not Zero Word | 11’bXX001000010 | 5 |
| **64** | Branch If Zero Word | 11’bXX001000000 | 5 |
| **65** | Branch If not Zero HalfWord | 11’bXX001000110 | 5 |
| **66** | Branch If Zero Halfword | 11’bXX001000100 | 5 |
| **Floating point operations** | | | |
| **67** | Floating Add | 11’b01011000100 | 1 |
| **68** | Floating Subtract | 11’b01011000101 | 1 |
| **69** | Floating Multiply | 11’b01011000110 | 1 |
| **70** | Floating Multiply and Add | 11’bXXXXXXX1110 | 2 |
| **71** | Floating Multiply and Subtract | 11’bXXXXXXX1111 | 2 |
| **72** | Floating Compare Equal | 11’b01111000010 | 1 |
| **73** | Floating Compare Magnitude Equal | 11’b01111001010 | 1 |
| **74** | Floating Compare greater than | 11’b01011000010 | 1 |
| **75** | Floating Compare Magnitude greater than | 11’b01011001010 | 1 |
| **Other miscellaneous Instructions** | | | |

| | | | |
|---|---|---|---|
| **76** | Multiply | 11’b01111000100 | 1 |
| **77** | Multiply and add | 11’bXXXXXXX1100 | 2 |
| **78** | Multiply immediate | 11’bXXX01110100 | 4 |
| **79** | Stop | 11’b00000000000 | |
| **80** | Count Ones in Bytes | 11’b01010110100 | 1 |
| **81** | Average Bytes | 11’b00011010011 | 1 |
| **82** | Absolute difference of bytes | 11’b00001010011 | 1 |
| **83** | Sum bytes into Halfword | 11’b01001010011 | 1 |
| **84** | No Operation (Load) | 11’b00000000001 | |
| **85** | No Operation (Execute) | 11’b01000000001 | |
| **86** | Cache Miss | 11’b11111111111 | |

# 2 Decoded Instruction Format

## 2.1 SPU Assembly Format

To the test our SPU mini implementation with different programs, we created a simple assembly format. In this format the instructions are abbreviated using mnemonics and registers with ‘$’ sign before them (except for register ‘0’ and ‘1’). Register ‘0’ is represented with special symbol ‘**$lr**’ and is always used as link register. Register ‘1’ is represented with special symbol ‘**$sp**’ and is always used as stack pointer. *Figure* 2-1 shows an example of SPU mini assembly program.



```
ila 1024,$3
ila 1024,$4
shlqbii 1,$3,$3
rotqbyi 1,$4,$4
ila 1024,$5
shlqbii 1,$5,$5
nop
lnop
nop
lnop
nop
lnop
nop
lnop
nop
lnop
stop
```

```
01000010000000100000000000000011
01000010000000100000000000000100
00111111011000000100000110000011
00111111100000000100001000000100
01000010000000100000000000000101
00111111011000000100001010000101
01000000001000000000000000000000
00000000001000000000000000000000
01000000001000000000000000000000
00000000001000000000000000000000
01000000001000000000000000000000
00000000001000000000000000000000
01000000001000000000000000000000
00000000001000000000000000000000
01000000001000000000000000000000
00000000001000000000000000000000
00000000000000000000000000000000
01000000001000000000000000000000
00000000001000000000000000000000
01000000001000000000000000000000
```

*Figure 2-1 Example SPU mini assembly program with disassembly*

The parser code also generates the memory load file (**ls_load_file.txt**) for used by the testbench. The parser code to decode the assembly into machine code is written in python and listed in *Appendix A – Assembly Parser Code (sasm.py)*.

## 2.2 SPU Pipes Assembly Format

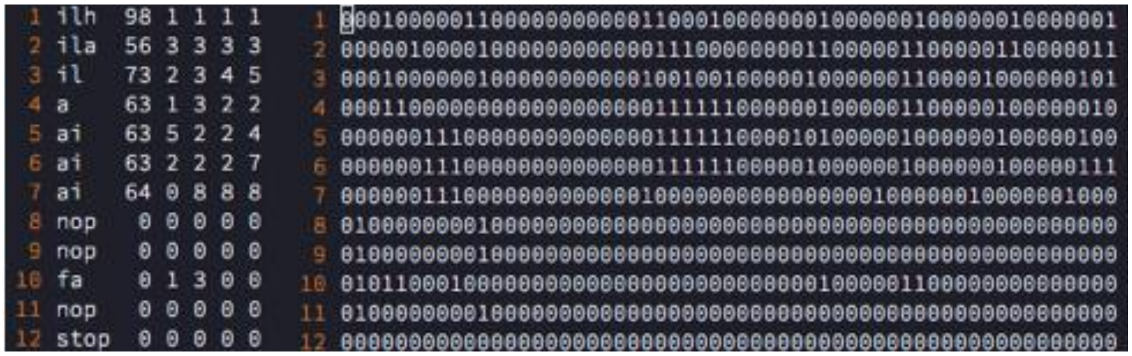
This format was used when we were testing only the implementation of pipes. The explanation of the format is given in below table, with some example instructions.

*Table 6 SPU Pipes Assembly Format*

| Instruction Mnemonic | Immediate Value | RA address | RB address | RC address | RT address |
|---|---|---|---|---|---|
| **ilh** | 98 | X | X | X | 1 |
| **ai** | 63 | 2 | X | X | 3 |
| **nop** | X | X | X | X | X |

Instruction mnemonic is followed by Immediate value, which is interpreted as I8, I10, I16 or I18 depending on the instruction. Following that we have the addresses of four registers RA, RB, RC and RT. Only some of these values will be used depending on the instruction (rest being don't care).

We created instruction file for each of the pipes. The files are converted to their bit representation by a python utility we wrote. The bit representations are then read by the testbench and fed to the relevant pipes. Below is the example of instructions file for the even pipe:



```
ilh   98 1 1 1 1
ila   56 3 3 3 3
il    73 2 3 4 5
a     63 1 3 2 2
ai    63 5 2 2 4
ai    63 2 2 2 7
ai    64 0 8 8 8
nop    0 0 0 0 0
nop    0 0 0 0 0
fa     0 1 3 0 0
nop    0 0 0 0 0
stop   0 0 0 0 0
```

```
000100000110000000000011000100000001000000100000010000001
000001000010000000000001110000000011000001100000110000011
000100000010000000000010010010000010000001100001000000101
000110000000000000000001111110000001000001100000100000010
000000111000000000000001111110000101000001000000100000100
000000111000000000000001111110000010000001000000100000111
000000111000000000000010000000000000000100000010000001000
010000000010000000000000000000000000000000000000000000000
010000000010000000000000000000000000000000000000000000000
010110001000000000000000000000000001000001100000000000000
010000000010000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000
```

*Figure 2-2 SPU Pipes Instruction Format and Machine Translation*

The use of the above intermediate decoded format greatly helped the ease of debugging of SPU instructions.

# 3 Block Descriptions

## 3.1 Local Store (local_store.sv)

A block of memory which is 32kB in length and 8 bits wide is provided to the SPU as the local storage. The instructions and the associated data which are loaded in the local store during the start of the simulation (from the file "ls_load_file.txt", generated by the parser utility). The local store communicates with SPU 16 bytes per clock cycle through the ports **ls_data_wr** and

**ls_data_rd**. It also provides interface to instruction cache using **cache_out** and **cache_wr** signals. *Figure* 3-1 shows example local store operation with cache miss and store operation.

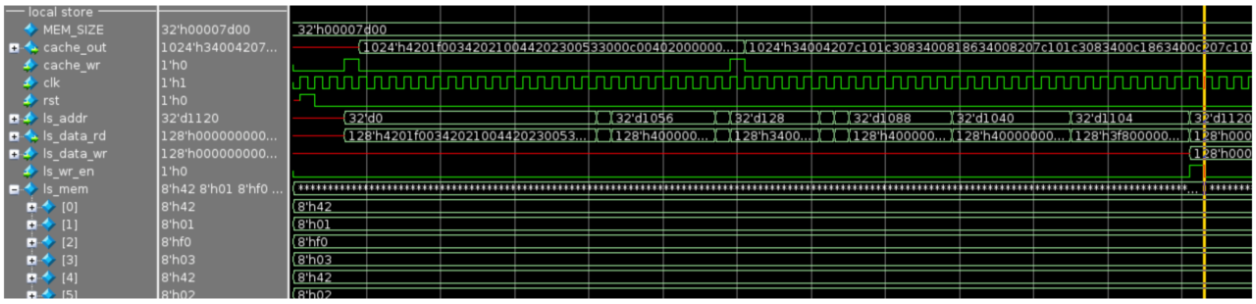


*Figure 3-1 Example Local Store Operation*

The complete listing is given in *Appendix B – Local Store (local_*store.sv)

## 3.2 Fetch (fetch.sv)

The instructions are brought into the execution pipelines using the fetch module. A 2-way fully associative cache memory is implemented to provide continuous instructions for the SPU to execute. The data is written into the cache by port **cache_out** when the control signal **cache_wr** goes high. The cache is of 32*64 bit in length. If the next instruction to be executed is not found in the cache memory, the **cmiss** instruction signal is generated. This indicates to the local store to source a new block of memory into the cache.

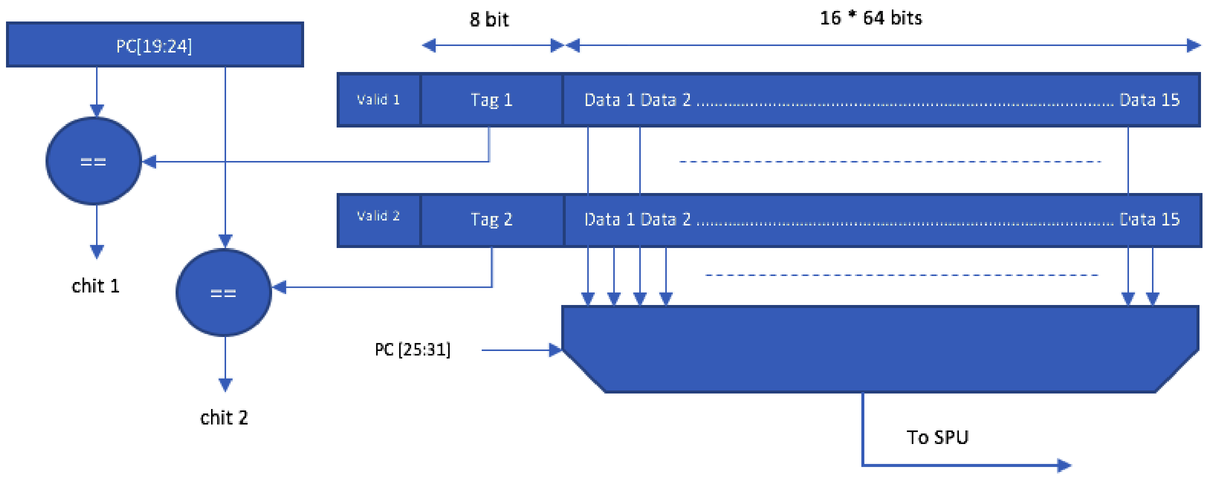

8 bit
16 * 64 bits
PC[19:24]
Valid 1
Tag 1
Data 1 Data 2 .................... Data 15
==
Valid 2
Tag 2
Data 1 Data 2 .................... Data 15
chit 1
==
PC [25:31]
chit 2
To SPU

*Figure 3-2 Cache Implementation*

The data is searched through the 2 blocks using **blk_tag** and its validity is checked using the **blk_valid** bit. When the data is available in the cache it asserts the cache hit signal (chit). If the data is not present on the cache memory the cache miss signal is asserted. The cache miss signal leads insertion of **cmiss** instruction to **odd_pipe**, which eventually loads the cache with appropriate data. If there is no free block in the cache, the least recently used block of memory is replaced.

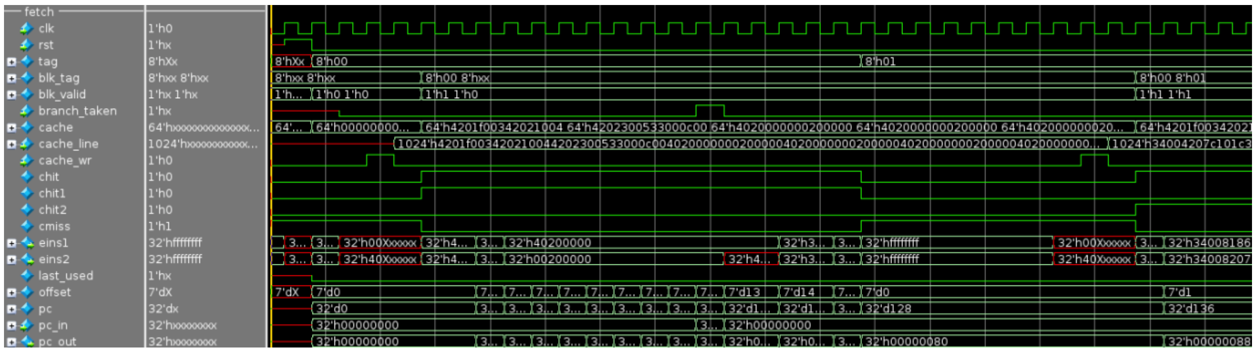


*Figure 3-3 Operation of Fetch on cache miss*

Fetch module is also responsible for saving the state of executing during a stall. When a stall is initiated by the dependency or the decode stage, no new instructions are fetched from the cache memory. The PC of the current instruction under execution is not advanced but instead is fed back until the **stall_done** is given out.

## 3.3 Decode (decode.sv)

The decode stage is responsible for the decoding of the instructions that are brought in by the fetch module. Decode module slices the bit array to obtain the opcode. Depending on the opcode type, the rest of the instruction and the operands are decoded. The Opcode of the instruction also determines the pipeline in which the instruction needs to be executed in.

When two instructions are fed to the SPU pipelines, the decode stage detects structural hazard in the execution of the program. The structural hazard is taken care of by stalling the execution of instructions until the necessary resources are vacated. Upon the detection of a structural hazard a decode stall '**dec_stall'** signal is asserted and until the '**stall_done'** signal goes high no new instructions are brought in from fetch module.

A sample code to demonstrate the decoding stall caused due to structural hazard is illustrated in the code below

```
ila 1024,$3
ila 1024,$4
shlqbii 1,$3,$3
rotqbyi 1,$4,$4
ila 1024,$5
shlqbii 1,$5,$5
nop
lnop
```

When 2 load immediate commands (ila) are brought in by the fetch module, both try to access the resource of the even pipe in the decoding stage causing a structural hazard and thereby a stall (dec_stall).  The decode stage on lets only the first instructions go and stalls until the hazard is no longer there to issue the remaining instruction.

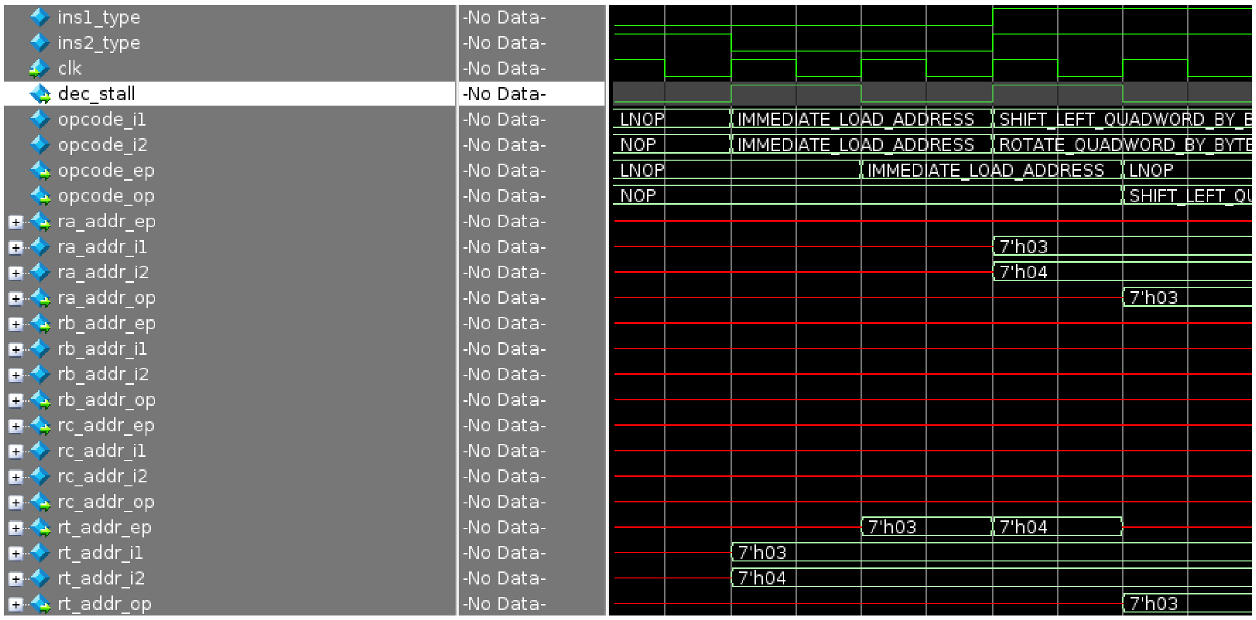


*Figure 3-4 Structural Hazard Resolution in Decode Stage*

## 3.4 Dependency (dependency.sv)

The dependency module is responsible for checking the data dependency in the instructions coming from the decode module. This is done by comparing the operands of an instruction which need to be sent to execution pipeline to the operands of the instructions that are being executed in the pipeline. When a dependency is detected the system will issue a stall signal '**dep_stall**' and that halts the fetching of any new instructions. A sample program to demonstrate dependency is given below.

```
ila 7,$2
ila 2,$3
ila 2,$4
ila 0,$5
lqd 0,$5,$6
a $4,$6,$7
mpya $6,$2,$3,$4
stqd 64,$5,$6
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
stop
```

The 'addition' command requires the value for the register 6 and it is loaded by the previous command (lqd) from the local store memory. This causes a RAW data hazard and a dependency stall is necessary to get the correct output.

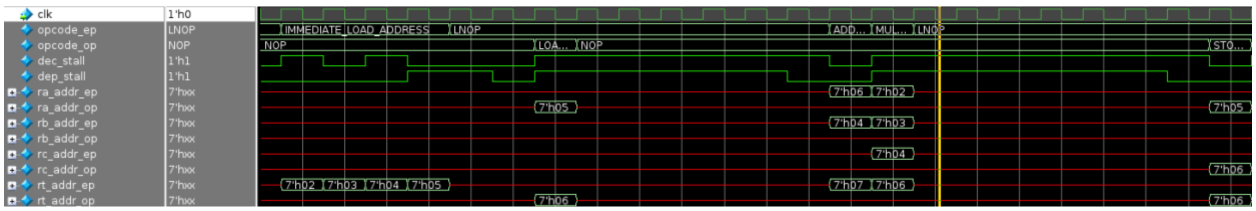


Figure 3-5 RAW Hazard Resolution in Dependency Module

During the execution of the 'lqd' command the **dep_stall** goes high by detecting a dependency and stays high until its completion. Once the command has successfully executed the **dep_stall** goes low and a **stall_done** signal is given out. This indicates the system to proceed with the next command which is the addition.

## 3.5 SPU Pipes Top (spu_pipes_top.sv)

SPU has a dual pipeline model which allows it to evaluate 2 different instructions simultaneously. The execution units are organized into 2 pipelines, even and odd. SPU_PIPES_TOP is a wrapper which instantiates register file, forwarding macro, even pipeline and odd pipeline. The only function of this file is to organize and connect its sub-modules.

*Note: The definitions package (defines_pkg.sv file) contains the binary equivalent opcodes for all the instructions and is imported in the other modules. It also defines the standard parameters such as length of preferred slots (Byte, Halfwords etc..) and files from where the input instructions are read.*

## 3.6 SPU Top (spu_top.sv)

SPU top is the top most wrapper which instantiates fetch, decode, local_store, dependency and spu_pipes_top modules. The only function of this file is to organize and connect its sub-modules.

## 3.7 Even Pipe (even_pipe.sv)

Even pipe contains the below executional units, along with the instruction latency. The execution of each instruction type is explained in the following section.

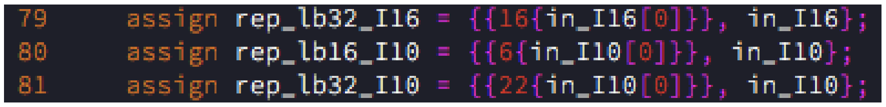
| Executional Unit | Instruction type | Latency |
|---|---|---|
| Simple Fixed | Constant-Formation Instructions | 2 |
| | Integer and Logical Instructions | 2 |
| | Compare Instructions | 2 |
| Shift | Shift and Rotate Instructions | 4 |
| Single precision | Floating-Point Instructions | 6 |

The latency column denotes the number of cycles before the result is available in the forwarding macro.

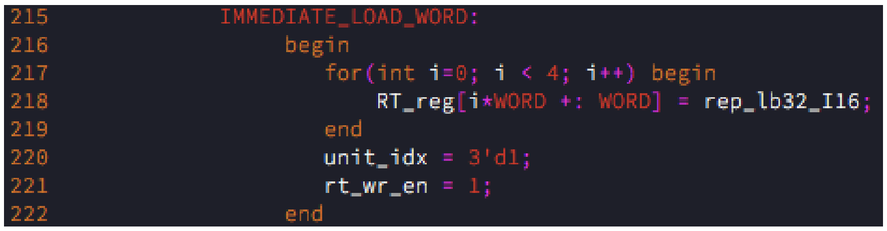
### 3.7.1 Constant Formation Instructions

These instructions are used to load an immediate value or an address to one of the general-purpose registers. These commands were implemented in the even pipe and were given a delay of 2 clock cycles before appearing in forwarding macro as they would behave in SPU.

The snippets of implementation and the simulation results of the command Immediate Load Word are shown below.



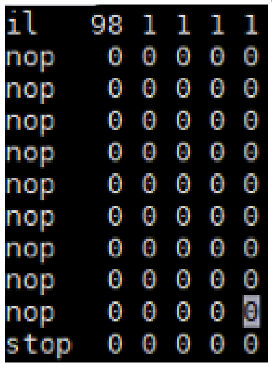
```
assign rep_lb32_I16 = {{16{in_I16[0]}}, in_I16};
assign rep_lb16_I10 = {{6{in_I10[0]}}, in_I10};
assign rep_lb32_I10 = {{22{in_I10[0]}}, in_I10};
```



```
            IMMEDIATE_LOAD_WORD:
                begin
                   for(int i=0; i < 4; i++) begin
                       RT_reg[i*WORD +: WORD] = rep_lb32_I16;
                   end
                   unit_idx = 3'd1;
                   rt_wr_en = 1;
                end
```

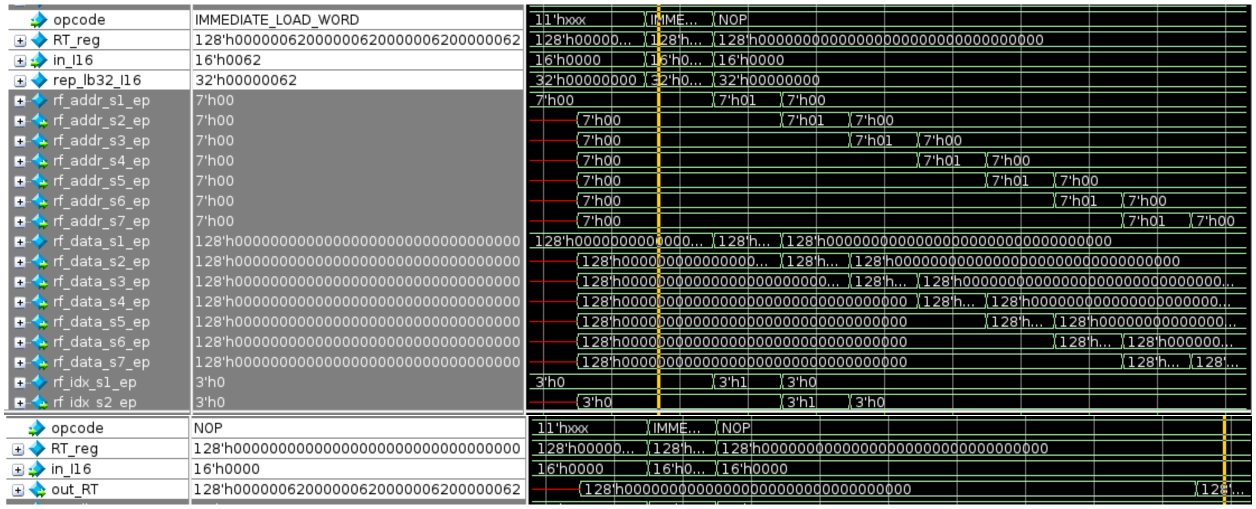
This command is used to load a word into the register RT, which is defined as **RT_reg** in the above code. The parameter **rep_lb32_I16** is assigned to extend sign bits of I16 value to 32 bits. The **unit_idx** variable determines the latency of appearance of output in forwarding macro.

Instructions to even pipe:



```
il    98 1 1 1 1
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
stop   0 0 0 0 0
```

Simulation results:
The value of opcode, **Rt_reg** and **rep_lb32_I16** are seen to be loaded correctly. The value propagates for 7 cycles to reach the output register **out_RT** and is written into it which is written to the register file.



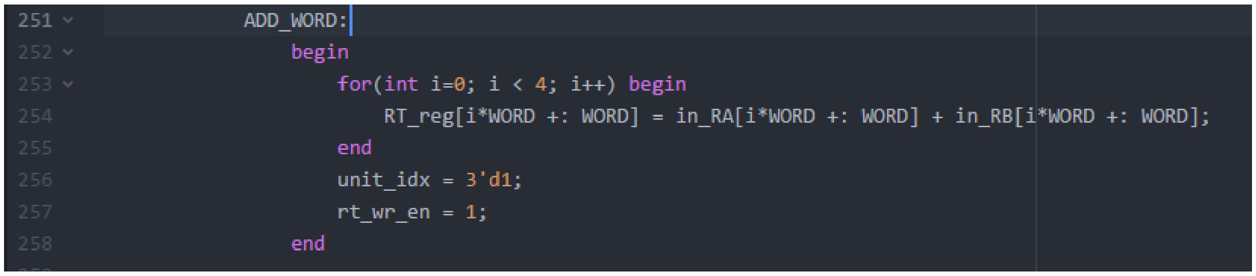
```
opcode          IMMEDIATE_LOAD_WORD
RT_reg          128'h00000062000000620000006200000062
in_I16          16'h0062
rep_lb32_I16    32'h00000062
rf_addr_s1_ep   7'h00
rf_addr_s2_ep   7'h00
rf_addr_s3_ep   7'h00
rf_addr_s4_ep   7'h00
rf_addr_s5_ep   7'h00
rf_addr_s6_ep   7'h00
rf_addr_s7_ep   7'h00
rf_data_s1_ep   128'h00000000000000000000000000000000
rf_data_s2_ep   128'h00000000000000000000000000000000
rf_data_s3_ep   128'h00000000000000000000000000000000
rf_data_s4_ep   128'h00000000000000000000000000000000
rf_data_s5_ep   128'h00000000000000000000000000000000
rf_data_s6_ep   128'h00000000000000000000000000000000
rf_data_s7_ep   128'h00000000000000000000000000000000
rf_idx_s1_ep    3'h0
rf_idx_s2_ep    3'h0

opcode          NOP
RT_reg          128'h00000000000000000000000000000000
in_I16          16'h0000
out_RT          128'h00000062000000620000006200000062
```

### 3.7.2 Integer and Logical Instructions

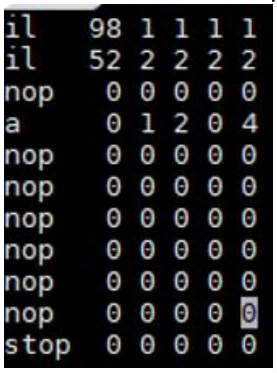
The operations involving arithmetic and logical manipulation of the inputs are performed with this executional unit. This also has a latency of 2 clock cycles before the result is made available.

The implementation of the instruction “**add word**” is demonstrated below along with its simulation result.



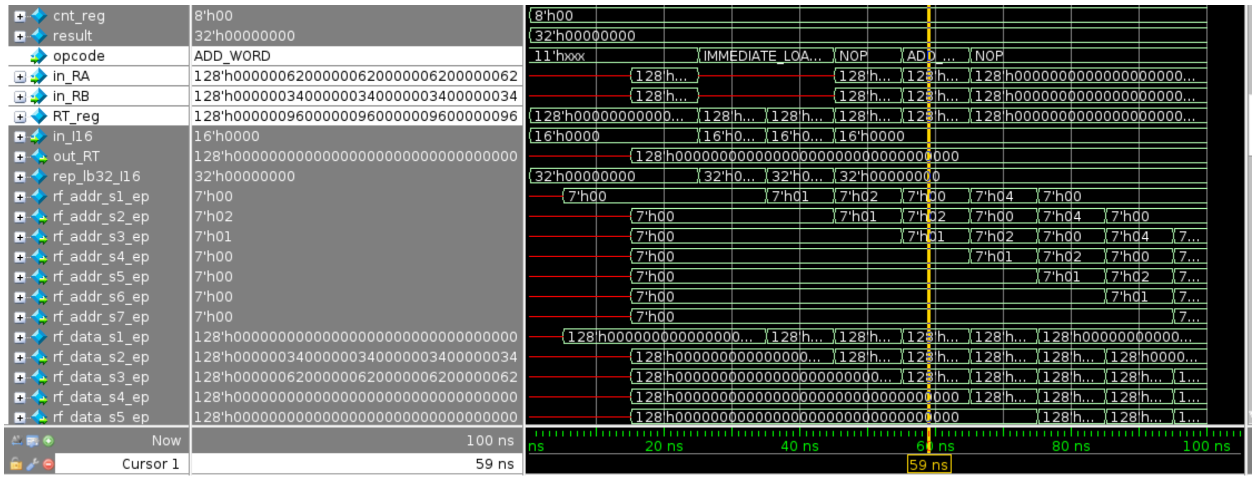
```
            ADD_WORD:
                begin
                    for(int i=0; i < 4; i++) begin
                        RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] + in_RB[i*WORD +: WORD];
                    end
                    unit_idx = 3'd1;
                    rt_wr_en = 1;
                end
```

Instructions to even pipe:



```
il    98 1 1 1 1
il    52 2 2 2 2
nop    0 0 0 0 0
a      0 1 2 0 4
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
stop   0 0 0 0 0
```

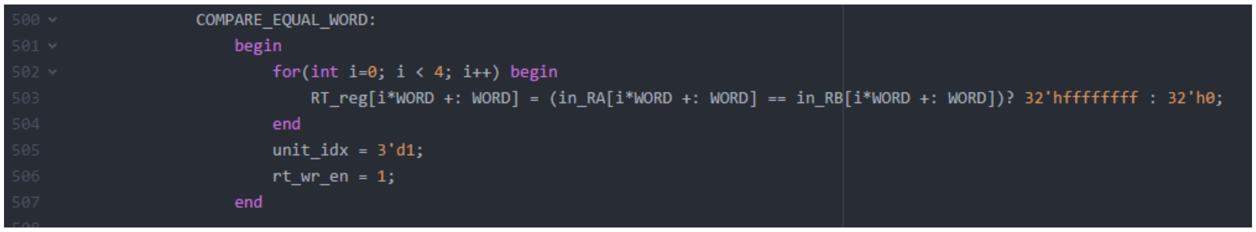
Simulation results:
The result of the addition is written to **Rt_reg** and propagates for 7 more cycles before being written to **out_Rt**. (The values shown below in the simulation are in Hex format).



```
cnt_reg          8'h00
result           32'h00000000
opcode           ADD_WORD
in_RA            128'h00000062000000620000006200000062
in_RB            128'h00000034000000340000003400000034
RT_reg           128'h00000096000000960000009600000096
in_I16           16'h0000
out_RT           128'h00000000000000000000000000000000
rep_lb32_I16     32'h00000000
rf_addr_s1_ep    7'h00
rf_addr_s2_ep    7'h02
rf_addr_s3_ep    7'h01
rf_addr_s4_ep    7'h00
rf_addr_s5_ep    7'h00
rf_addr_s6_ep    7'h00
rf_addr_s7_ep    7'h00
rf_data_s1_ep    128'h00000000000000000000000000000000
rf_data_s2_ep    128'h00000034000000340000003400000034
rf_data_s3_ep    128'h00000062000000620000006200000062
rf_data_s4_ep    128'h00000000000000000000000000000000
rf_data_s5_ep    128'h00000000000000000000000000000000
Now              100 ns
Cursor 1         59 ns
```

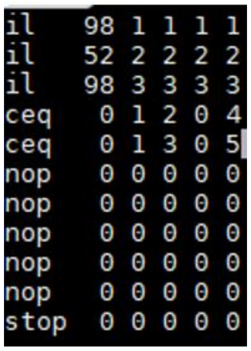
## 3.7.3 Compare Instructions

When the 2 inputs need to be checked for their equality, the compare instructions are used. The output register value is used as a flag whose value will be set to 1 if they are equal, 0 otherwise._The implementation of "**Compare_Equal_Word**" which compares each word of register **Ra** with that of **Rb** and stores the result in **Rt** is presented below.



```
                COMPARE_EQUAL_WORD:
                    begin
                        for(int i=0; i < 4; i++) begin
                            RT_reg[i*WORD +: WORD] = (in_RA[i*WORD +: WORD] == in_RB[i*WORD +: WORD])? 32'hffffffff : 32'h0;
                        end
                        unit_idx = 3'd1;
                        rt_wr_en = 1;
                    end
```

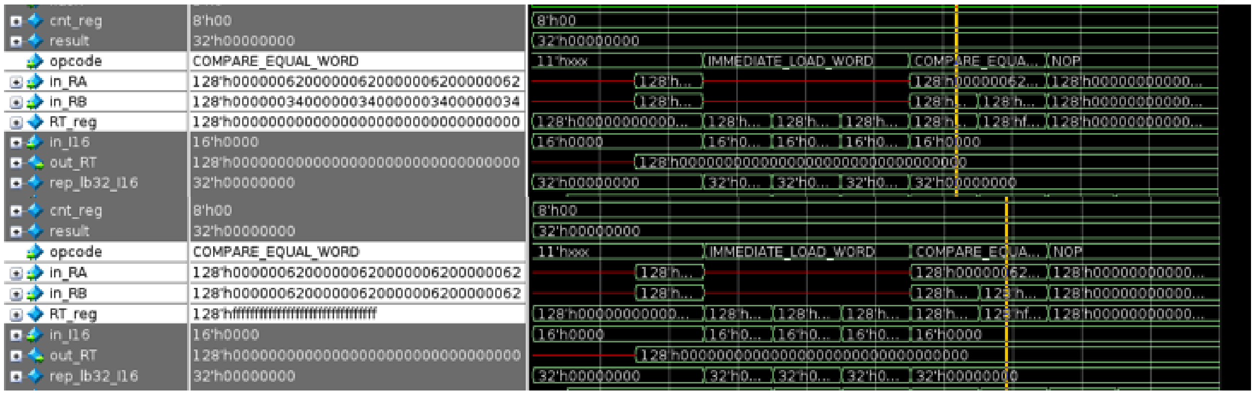
Instructions:



```
il    98 1 1 1 1
il    52 2 2 2 2
il    98 3 3 3 3
ceq    0 1 2 0 4
ceq    0 1 3 0 5
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
stop   0 0 0 0 0
```

Simulation result:

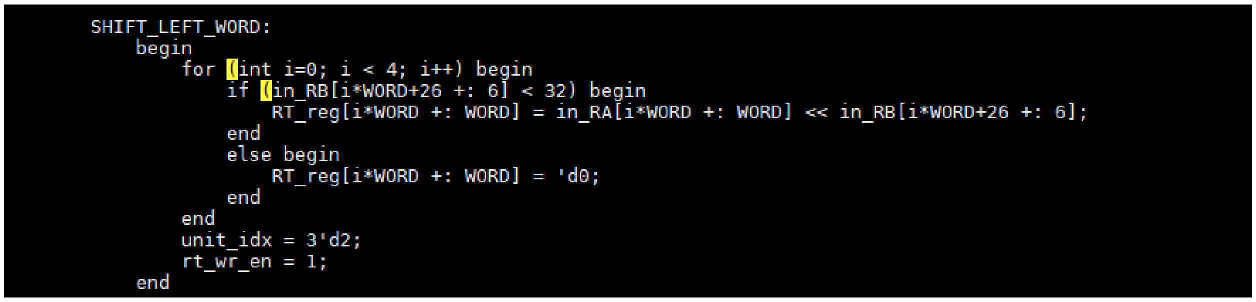
As shown in the waveform, the values in registers **1 and 2** are compared giving a result of 0 in **Rt_reg**. Later registers **1 and 3** are compared setting all the bits of **Rt_reg** to 1.



```
cnt_reg       8'h00
result        32'h00000000
opcode        COMPARE_EQUAL_WORD
in_RA         128'h00000062000000620000006200000062
in_RB         128'h00000034000000340000003400000034
RT_reg        128'h00000000000000000000000000000000
in_I16        16'h0000
out_RT        128'h00000000000000000000000000000000
rep_lb32_I16  32'h00000000
cnt_reg       8'h00
result        32'h00000000
opcode        COMPARE_EQUAL_WORD
in_RA         128'h00000062000000620000006200000062
in_RB         128'h00000062000000620000006200000062
RT_reg        128'hffffffffffffffffffffffffffffffff
in_I16        16'h0000
out_RT        128'h00000000000000000000000000000000
rep_lb32_I16  32'h00000000
```

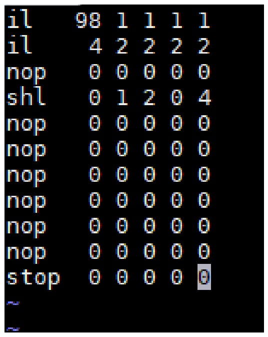
## 3.7.4 Shift and Rotate Instructions

The shift and rotate functions restricted to the preferred slots of input are performed in this execution unit. A unit delay of 4 clock cycles is assigned to these commands before making it available in the forwarding macro. The implementation and the results for the command "**Shift_Left_Word**" is illustrated below.



```
        SHIFT_LEFT_WORD:
            begin
                for (int i=0; i < 4; i++) begin
                    if (in_RB[i*WORD+26 +: 6] < 32) begin
                        RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] << in_RB[i*WORD+26 +: 6];
                    end
                    else begin
                        RT_reg[i*WORD +: WORD] = 'd0;
                    end
                end
                unit_idx = 3'd2;
                rt_wr_en = 1;
            end
```

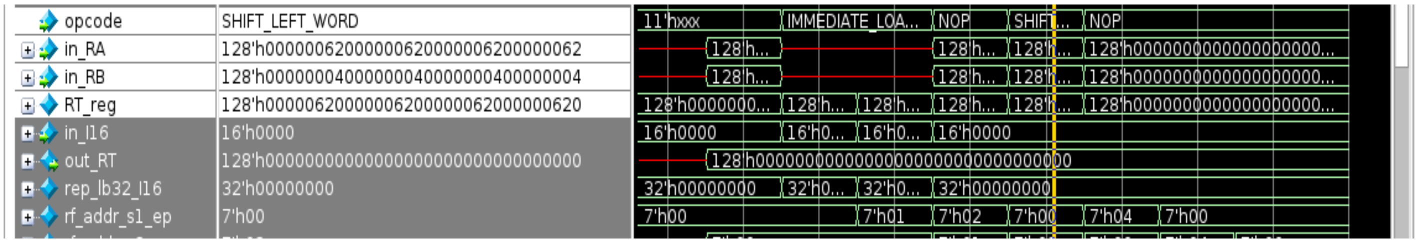
Instructions:



```
il    98 1 1 1 1
il     4 2 2 2 2
nop    0 0 0 0 0
shl    0 1 2 0 4
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
stop   0 0 0 0 0
~
~
```

Simulation result:

The result in Rt_reg is left shifted by 4 bits from RA register.

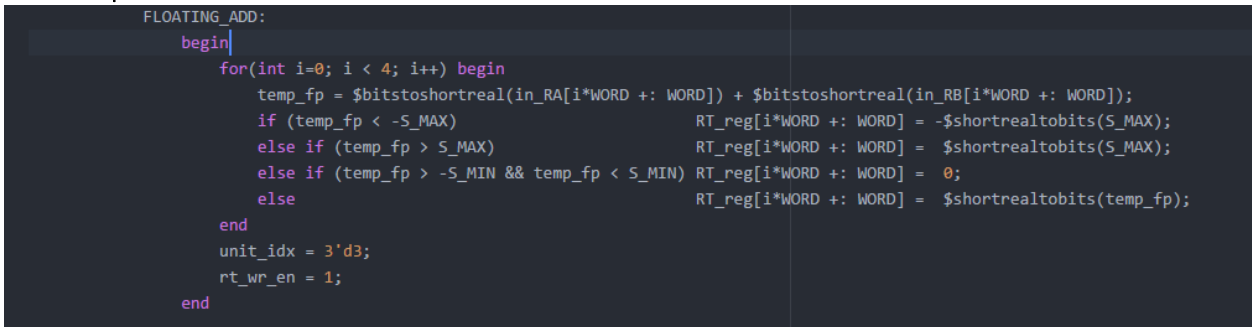


```
opcode         SHIFT_LEFT_WORD                            11'hxxx   IMMEDIATE_LOA...  NOP  SHIFT...  NOP
in_RA          128'h00000062000000620000006200000062       128'h...  128'h...  128'h...  128'h000000000000000000...
in_RB          128'h00000004000000040000000400000004       128'h...  128'h...  128'h...  128'h000000000000000000...
RT_reg         128'h00000620000006200000062000000620       128'h0000000...  128'h...  128'h...  128'h...  128'h...  128'h000000000000000000...
in_I16         16'h0000                                   16'h0000  16'h0...  16'h0...  16'h0000
out_RT         128'h00000000000000000000000000000000       128'h00000000000000000000000000000000
rep_lb32_I16   32'h00000000                               32'h00000000  32'h0...  32'h0...  32'h00000000
rf_addr_s1_ep  7'h00                                      7'h00  7'h01  7'h02  7'h00  7'h04  7'h00
```

## 3.7.5 Floating-Point Instructions

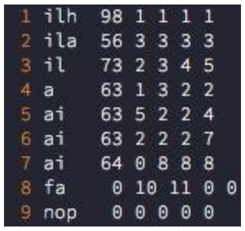
The floating point inputs are evaluated in this execution unit. A unit delay of 6 cycles is assigned to floating point instructions. The implementation of **floating add** is illustrated below,

Code implementation:



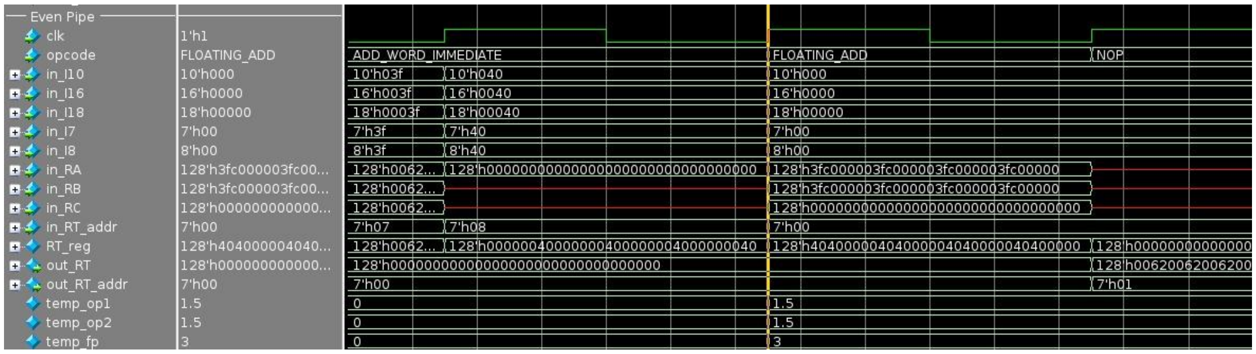
```
FLOATING_ADD:
    begin
        for(int i=0; i < 4; i++) begin
            temp_fp = $bitstoshortreal(in_RA[i*WORD +: WORD]) + $bitstoshortreal(in_RB[i*WORD +: WORD]);
            if (temp_fp < -S_MAX)                       RT_reg[i*WORD +: WORD] = -$shortrealtobits(S_MAX);
            else if (temp_fp > S_MAX)                   RT_reg[i*WORD +: WORD] =  $shortrealtobits(S_MAX);
            else if (temp_fp > -S_MIN && temp_fp < S_MIN) RT_reg[i*WORD +: WORD] =  0;
            else                                        RT_reg[i*WORD +: WORD] =  $shortrealtobits(temp_fp);
        end
        unit_idx = 3'd3;
        rt_wr_en = 1;
    end
```

Instructions:



```
ilh   98 1 1 1 1
ila   56 3 3 3 3
il    73 2 3 4 5
a     63 1 3 2 2
ai    63 5 2 2 4
ai    63 2 2 2 7
ai    64 0 8 8 8
fa     0 10 11 0 0
nop    0 0 0 0 0
```

Simulation result :



```
Even Pipe
clk           1'h1
opcode        FLOATING_ADD              ADD_WORD_IMMEDIATE                          FLOATING_ADD                                  NOP
in_I10        10'h000                   10'h03f     10'h040                         10'h000
in_I16        16'h0000                  16'h003f    16'h0040                        16'h0000
in_I18        18'h00000                 18'h0003f   18'h00040                       18'h00000
in_I7         7'h00                     7'h3f       7'h40                           7'h00
in_I8         8'h00                     8'h3f       8'h40                           8'h00
in_RA         128'h3fc000003fc00...     128'h0062...  128'h00000000000000000000000000000000  128'h3fc000003fc000003fc000003fc00000
in_RB         128'h3fc000003fc00...     128'h0062...                                128'h3fc000003fc000003fc000003fc00000
in_RC         128'h000000000000...      128'h0062...                                128'h00000000000000000000000000000000
in_RT_addr    7'h00                     7'h07       7'h08                           7'h00
RT_reg        128'h404000004040...      128'h0062...  128'h00000040000000400000004000000040  128'h40400000404000004040000040400000  128'h0000000000000000
out_RT        128'h000000000000...      128'h00000000000000000000000000000000                                                     128'h0062006200620O
out_RT_addr   7'h00                     7'h00                                                                                     7'h01
temp_op1      1.5                       0                                           1.5
temp_op2      1.5                       0                                           1.5
temp_fp       3                         0                                           3
```

## 3.8 Odd Pipe (odd_pipe.sv)

Even pipe contains the following executional units mentioned, along with the instruction latency. The execution of each instruction type is explained in the following section.

| Executional Unit | Instruction type | Latency |
|---|---|---|
| Simple Fixed | Memory-Load/Store Instructions | 6 |
| Permute | Shift-Rotate Instructions | 4 |
| Branch | Branch, and Halt Instructions | 4 |

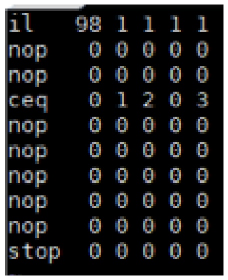
The latency column denotes the number of cycles before the result is available in the forwarding macro.

### 3.8.1 Memory-Load/Store Instructions

Memory load – Store instructions load or store the (immediate value or register value) to the local store. The output address is calculated (parameter LSA) and is written onto the port out_ls_addr. The data from the local store is fed to the odd pipe using the port in_ls_data. The implementation of Load_quadword_absolute is illustrated below.

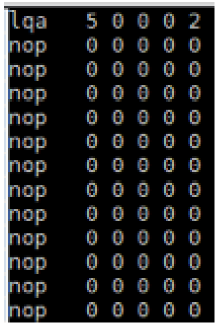
Instructions :

Even Pipe



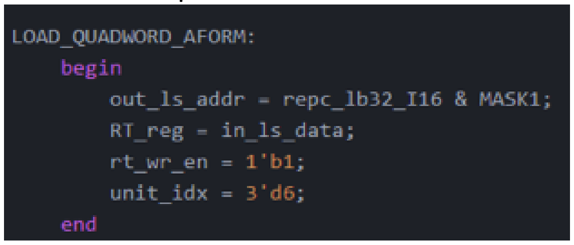
```
il    98 1 1 1 1
nop    0 0 0 0 0
nop    0 0 0 0 0
ceq    0 1 2 0 3
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
stop   0 0 0 0 0
```

Odd pipe



```
lqa    5 0 0 0 2
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
```

Command implementation:

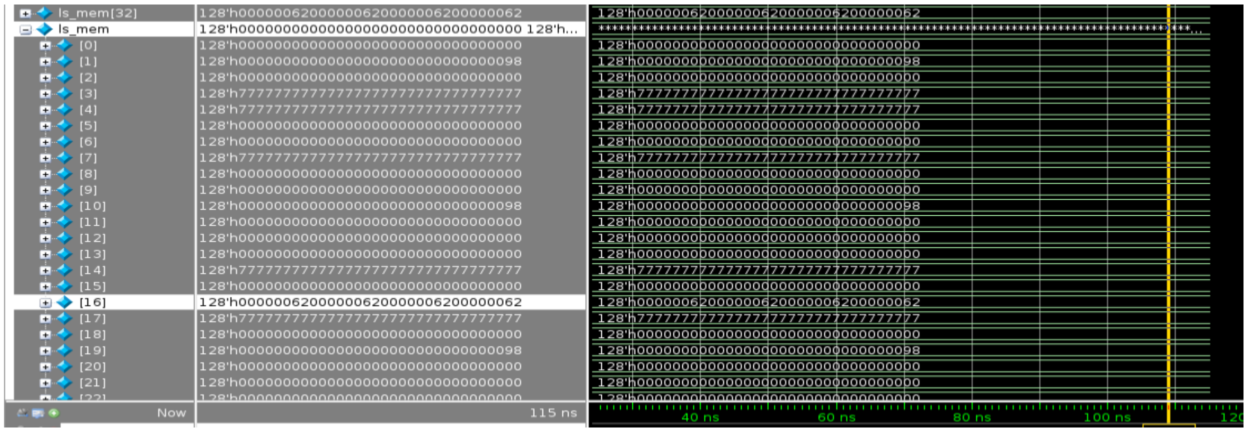


```
LOAD_QUADWORD_AFORM:
    begin
        out_ls_addr = repc_lb32_I16 & MASK1;
        RT_reg = in_ls_data;
        rt_wr_en = 1'b1;
        unit_idx = 3'd6;
    end
```

Simulation Results:

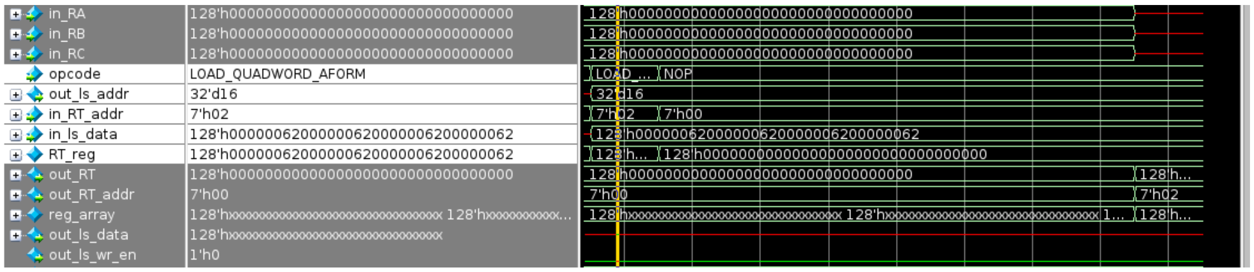
The values are preloaded into the memory for demonstration purpose. The LSA for the instruction is calculated as,

LSA = out_ls_addr = repc_lb32_I16 & 128'hFFFFFFF0 = 32’d0016.
At ls_mem[16] a arbitrary value is loaded,



```
ls_mem[32]   128'h00000062000000620000006200000062
ls_mem       128'h00000000000000000000000000000000 128'h...
  [0]        128'h00000000000000000000000000000000
  [1]        128'h00000000000000000000000000000098
  [2]        128'h00000000000000000000000000000000
  [3]        128'h77777777777777777777777777777777
  [4]        128'h77777777777777777777777777777777
  [5]        128'h00000000000000000000000000000000
  [6]        128'h00000000000000000000000000000000
  [7]        128'h77777777777777777777777777777777
  [8]        128'h00000000000000000000000000000000
  [9]        128'h00000000000000000000000000000000
  [10]       128'h00000000000000000000000000000098
  [11]       128'h00000000000000000000000000000000
  [12]       128'h00000000000000000000000000000000
  [13]       128'h00000000000000000000000000000000
  [14]       128'h77777777777777777777777777777777
  [15]       128'h00000000000000000000000000000000
  [16]       128'h00000062000000620000006200000062
  [17]       128'h77777777777777777777777777777777
  [18]       128'h00000000000000000000000000000000
  [19]       128'h00000000000000000000000000000098
  [20]       128'h00000000000000000000000000000000
  [21]       128'h00000000000000000000000000000000
Now          115 ns
```

The value at ls_mem[16] is loaded in to Rt_reg  by lqa command in this instance.



```
in_RA        128'h00000000000000000000000000000000
in_RB        128'h00000000000000000000000000000000
in_RC        128'h00000000000000000000000000000000
opcode       LOAD_QUADWORD_AFORM
out_ls_addr  32'd16
in_RT_addr   7'h02
in_ls_data   128'h00000062000000620000006200000062
RT_reg       128'h00000062000000620000006200000062
out_RT       128'h00000000000000000000000000000000
out_RT_addr  7'h00
reg_array    128'hxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 128'hxxxxxxxxxxx...
out_ls_data  128'hxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
out_ls_wr_en 1'h0
```

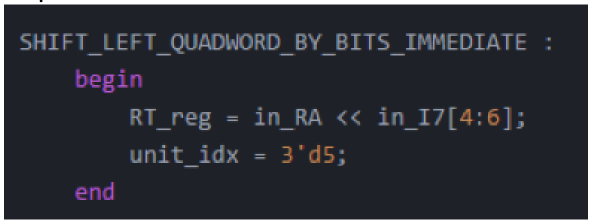
This value is written to the output register (reg_mem[2]) after 6 clock cycles ans is made available for future instructions.

### 3.8.2 Shift-Rotate Instructions

The Shift and rotate instructions in the permute unit are also defined in the odd pipe. They are assigned a latency of 4. The Shift-rotate operations in this section works on the whole of 128 bits of the register data instead of operating on data in the preferred slots.

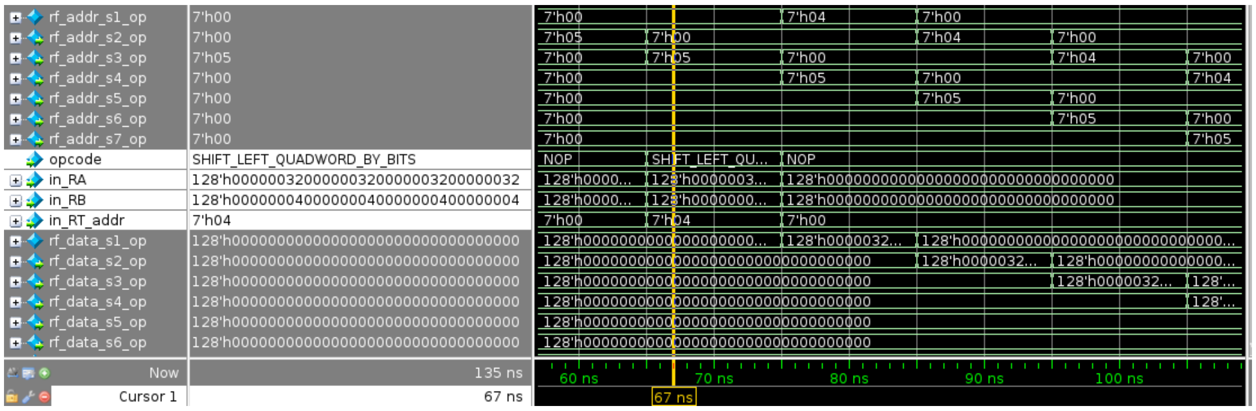
The implementation and the Shift_Left_QuadWord_by_Bits is illustrated below.

Implementation:

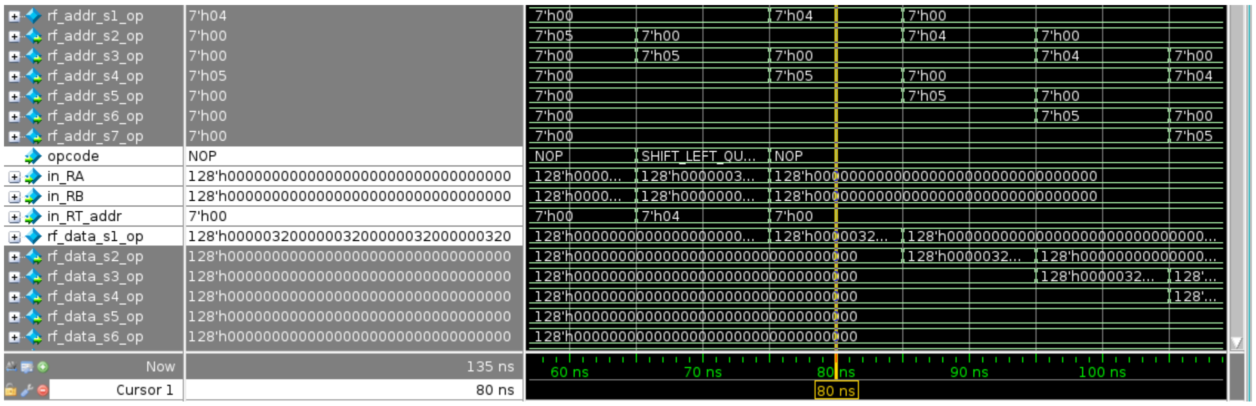


```
SHIFT_LEFT_QUADWORD_BY_BITS_IMMEDIATE :
    begin
        RT_reg = in_RA << in_I7[4:6];
        unit_idx = 3'd5;
    end
```

Simulation output:
Register reg_2 contains 8’h00000032 in each word slot which is loaded in to register RA(shown at in_RA). This value is shifted by 4 bits and is stored in rf_data_s1_op. The value becomes available in register 4 (Output register, out_RT) after 4 clock cycles.



```
rf_addr_s1_op    7'h00
rf_addr_s2_op    7'h00
rf_addr_s3_op    7'h05
rf_addr_s4_op    7'h00
rf_addr_s5_op    7'h00
rf_addr_s6_op    7'h00
rf_addr_s7_op    7'h00
opcode           SHIFT_LEFT_QUADWORD_BY_BITS
in_RA            128'h00000032000000320000003200000032
in_RB            128'h00000004000000040000000400000004
in_RT_addr       7'h04
rf_data_s1_op    128'h00000000000000000000000000000000
rf_data_s2_op    128'h00000000000000000000000000000000
rf_data_s3_op    128'h00000000000000000000000000000000
rf_data_s4_op    128'h00000000000000000000000000000000
rf_data_s5_op    128'h00000000000000000000000000000000
rf_data_s6_op    128'h00000000000000000000000000000000
Now              135 ns
Cursor 1         67 ns
```

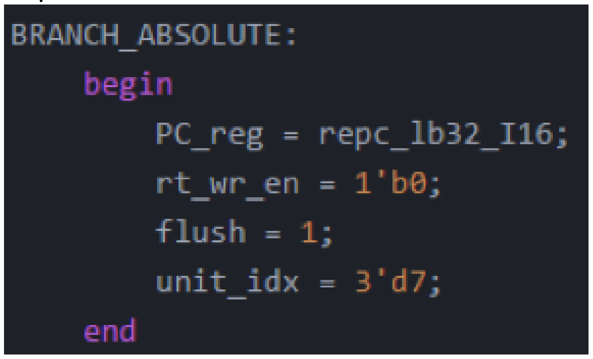


```
rf_addr_s1_op    7'h04
rf_addr_s2_op    7'h00
rf_addr_s3_op    7'h00
rf_addr_s4_op    7'h05
rf_addr_s5_op    7'h00
rf_addr_s6_op    7'h00
rf_addr_s7_op    7'h00
opcode           NOP
in_RA            128'h00000000000000000000000000000000
in_RB            128'h00000000000000000000000000000000
in_RT_addr       7'h00
rf_data_s1_op    128'h00000320000003200000032000000320
rf_data_s2_op    128'h00000000000000000000000000000000
rf_data_s3_op    128'h00000000000000000000000000000000
rf_data_s4_op    128'h00000000000000000000000000000000
rf_data_s5_op    128'h00000000000000000000000000000000
rf_data_s6_op    128'h00000000000000000000000000000000
Now              135 ns
Cursor 1         80 ns
```

### 3.8.3 Branch Instruction

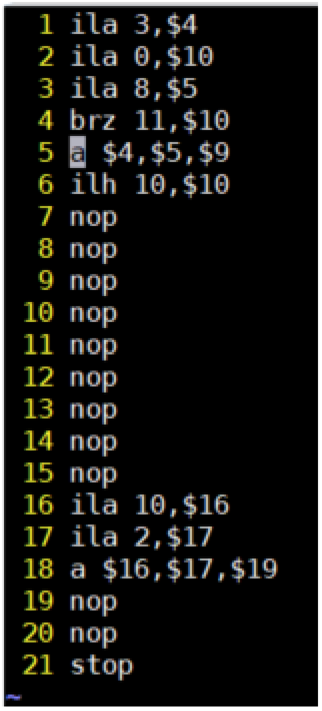
The current implementation of the branch instruction always assumes the branch as "not taken". The implementation and the working of  "branch absolute" command is demonstrated in the following section.

Implementation:



```
BRANCH_ABSOLUTE:
    begin
        PC_reg = repc_lb32_I16;
        rt_wr_en = 1'b0;
        flush = 1;
        unit_idx = 3'd7;
    end
```

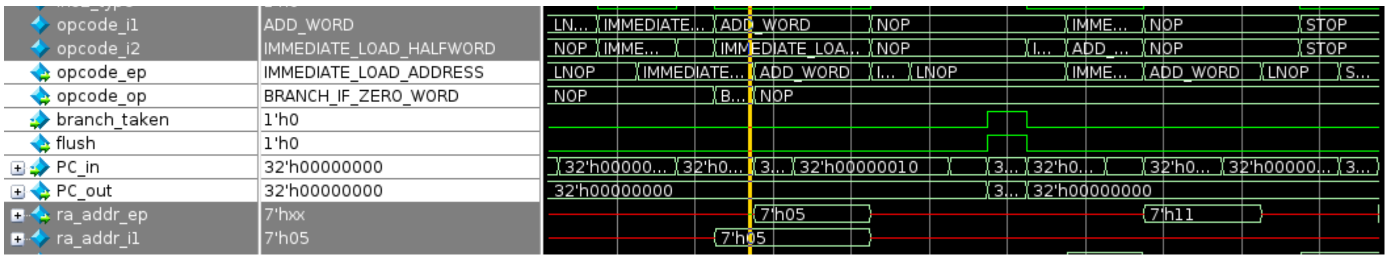
Instructions:



```
 1 ila 3,$4
 2 ila 0,$10
 3 ila 8,$5
 4 brz 11,$10
 5 a $4,$5,$9
 6 ilh 10,$10
 7 nop
 8 nop
 9 nop
10 nop
11 nop
12 nop
13 nop
14 nop
15 nop
16 ila 10,$16
17 ila 2,$17
18 a $16,$17,$19
19 nop
20 nop
21 stop
~
```

In the above program a branch is taken on the instruction 4 and program is directed to jump to instruction 16. The addition and load instructions executed immediately after branch command are flushed when a branch taken signal is given out.

Simulation results:
The execution pipeline of both odd and even pipe along with branch taken signal is shown below. A flush signal is given out when the branch is taken instructing the system to  flush out all the commands which were executed before taking the branch. The PC is updated after the branch instruction is determined as taken.



```
opcode_i1      ADD_WORD
opcode_i2      IMMEDIATE_LOAD_HALFWORD
opcode_ep      IMMEDIATE_LOAD_ADDRESS
opcode_op      BRANCH_IF_ZERO_WORD
branch_taken   1'h0
flush          1'h0
PC_in          32'h00000000
PC_out         32'h00000000
ra_addr_ep     7'hxx
ra_addr_i1     7'h05
```

LN... IMMEDIATE... ADD_WORD NOP IMME... NOP STOP
NOP IMME... IMMEDIATE_LOA... NOP I... ADD_... NOP STOP
LNOP IMMEDIATE... ADD_WORD I... LNOP IMME... ADD_WORD LNOP S...
NOP B... NOP
32'h00000... 32'h0... 3... 32'h00000010 3... 32'h0... 32'h0... 32'h00000... 3...
32'h00000000 3... 32'h00000000
7'h05 7'h11
7'h05

## 3.9 Register File (reg_file.sv)

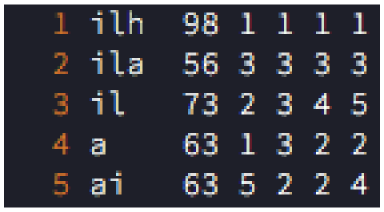
General-purpose registers are defined in the file. There are six read ports and two write ports.

## 3.10 Forwarding macro (fw_macro.sv)

The forwarding macro stores the intermediate results of the instructions in the pipeline. Depending on the assigned delay values, the results of the instructions become available in the forwarding macro to be used by future instructions. The intermediate results that are evaluated in one pipe can be forwarded to other pipe during the execution of a program. The illustration of data forwarding is demonstrated in the following examples.

### 3.10.1 Example execution (even pipe to even pipe forwarding)

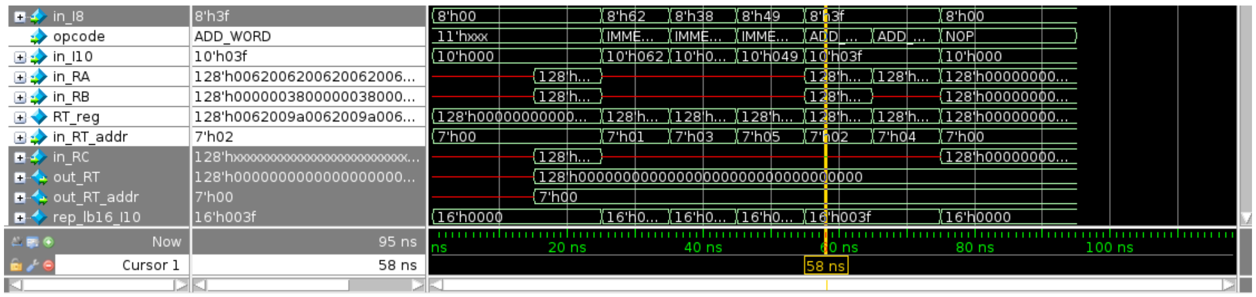
Instructions:



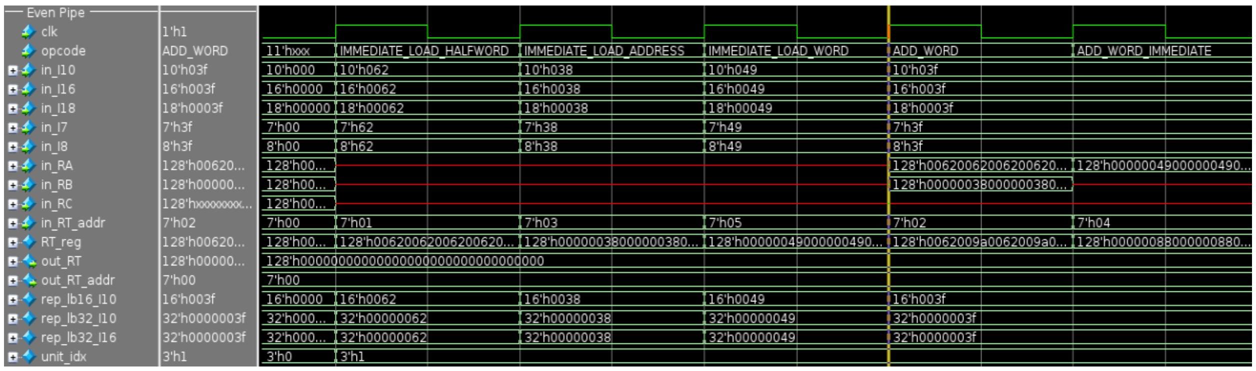
```
1 ilh  98 1 1 1 1
2 ila  56 3 3 3 3
3 il   73 2 3 4 5
4 a    63 1 3 2 2
5 ai   63 5 2 2 4
```

Waveforms:

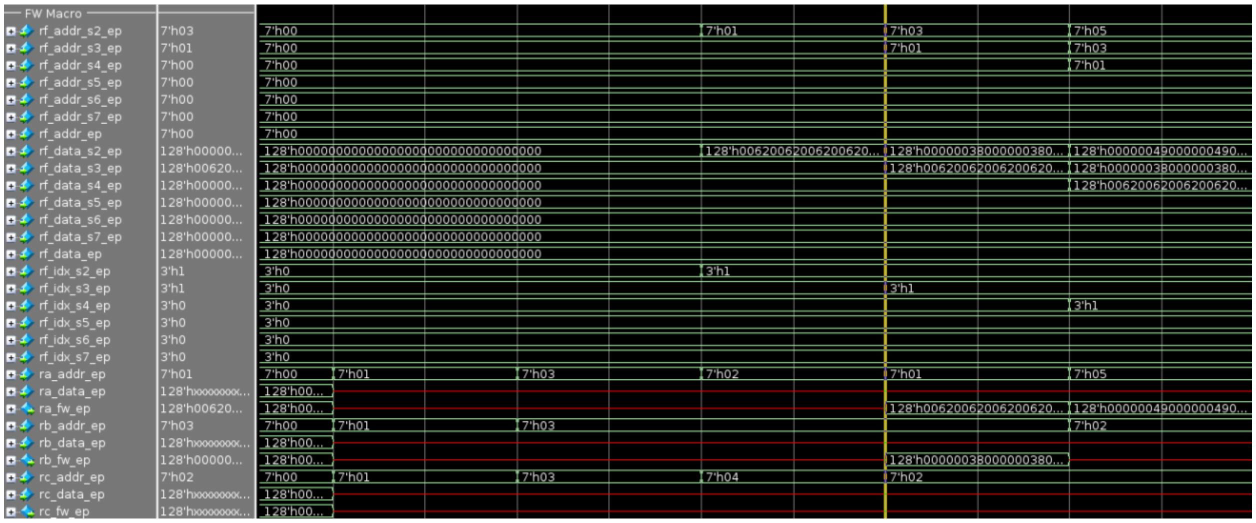
Even Pipe to even pipe: The values that are loaded using immediate load commands are used in "add word" opcode. The results of commands 1 and 2 becomes available after 2 clock cycles, i.e at the execution of command line 4.



```
in_I8          8'h3f
opcode         ADD_WORD
in_I10         10'h03f
in_RA          128'h0062006200620062006...
in_RB          128'h0000003800000038000...
RT_reg         128'h0062009a0062009a006...
in_RT_addr     7'h02
in_RC          128'hxxxxxxxxxxxxxxxxxxxxxxxxxx...
out_RT         128'h00000000000000000000...
out_RT_addr    7'h00
rep_lb16_I10   16'h003f
Now            95 ns
Cursor 1       58 ns
```



```
Even Pipe
clk            1'h1
opcode         ADD_WORD
in_I10         10'h03f
in_I16         16'h003f
in_I18         18'h0003f
in_I7          7'h3f
in_I8          8'h3f
in_RA          128'h00620...
in_RB          128'h00000...
in_RC          128'hxxxxxxxx...
in_RT_addr     7'h02
RT_reg         128'h00620...
out_RT         128'h00000...
out_RT_addr    7'h00
rep_lb16_I10   16'h003f
rep_lb32_I10   32'h0000003f
rep_lb32_I16   32'h0000003f
unit_idx       3'h1
```



## 3.10.2 Example execution (odd pipe to even pipe forwarding)

Instructions:

Odd Pipe

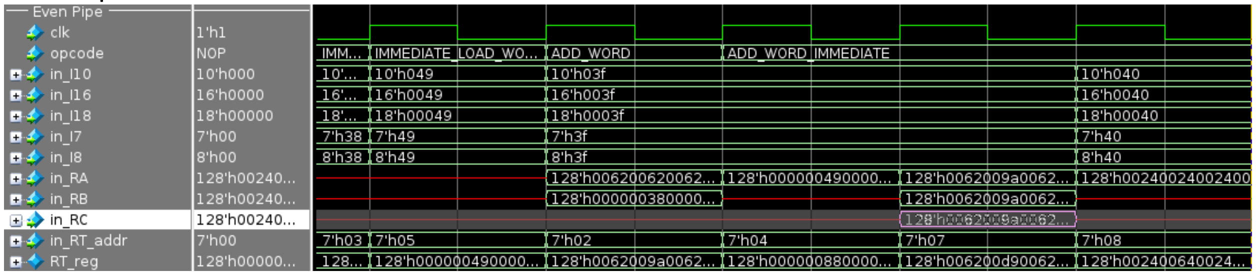
```
lqa    4 0 0 1 0
stqa   6 0 0 0 1
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
nop    0 0 0 0 0
```

Even Pipe

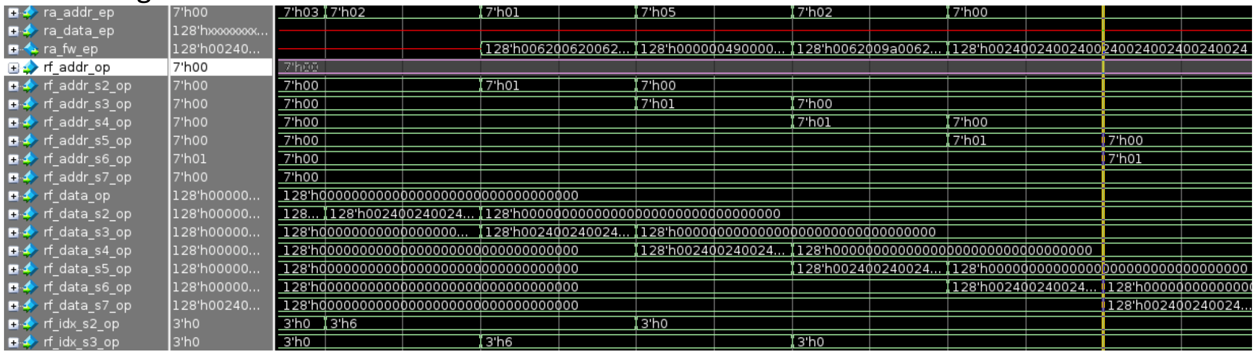
```
ilh   98 1 1 1 1
ila   56 3 3 3 3
il    73 2 3 4 5
a     63 1 3 2 2
ai    63 5 2 2 4
ai    63 2 2 2 7
ai    64 0 8 8 8
```

Waveforms:

Even Pipe



Forwarding Macro



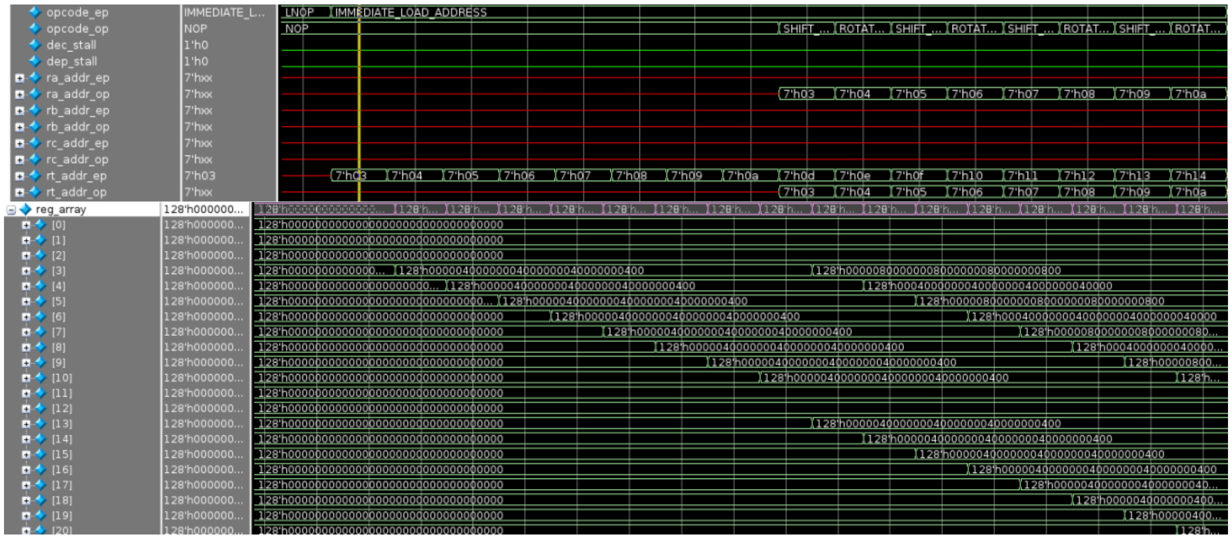
# 4 Example Programs

## 4.1 No Hazards

### 4.1.1 Code

```
ila 1024,$3
nop
ila 1024,$4
nop
ila 1024,$5
nop
ila 1024,$6
nop
ila 1024,$7
nop
ila 1024,$8
nop
ila 1024,$9
nop
ila 1024,$10
nop
shlqbii 1,$3,$3
ila 1024,$13
rotqbyi 1,$4,$4
ila 1024,$14
shlqbii 1,$5,$5
ila 1024,$15
rotqbyi 1,$6,$6
ila 1024,$16
shlqbii 1,$7,$7
ila 1024,$17
rotqbyi 1,$8,$8
ila 1024,$18
shlqbii 1,$9,$9
ila 1024,$19
rotqbyi 1,$10,$10
ila 1024,$20
nop
lnop
nop
lnop
nop
lnop
nop
lnop
nop
lnop
stop
nop
lnop
nop
```

### 4.1.2 Waveform

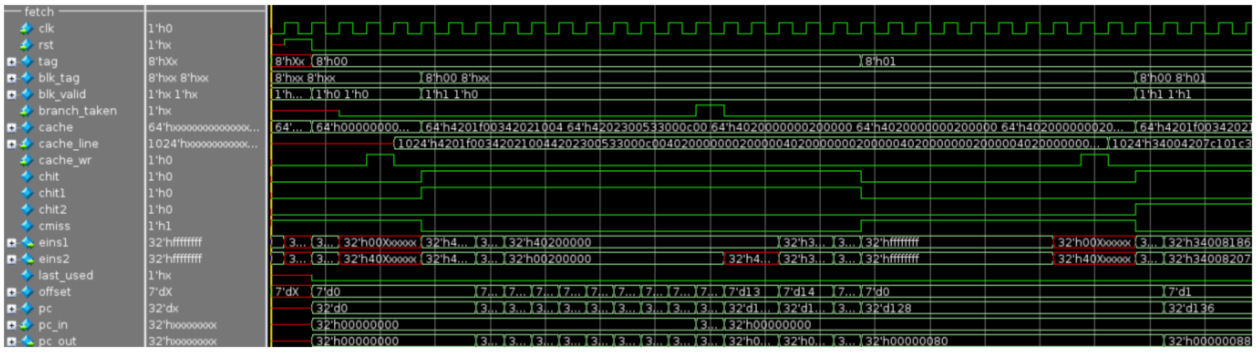


## 4.2 Cache Miss Handling

### 4.2.1 Code

```
ila 992,$3
ila 1056,$4
ila 1120,$5
brsl 24,$lr
nop
lnop
nop
lnop
nop
lnop
nop
lnop
nop
lnop
nop
lnop
nop
lnop
nop
lnop
nop
lnop
nop
lnop
nop
lnop
nop
stop
lqd 0,$3,$6
lqd 0,$4,$7
mpy $6,$7,$8
lqd 1,$3,$6
lqd 1,$4,$7
mpya $8,$7,$6,$8
lqd 2,$3,$6
lqd 2,$4,$7
mpya $8,$7,$6,$8
lqd 3,$3,$6
lqd 3,$4,$7
mpya $8,$7,$6,$8
stqd 0,$5,$8
bi $lr
lnop
nop
lnop
nop
lnop
nop
lnop
nop
```

### 4.2.2 Waveform



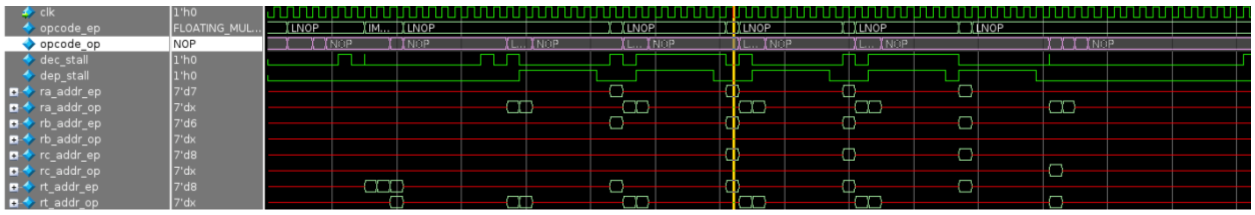
## 4.3 4x4 Floating Point Matrix Multiply

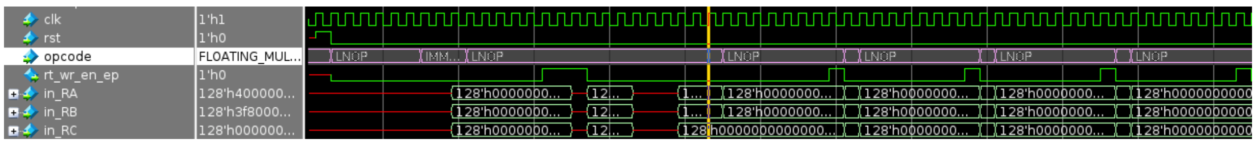
One of the matrix is assumed to be in transpose form.

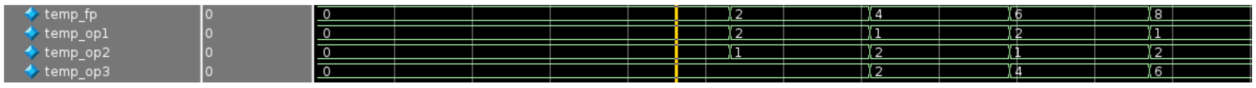
### 4.3.1 Code

```
ila 992,$3
ila 1056,$4
ila 1120,$5
brsl 15,$lr
nop
lnop
nop
lnop
nop
lnop
nop
lnop
nop
lnop
nop
lnop
nop
stop
lqd 0,$3,$6
lqd 0,$4,$7
fm $6,$7,$8
lqd 1,$3,$6
lqd 1,$4,$7
fma $8,$7,$6,$8
lqd 2,$3,$6
lqd 2,$4,$7
fma $8,$7,$6,$8
lqd 3,$3,$6
lqd 3,$4,$7
fma $8,$7,$6,$8
stqd 0,$5,$8
bi $lr
nop
lnop
nop
lnop
nop
lnop
nop
lnop
nop
lnop
```

### 4.3.2 Waveform







## 5 Appendix

### 5.1 Appendix A – Assembly Parser Code (sasm.py)

```
#!/usr/bin/env python2

import os
import sys
import re
import subprocess

from subprocess import call

ins_opcode = {}
ins_type   = {}

def print_usage():
    print sys.argv[0] + " <input file> <output file> <mode>."
    print "<input file> : Input assembly instruction file."
    print "<output file> : Output machine coded file."
    print "<mode> : 0 - Input is in assembly language, for full SPU sim."
    print "       : 1 - Input is in intermediate assembly language, for SPU pipes sim.\n"

print "\n!!Welcome to SPU Mini Assembler!!"

if(len(sys.argv) != 4):
    print "Invalid number of arguments\n"
    print_usage();
    sys.exit(0);
else:
    fname = sys.argv[1]
    fout = sys.argv[2]
    mode = int(sys.argv[3])
    print "Input file: " + fname + ", Output file: " + fout + ", Mode: " + sys.argv[3]

with open("ins_dict.txt", 'r') as f:
    for line in f:
        items = line.split()
        key, value1, value2, value3 = items[0], items[1], items[2], items[3]
        if(mode == 1):
            ins_opcode[key] = value2
        else:
            ins_opcode[key] = value1
        ins_type[key] = value3
    f.close()

with open(fname) as f:
    fw = open(fout, 'w')
    for line in f:
        ls = line.strip()
        if(ls[0] == '#'):
            continue

        sys.stdout.write('.')
        sys.stdout.flush()
        #TODO: may have to strip all spaces when splitting
        words = line.split()
        mcode = ""
        opcode = words[0]

        if(opcode[0] == '/' and opcode[1] == '/'):
            continue

        if(mode == 1):
            syms = words[1].split(',')
            mcode += str(bin(int(ins_opcode[opcode]))[2:].zfill(11))
            mcode += str(bin(int(syms[0]))[2:].zfill(18))
            mcode += str(bin(int(syms[1]))[2:].zfill(7))
            mcode += str(bin(int(syms[2]))[2:].zfill(7))
            mcode += str(bin(int(syms[3]))[2:].zfill(7))
            mcode += str(bin(int(syms[4]))[2:].zfill(7))
        else:
            #TODO: Have to write special cases for some instructions...
            itype = ins_type[opcode]
            syms = ""
            if(len(words) > 1):
                syms = words[1].split(',')
            if(len(syms) > 0):
                if(syms[0] == "$lr"):
                    syms[0] = '0'
                elif(syms[0] == "$sp"):
                    syms[0] = '1'
                elif(syms[0][0] == '$'):
                    syms[0] = syms[0][1:]
```

```
if(len(syms) > 1):
    if(syms[1] == "$lr"):
        syms[1] = '0'
    elif(syms[1] == "$sp"):
        syms[1] = '1'
    elif(syms[1][0] == '$'):
        syms[1] = syms[1][1:]
if(len(syms) > 2):
    if(syms[2] == "$lr"):
        syms[2] = '0'
    elif(syms[2] == "$sp"):
        syms[2] = '1'
    elif(syms[2][0] == '$'):
        syms[2] = syms[2][1:]
if(len(syms) > 3):
    if(syms[3] == "$lr"):
        syms[3] = '0'
    elif(syms[3] == "$sp"):
        syms[3] = '1'
    elif(syms[3][0] == '$'):
        syms[3] = syms[3][1:]

if(itype == "1" or itype == "3"): #RR or RI17
    #syms = words[1].split(',')
    mcode += str(bin(int(ins_opcode[opcode]))[2:].zfill(11))
    if(opcode == "bi"):
        mcode += str(bin(int("0") % (1 << 7))[2:].zfill(7))
        mcode += str(bin(int(syms[0]) % (1 << 7))[2:].zfill(7))
        mcode += str(bin(int("0") % (1 << 7))[2:].zfill(7))
    else:
        mcode += str(bin(int(syms[0]) % (1 << 7))[2:].zfill(7))
        mcode += str(bin(int(syms[1]) % (1 << 7))[2:].zfill(7))
        mcode += str(bin(int(syms[2]) % (1 << 7))[2:].zfill(7))
elif(itype == "2"): #RRR
    mcode += str(bin(int(ins_opcode[opcode]))[2:].zfill(4))
    mcode += str(bin(int(syms[0]) % (1 << 7))[2:].zfill(7))
    mcode += str(bin(int(syms[1]) % (1 << 7))[2:].zfill(7))
    mcode += str(bin(int(syms[2]) % (1 << 7))[2:].zfill(7))
    mcode += str(bin(int(syms[3]) % (1 << 7))[2:].zfill(7))
elif(itype == "4"): #RI10
    mcode += str(bin(int(ins_opcode[opcode]))[2:].zfill(8))
    mcode += str(bin(int(syms[0]) % (1 << 10))[2:].zfill(10))
    mcode += str(bin(int(syms[1]) % (1 << 7))[2:].zfill(7))
    mcode += str(bin(int(syms[2]) % (1 << 7))[2:].zfill(7))
```

```
            elif(itype == "5"): #RI16
                mcode += str(bin(int(ins_opcode[opcode]))[2:].zfill(9))
                if(opcode == "br" or opcode == "bra"):
                    mcode += str(bin(int(syms[0]) % (1 << 16))[2:].zfill(16))
                    mcode += str(bin(int("0") % (1 << 7))[2:].zfill(7))
                else:
                    mcode += str(bin(int(syms[0]) % (1 << 16))[2:].zfill(16))
                    mcode += str(bin(int(syms[1]) % (1 << 7))[2:].zfill(7))
            elif(itype == "6"): #RI18
                mcode += str(bin(int(ins_opcode[opcode]))[2:].zfill(7))
                mcode += str(bin(int(syms[0]) % (1 << 18))[2:].zfill(18))
                mcode += str(bin(int(syms[1]) % (1 << 7))[2:].zfill(7))
            else:
                mcode += str(bin(int(ins_opcode[opcode]))[2:].zfill(11))
                mcode += str(bin(int("0") % (1 << 21))[2:].zfill(21))

        fw.write(mcode + "\n")

    print "\n"
    fw.close()
    f.close()

with open(fout) as f:
    num_ins = 0
    fw = open("ls_load_file.txt", 'w')
    for line in f:
        by = re.findall('........', line)
        for word in by:
            fw.write(word + "\n")
            num_ins+=1

    word = "00000000"
    while num_ins < 992:
        fw.write(word + "\n")
        num_ins+=1

    fd = open("data_file.txt", 'r')
    for line in fd:
        fw.write(line)

print "!!SPU Mini Assembler finished successfully!!\n"
```

## 5.2 Appendix B – Local Store (local_store.sv)

```
import defines_pkg::*;

module local_store #(parameter MEM_SIZE = LS_SIZE)
(
   input  logic                clk,
   input  logic                rst,
   input  logic [0:127]         ls_data_wr,
   input  logic [0:31]          ls_addr,
   input  logic                ls_wr_en,
   input  logic                cache_wr,
   output logic [0:1023]        cache_out,
   output logic [0:127]         ls_data_rd
);

   logic [0:7] ls_mem[MEM_SIZE];

   initial begin
      $readmemb(LSLOADFILE, ls_mem);
   end

   always_ff @(posedge clk) begin
      if(ls_wr_en) begin
         for(int i = 0; i < 16; i++) begin
            ls_mem[ls_addr + i] <= ls_data_wr[i*8 +: 8];
         end
      end

      if(cache_wr) begin
         for(int i = 0; i < 256; i++) begin
            cache_out[i*8 +: 8] = ls_mem[ls_addr + i];
         end
      end
   end

   always_comb begin
      for(int i = 0; i < 16; i++) begin
         ls_data_rd[i*8 +: 8] = (ls_wr_en) ? ls_data_wr[i*8 +: 8] : ls_mem[ls_addr + i];
      end
   end

endmodule
//end of file.
```

## 5.3 Appendix C – Fetch (fetch.sv)

```
import defines_pkg::*;

module fetch
(
   input  logic              clk,
   input  logic              rst,
   input  logic  [0:1023]   cache_line,
   input  logic              cache_wr,
   input  logic              branch_taken,
   input  logic              dec_stall,
   input  logic              dep_stall,
   input  logic  [0:31]     pc_in,
   output logic  [0:31]     pc_out,
   output logic  [0:31]     eins1,
   output logic  [0:31]     eins2
);

   logic [0:63]        cache[32];
   logic [0:31]        pc;
   logic [0:7]         blk_tag[2];
   logic               blk_valid[2];

   logic [0:7]         tag;
   logic [0:6]         offset;
   logic               last_used;
   logic               chit1;
   logic               chit2;
   logic               chit;
   logic               cmiss;
   logic               cw_pending;
   logic               cache_wr_dly;
   logic               only1_inst;

   always_ff @(posedge clk) begin
      if(rst) begin
         cw_pending <= 'd0;
      end
      else begin
         if(cw_pending && chit) begin
            cw_pending <= 'd0;
         end
         else if(cmiss && !dep_stall) begin
```

```
            cw_pending <= 'd1;
        end
    end
end

always_ff @(posedge clk) begin
    if(rst) begin
        cache_wr_dly <= 'd0;
    end
    else begin
        cache_wr_dly <= cache_wr;
    end
end

always_ff @(posedge clk) begin
    if(rst) begin
        pc <= 'd0;
    end
    else begin
        if(branch_taken) begin
            pc <= pc_in;
        end
        else if(dec_stall || dep_stall) begin
            pc <= pc;
        end
        else if(!cmiss) begin
            pc <= (only1_inst) ? pc + 4 : pc + 8;
        end
    end
end

always_ff @(posedge clk) begin
    if(rst) begin
        cache <= '{default:'0};
        blk_tag <= '{default:'dx};
        blk_valid <= '{default:'0};
        last_used <= 'd0;
    end
    else if(cache_wr_dly) begin
        if(!blk_valid[0]) begin
            blk_tag[0] <= tag;
            blk_valid[0] <= 1;
            for(int i = 0; i < 16; i++) begin
                cache[i] <= cache_line[i*64 +: 64];
            end
        end
        else if(!blk_valid[1]) begin
            blk_tag[1] <= tag;
            blk_valid[1] <= 1;
            for(int i = 0; i < 16; i++) begin
                cache[i+16] <= cache_line[i*64 +: 64];
            end
        end
        else if(last_used) begin
            blk_tag[0] <= tag;
            blk_valid[0] <= 1;
            last_used <= ~last_used;
            for(int i = 0; i < 16; i++) begin
                cache[i] <= cache_line[i*64 +: 64];
            end
        end
        else begin
            blk_tag[1] <= tag;
            blk_valid[1] <= 1;
            last_used <= ~last_used;
            for(int i = 0; i < 16; i++) begin
                cache[i+16] <= cache_line[i*64 +: 64];
            end
        end
    end
end

always_comb begin
    tag = pc[19:24];
    offset = pc[25:31] >> 3;
    only1_inst = pc[29];
    pc_out = pc;
    chit1 = (tag === blk_tag[0] && blk_valid[0]);
    chit2 = (tag === blk_tag[1] && blk_valid[1]);
    chit  = chit1 || chit2;
    cmiss = ~chit;
    if(chit1) begin
        eins1 = cache[offset][0:31];
        eins2 = (only1_inst) ? {11'b01000000001, 21'dx} : cache[offset][32:63];
    end
    else if(chit2) begin
        eins1 = cache[16 + offset][0:31];
        eins2 = (only1_inst) ? {11'b01000000001, 21'dx} : cache[16 + offset][32:63];
        end
        else if(cw_pending) begin
            eins1 = {11'b00000000001, 21'dx};
            eins2 = {11'b01000000001, 21'dx};
        end
        else begin
            eins1 = 32'hffffffff;
            eins2 = 32'hffffffff;
        end
    end

endmodule
//end of file.
```

## 5.4 Appendix D – Decode (decode.sv)

```
import defines_pkg::*;

module decode
(
    input  logic            clk,
    input  logic            rst,
    input  logic [0:31]     eins1,
    input  logic [0:31]     eins2,
    input  logic            dep_stall,
    input  logic            flush,
    input  logic [0:31]     pc_in,
    output logic [0:31]     pc_out,
    output logic            dec_stall,
    output Opcodes          opcode_ep,
    output Opcodes          opcode_op,
    output logic [0:6]      ra_addr_ep,
    output logic [0:6]      rb_addr_ep,
    output logic [0:6]      rc_addr_ep,
    output logic [0:6]      rt_addr_ep,
    output logic [0:6]      ra_addr_op,
    output logic [0:6]      rb_addr_op,
    output logic [0:6]      rc_addr_op,
    output logic [0:6]      rt_addr_op,
    output logic [0:6]      in_I7e,
    output logic [0:7]      in_I8e,
    output logic [0:9]      in_I10e,
    output logic [0:15]     in_I16e,
    output logic [0:17]     in_I18e,
    output logic [0:6]      in_l7o,
    output logic [0:7]      in_l8o,
    output logic [0:9]      in_l10o,
    output logic [0:15]     in_l16o,
    output logic [0:17]     in_l18o
);

    localparam EVEN = 0;
    localparam ODD  = 1;

    Opcodes     opcode_i1;
    Opcodes     opcode_i2;
    logic [0:6]  ra_addr_i1;
    logic [0:6]  rb_addr_i1;
    logic [0:6]  rc_addr_i1;
    logic [0:6]  rt_addr_i1;
    logic [0:6]  ra_addr_i2;
    logic [0:6]  rb_addr_i2;
    logic [0:6]  rc_addr_i2;
    logic [0:6]  rt_addr_i2;
    logic [0:6]  in_l7_i1;
    logic [0:7]  in_l8_i1;
    logic [0:9]  in_l10_i1;
    logic [0:15] in_l16_i1;
    logic [0:17] in_l18_i1;
    logic [0:6]  in_l7_i2;
    logic [0:7]  in_l8_i2;
    logic [0:9]  in_l10_i2;
    logic [0:15] in_l16_i2;
    logic [0:17] in_l18_i2;

    logic       stall_done;
    logic [0:3]  eoc1_4b;
    logic [0:3]  eoc2_4b;
    logic [0:6]  eoc1_7b;
    logic [0:6]  eoc2_7b;
    logic [0:7]  eoc1_8b;
    logic [0:7]  eoc2_8b;
    logic [0:8]  eoc1_9b;
    logic [0:8]  eoc2_9b;
    logic [0:10] eoc1_11b;
    logic [0:10] eoc2_11b;

    logic       ins1_type;
logic       ins2_type;

assign eoc1_4b  = eins1[0:3];
assign eoc2_4b  = eins2[0:3];
assign eoc1_7b  = eins1[0:6];
assign eoc2_7b  = eins2[0:6];
assign eoc1_8b  = eins1[0:7];
assign eoc2_8b  = eins2[0:7];
assign eoc1_9b  = eins1[0:8];
assign eoc2_9b  = eins2[0:8];
assign eoc1_11b = eins1[0:10];
assign eoc2_11b = eins2[0:10];

always_ff @(posedge clk) begin
  if(rst) begin
    stall_done <= 'd0;
    opcode_ep <= LNOP;
    opcode_op <= NOP;
    ra_addr_ep <= 'dx;
    rb_addr_ep <= 'dx;
    rc_addr_ep <= 'dx;
    rt_addr_ep <= 'dx;
    ra_addr_op <= 'dx;
    rb_addr_op <= 'dx;
    rc_addr_op <= 'dx;
    rt_addr_op <= 'dx;
    in_I7e <= 'dx;
    in_I8e <= 'dx;
    in_I10e <= 'dx;
    in_I16e <= 'dx;
    in_I18e <= 'dx;
    in_I7o <= 'dx;
    in_I8o <= 'dx;
    in_I10o <= 'dx;
    in_I16o <= 'dx;
    in_I18o <= 'dx;
    pc_out <= 'dx;
  end
  else begin
    if(flush) begin
      stall_done <= 'd0;
      opcode_ep <= LNOP;
      opcode_op <= NOP;
      ra_addr_ep <= 'dx;
      rb_addr_ep <= 'dx;
      rc_addr_ep <= 'dx;
      rt_addr_ep <= 'dx;
      ra_addr_op <= 'dx;
      rb_addr_op <= 'dx;
      rc_addr_op <= 'dx;
      rt_addr_op <= 'dx;
      in_l7e <= 'dx;
      in_l8e <= 'dx;
      in_l10e <= 'dx;
      in_l16e <= 'dx;
      in_l18e <= 'dx;
      in_l7o <= 'dx;
      in_l8o <= 'dx;
      in_l10o <= 'dx;
      in_l16o <= 'dx;
      in_l18o <= 'dx;
      pc_out <= pc_in;
  end
  else if(dep_stall) begin
      stall_done <= stall_done;
      opcode_ep <= opcode_ep;
      opcode_op <= opcode_op;
      ra_addr_ep <= ra_addr_ep;
      rb_addr_ep <= rb_addr_ep;
      rc_addr_ep <= rc_addr_ep;
      rt_addr_ep <= rt_addr_ep;
      ra_addr_op <= ra_addr_op;
      rb_addr_op <= rb_addr_op;
      rc_addr_op <= rc_addr_op;
      rt_addr_op <= rt_addr_op;
      in_l7e <= in_l7e;
      in_l8e <= in_l8e;
      in_l10e <= in_l10e;
      in_l16e <= in_l16e;
      in_l18e <= in_l18e;
      in_l7o <= in_l7o;
      in_l8o <= in_l8o;
      in_l10o <= in_l10o;
      in_l16o <= in_l16o;
      in_l18o <= in_l18o;
      pc_out <= pc_in;
  end
  else if(dec_stall || stall_done) begin
```

```
if(stall_done) begin
    if(ins2_type == EVEN) begin
        opcode_ep  <= opcode_i2;
        ra_addr_ep <= ra_addr_i2;
        rb_addr_ep <= rb_addr_i2;
        rc_addr_ep <= rc_addr_i2;
        rt_addr_ep <= rt_addr_i2;
        in_I7e     <= in_I7_i2;
        in_I8e     <= in_I8_i2;
        in_I10e    <= in_I10_i2;
        in_I16e    <= in_I16_i2;
        in_I18e    <= in_I18_i2;
        opcode_op  <= NOP;
        ra_addr_op <= 'dx;
        rb_addr_op <= 'dx;
        rc_addr_op <= 'dx;
        rt_addr_op <= 'dx;
        in_I7o     <= 'dx;
        in_I8o     <= 'dx;
        in_I10o    <= 'dx;
        in_I16o    <= 'dx;
        in_I18o    <= 'dx;
        pc_out <= pc_in;
    end
    else begin
        opcode_op  <= opcode_i2;
        ra_addr_op <= ra_addr_i2;
        rb_addr_op <= rb_addr_i2;
        rc_addr_op <= rc_addr_i2;
        rt_addr_op <= rt_addr_i2;
        in_I7o     <= in_I7_i2;
        in_I8o     <= in_I8_i2;
        in_I10o    <= in_I10_i2;
        in_I16o    <= in_I16_i2;
        in_I18o    <= in_I18_i2;
        opcode_ep  <= LNOP;
        ra_addr_ep <= 'dx;
        rb_addr_ep <= 'dx;
        rc_addr_ep <= 'dx;
        rt_addr_ep <= 'dx;
        in_I7e     <= 'dx;
        in_I8e     <= 'dx;
        in_I10e    <= 'dx;
        in_I16e    <= 'dx;
```

```
            in_I18e    <= 'dx;
            pc_out     <= pc_in + 32'd4;
        end
        stall_done <= (dep_stall == 1) ? 1'b1 : 1'b0;
    end
    else begin
        if(ins1_type == EVEN) begin
            opcode_ep  <= opcode_i1;
            ra_addr_ep <= ra_addr_i1;
            rb_addr_ep <= rb_addr_i1;
            rc_addr_ep <= rc_addr_i1;
            rt_addr_ep <= rt_addr_i1;
            in_I7e     <= in_I7_i1;
            in_I8e     <= in_I8_i1;
            in_I10e    <= in_I10_i1;
            in_I16e    <= in_I16_i1;
            in_I18e    <= in_I18_i1;
            opcode_op  <= NOP;
            ra_addr_op <= 'dx;
            rb_addr_op <= 'dx;
            rc_addr_op <= 'dx;
            rt_addr_op <= 'dx;
            in_I7o     <= 'dx;
            in_I8o     <= 'dx;
            in_I10o    <= 'dx;
            in_I16o    <= 'dx;
            in_I18o    <= 'dx;
            pc_out <= pc_in;
        end
        else begin
            opcode_op  <= opcode_i1;
            ra_addr_op <= ra_addr_i1;
            rb_addr_op <= rb_addr_i1;
            rc_addr_op <= rc_addr_i1;
            rt_addr_op <= rt_addr_i1;
            in_I7o     <= in_I7_i1;
            in_I8o     <= in_I8_i1;
            in_I10o    <= in_I10_i1;
            in_I16o    <= in_I16_i1;
            in_I18o    <= in_I18_i1;
            opcode_ep  <= LNOP;
            ra_addr_ep <= 'dx;
            rb_addr_ep <= 'dx;
            rc_addr_ep <= 'dx;
            rt_addr_ep <= 'dx;
            in_l7e     <= 'dx;
            in_l8e     <= 'dx;
            in_l10e    <= 'dx;
            in_l16e    <= 'dx;
            in_l18e    <= 'dx;
            pc_out     <= pc_in;
        end
        stall_done <= (dep_stall == 1) ? 1'b0 : 1'b1;
    end
end
else begin
    if(ins1_type == EVEN) begin
        opcode_ep  <= opcode_i1;
        ra_addr_ep <= ra_addr_i1;
        rb_addr_ep <= rb_addr_i1;
        rc_addr_ep <= rc_addr_i1;
        rt_addr_ep <= rt_addr_i1;
        in_l7e     <= in_l7_i1;
        in_l8e     <= in_l8_i1;
        in_l10e    <= in_l10_i1;
        in_l16e    <= in_l16_i1;
        in_l18e    <= in_l18_i1;
        opcode_op  <= opcode_i2;
        ra_addr_op <= ra_addr_i2;
        rb_addr_op <= rb_addr_i2;
        rc_addr_op <= rc_addr_i2;
        rt_addr_op <= rt_addr_i2;
        in_l7o     <= in_l7_i2;
        in_l8o     <= in_l8_i2;
        in_l10o    <= in_l10_i2;
        in_l16o    <= in_l16_i2;
        in_l18o    <= in_l18_i2;
        pc_out     <= pc_in + 32'd4;
    end
    else begin
        opcode_ep  <= opcode_i2;
        ra_addr_ep <= ra_addr_i2;
        rb_addr_ep <= rb_addr_i2;
        rc_addr_ep <= rc_addr_i2;
        rt_addr_ep <= rt_addr_i2;
        in_l7e     <= in_l7_i2;
        in_l8e     <= in_l8_i2;
        in_l10e    <= in_l10_i2;
```

```
            in_l16e    <= in_l16_i2;
            in_l18e    <= in_l18_i2;
            opcode_op  <= opcode_i1;
            ra_addr_op <= ra_addr_i1;
            rb_addr_op <= rb_addr_i1;
            rc_addr_op <= rc_addr_i1;
            rt_addr_op <= rt_addr_i1;
            in_l7o     <= in_l7_i1;
            in_l8o     <= in_l8_i1;
            in_l10o    <= in_l10_i1;
            in_l16o    <= in_l16_i1;
            in_l18o    <= in_l18_i1;
            pc_out     <= pc_in;
         end
      end
   end
end

always_comb begin
   opcode_i1  = LNOP;
   opcode_i2  = NOP;
   ins1_type  = EVEN;
   ins2_type  = ODD;
   ra_addr_i1 = 'dx;
   rb_addr_i1 = 'dx;
   rc_addr_i1 = 'dx;
   rt_addr_i1 = 'dx;
   ra_addr_i2 = 'dx;
   rb_addr_i2 = 'dx;
   rc_addr_i2 = 'dx;
   rt_addr_i2 = 'dx;
   in_l7_i1   = 'dx;
   in_l8_i1   = 'dx;
   in_l10_i1  = 'dx;
   in_l16_i1  = 'dx;
   in_l18_i1  = 'dx;
   in_l7_i2   = 'dx;
   in_l8_i2   = 'dx;
   in_l10_i2  = 'dx;
   in_l16_i2  = 'dx;
   in_l18_i2  = 'dx;

   if(eins1 == 32'hffffffff) begin
      opcode_i1 = CMISS;
```

```
    ins1_type = ODD;
end
else if(eoc1_4b == 4'b1100 || eoc1_4b == 4'b1110 || eoc1_4b == 4'b1111) begin
    ins1_type  = EVEN;
    rt_addr_i1 = eins1[4:10];
    ra_addr_i1 = eins1[11:17];
    rb_addr_i1 = eins1[18:24];
    rc_addr_i1 = eins1[25:31];
    case(eoc1_4b)
        4'b1100:
            begin
                opcode_i1 = MULTIPLY_AND_ADD;
            end
        4'b1110:
            begin
                opcode_i1 = FLOATING_MULTIPLY_AND_ADD;
            end
        4'b1111:
            begin
                opcode_i1 = FLOATING_MULTIPLY_AND_SUBTRACT;
            end
    endcase
end
else if(eoc1_7b == 7'b0100001) begin
    ins1_type = EVEN;
    rt_addr_i1 = eins1[25:31];
    in_I18_i1  = eins1[7:24];
    opcode_i1  = IMMEDIATE_LOAD_ADDRESS;
end
else if(eoc1_8b == 8'b00011101 || eoc1_8b == 8'b00011100 || eoc1_8b == 8'b00001101 || eoc1_8b == 8'b00001100 ||
        eoc1_8b == 8'b00010100 || eoc1_8b == 8'b00010101 || eoc1_8b == 8'b00010110 || eoc1_8b == 8'b00000100 ||
        eoc1_8b == 8'b00000101 || eoc1_8b == 8'b00000110 || eoc1_8b == 8'b01000100 || eoc1_8b == 8'b01000110 ||
        eoc1_8b == 8'b01111100 || eoc1_8b == 8'b01111110 || eoc1_8b == 8'b01001100 || eoc1_8b == 8'b01001101 ||
        eoc1_8b == 8'b00110100 || eoc1_8b == 8'b00100100 || eoc1_8b == 8'b01110100 || eoc1_8b == 8'b01000101) begin
    rt_addr_i1 = eins1[25:31];
    ra_addr_i1 = eins1[18:24];
    in_I10_i1  = eins1[8:17];
    case(eoc1_8b)
        8'b00011101:
            begin
                ins1_type = EVEN;
                opcode_i1  = ADD_HALF_WORD_IMMEDIATE;
            end
        8'b00011100:
```

```
   begin
      ins1_type = EVEN;
      opcode_i1  = ADD_WORD_IMMEDIATE;
   end
8'b00001101:
   begin
      ins1_type = EVEN;
      opcode_i1  = SUBTRACT_FROM_HALFWORD_IMMEDIATE;
   end
8'b00001100:
   begin
      ins1_type = EVEN;
      opcode_i1  = SUBTRACT_FROM_WORD_IMMEDIATE;
   end
8'b00010100:
   begin
      ins1_type = EVEN;
      opcode_i1  = AND_WORD_IMMEDIATE;
   end
8'b00010101:
   begin
      ins1_type = EVEN;
      opcode_i1  = AND_HALFWORD_IMMEDIATE;
   end
8'b00010110:
   begin
      ins1_type = EVEN;
      opcode_i1  = AND_BYTE_IMMEDIATE;
   end
8'b00000100:
   begin
      ins1_type = EVEN;
      opcode_i1  = OR_WORD_IMMEDIATE;
   end
8'b00000101:
   begin
      ins1_type = EVEN;
      opcode_i1  = OR_HALFWORD_IMMEDIATE;
   end
8'b00000110:
   begin
      ins1_type = EVEN;
      opcode_i1  = OR_BYTE_IMMEDIATE;
   end
```

```
8'b01000100:
   begin
      ins1_type = EVEN;
      opcode_i1  = EXCLUSIVE_OR_WORD_IMMEDIATE;
   end
8'b01000101:
   begin
      ins1_type = EVEN;
      opcode_i1  = EXCLUSIVE_OR_HALFWORD_IMMEDIATE;
   end
8'b01000110:
   begin
      ins1_type = EVEN;
      opcode_i1  = EXCLUSIVE_OR_BYTE_IMMEDIATE;
   end
8'b01111100:
   begin
      ins1_type = EVEN;
      opcode_i1  = COMPARE_EQUAL_WORD_IMMEDIATE;
   end
8'b01111110:
   begin
      ins1_type = EVEN;
      opcode_i1  = COMPARE_EQUAL_HALFWORD_IMMEDIATE;
   end
8'b01001100:
   begin
      ins1_type = EVEN;
      opcode_i1  = COMPARE_GREATER_THAN_WORD_IMMEDIATE;
   end
8'b01001101:
   begin
      ins1_type = EVEN;
      opcode_i1  = COMPARE_GREATER_THAN_HALFWORD_IMMEDIATE;
   end
8'b00110100:
   begin
      ins1_type = ODD;
      opcode_i1  = LOAD_QUADWORD_DFORM;
   end
8'b00100100:
   begin
      ins1_type  = ODD;
      opcode_i1  = STORE_QUADWORD_DFORM;
```

```
            rc_addr_i1 = eins1[25:31];
          end
        8'b01110100:
          begin
            ins1_type = EVEN;
            opcode_i1  = MULTIPLY_IMMEDIATE;
          end
    endcase
  end
  else if(eoc1_9b == 9'b010000011 || eoc1_9b == 9'b010000001 || eoc1_9b == 9'b001100001 ||
        eoc1_9b == 9'b001000001 || eoc1_9b == 9'b001100110 || eoc1_9b == 9'b001100010 ||
        eoc1_9b == 9'b001100100 || eoc1_9b == 9'b001100000 || eoc1_9b == 9'b001000010 ||
        eoc1_9b == 9'b001000000 || eoc1_9b == 9'b001000110 || eoc1_9b == 9'b001000100) begin
    in_I16_i1  = eins1[9:24];
    rt_addr_i1 = eins1[25:31];
    case(eoc1_9b)
        9'b010000011:
          begin
            ins1_type = EVEN;
            opcode_i1 = IMMEDIATE_LOAD_HALFWORD;
          end
        9'b010000001:
          begin
            ins1_type = EVEN;
            opcode_i1 = IMMEDIATE_LOAD_WORD;
          end
        9'b001100001:
          begin
            ins1_type = ODD;
            opcode_i1 = LOAD_QUADWORD_AFORM;
          end
        9'b001000001:
          begin
            ins1_type = ODD;
            opcode_i1 = STORE_QUADWORD_AFORM;
          end
        9'b001100110:
          begin
            ins1_type = ODD;
            opcode_i1 = BRANCH_RELATIVE_AND_SET_LINK;
          end
        9'b001100010:
          begin
            ins1_type = ODD;
            opcode_i1 = BRANCH_ABSOLUTE_AND_SET_LINK;
         end
      9'b001100100:
         begin
            ins1_type = ODD;
            opcode_i1 = BRANCH_RELATIVE;
         end
      9'b001100000:
         begin
            ins1_type = ODD;
            opcode_i1 = BRANCH_ABSOLUTE;
         end
      9'b001000010:
         begin
            ins1_type = ODD;
            opcode_i1 = BRANCH_IF_NOT_ZERO_WORD;
         end
      9'b001000000:
         begin
            ins1_type = ODD;
            opcode_i1 = BRANCH_IF_ZERO_WORD;
         end
      9'b001000110:
         begin
            ins1_type = ODD;
            opcode_i1 = BRANCH_IF_NOT_ZERO_HALFWORD;
         end
      9'b001000100:
         begin
            ins1_type = ODD;
            opcode_i1 = BRANCH_IF_ZERO_HALFWORD;
         end
   endcase
end
else begin
   rt_addr_i1 = eins1[25:31];
   ra_addr_i1 = eins1[18:24];
   rb_addr_i1 = eins1[11:17];
   case(eoc1_11b)
      11'b00011001000:
         begin
            ins1_type = EVEN;
            opcode_i1 = ADD_HALF_WORD;
         end
```

```
11'b00011000000:
   begin
      ins1_type = EVEN;
      opcode_i1 = ADD_WORD;
   end
11'b00001001000:
   begin
      ins1_type = EVEN;
      opcode_i1 = SUBTRACT_FROM_HALFWORD;
   end
11'b00001000000:
   begin
      ins1_type = EVEN;
      opcode_i1 = SUBTRACT_FROM_WORD;
   end
11'b01101000000:
   begin
      ins1_type = EVEN;
      opcode_i1 = ADD_EXTENDED;
   end
11'b01101000001:
   begin
      ins1_type = EVEN;
      opcode_i1 = SUBTRACT_FROM_EXTENDED;
   end
11'b00011000010:
   begin
      ins1_type = EVEN;
      opcode_i1 = CARRY_GENERATE;
   end
11'b00001000010:
   begin
      ins1_type = EVEN;
      opcode_i1 = BORROW_GENERATE;
   end
11'b01010100101:
   begin
      ins1_type  = EVEN;
      opcode_i1  = COUNT_LEADING_ZEROS;
      rb_addr_i1 = 'dx;
   end
11'b00011000001:
   begin
      ins1_type = EVEN;
```

```
        opcode_i1 = AND;
    end
11'b00001000001:
    begin
        ins1_type = EVEN;
        opcode_i1 = OR;
    end
11'b01001000001:
    begin
        ins1_type = EVEN;
        opcode_i1 = EXCLUSIVE_OR;
    end
11'b00011001001:
    begin
        ins1_type = EVEN;
        opcode_i1 = NAND;
    end
11'b00001001001:
    begin
        ins1_type = EVEN;
        opcode_i1 = NOR;
    end
11'b01001001001:
    begin
        ins1_type = EVEN;
        opcode_i1 = EQUIVALENT;
    end
11'b01111000000:
    begin
        ins1_type = EVEN;
        opcode_i1 = COMPARE_EQUAL_WORD;
    end
11'b01111001000:
    begin
        ins1_type = EVEN;
        opcode_i1 = COMPARE_EQUAL_HALFWORD;
    end
11'b01001000000:
    begin
        ins1_type = EVEN;
        opcode_i1 = COMPARE_GREATER_THAN_WORD;
    end
11'b01001001000:
    begin
```

```
        ins1_type = EVEN;
        opcode_i1 = COMPARE_GREATER_THAN_HALFWORD;
    end
11'b00001011111:
    begin
        ins1_type = EVEN;
        opcode_i1 = SHIFT_LEFT_HALFWORD;
    end
11'b00001111111:
    begin
        ins1_type = EVEN;
        opcode_i1 = SHIFT_LEFT_HALFWORD_IMMEDIATE;
        in_I7_i1  = eins1[11:17];
        rb_addr_i1 = 'dx;
    end
11'b00001011011:
    begin
        ins1_type = EVEN;
        opcode_i1 = SHIFT_LEFT_WORD;
    end
11'b00001111011:
    begin
        ins1_type = EVEN;
        opcode_i1 = SHIFT_LEFT_WORD_IMMEDIATE;
        in_I7_i1  = eins1[11:17];
        rb_addr_i1 = 'dx;
    end
11'b00111011011:
    begin
        ins1_type = ODD;
        opcode_i1 = SHIFT_LEFT_QUADWORD_BY_BITS;
    end
11'b00111111011:
    begin
        ins1_type = ODD;
        opcode_i1 = SHIFT_LEFT_QUADWORD_BY_BITS_IMMEDIATE;
        in_I7_i1  = eins1[11:17];
        rb_addr_i1 = 'dx;
    end
11'b00111011111:
    begin
        ins1_type = ODD;
        opcode_i1 = SHIFT_LEFT_QUADWORD_BY_BYTES;
    end
```

```
11'b00111111111:
   begin
      ins1_type = ODD;
      opcode_i1 = SHIFT_LEFT_QUADWORD_BY_BYTES_IMMEDIATE;
      in_I7_i1  = eins1[11:17];
      rb_addr_i1 = 'dx;
   end
11'b00001011000:
   begin
      ins1_type = EVEN;
      opcode_i1 = ROTATE_WORD;
   end
11'b00001111100:
   begin
      ins1_type = EVEN;
      opcode_i1 = ROTATE_WORD_IMMEDIATE;
      in_I7_i1  = eins1[11:17];
      rb_addr_i1 = 'dx;
   end
11'b00111011000:
   begin
      ins1_type = ODD;
      opcode_i1 = ROTATE_QUADWORD_BY_BITS;
   end
11'b00111111000:
   begin
      ins1_type = ODD;
      opcode_i1 = ROTATE_QUADWORD_BY_BITS_IMMEDIATE;
      in_I7_i1  = eins1[11:17];
      rb_addr_i1 = 'dx;
   end
11'b00111011100:
   begin
      ins1_type = ODD;
      opcode_i1 = ROTATE_QUADWORD_BY_BYTES;
   end
11'b00111111100:
   begin
      ins1_type = ODD;
      opcode_i1 = ROTATE_QUADWORD_BY_BYTES_IMMEDIATE;
      in_I7_i1  = eins1[11:17];
      rb_addr_i1 = 'dx;
   end
11'b00110101000:
   begin
      ins1_type = ODD;
      opcode_i1 = BRANCH_INDIRECT;
      ra_addr_i1 = eins1[18:24];
      rb_addr_i1 = 'dx;
      rt_addr_i1 = 'dx;
   end
11'b01011000100:
   begin
      ins1_type = EVEN;
      opcode_i1 = FLOATING_ADD;
   end
11'b01011000101:
   begin
      ins1_type = EVEN;
      opcode_i1 = FLOATING_SUBTRACT;
   end
11'b01011000110:
   begin
      ins1_type = EVEN;
      opcode_i1 = FLOATING_MULTIPLY;
   end
11'b01111000010:
   begin
      ins1_type = EVEN;
      opcode_i1 = FLOATING_COMPARE_EQUAL;
   end
11'b01111001010:
   begin
      ins1_type = EVEN;
      opcode_i1 = FLOATING_COMPARE_MAGNITUDE_EQUAL;
   end
11'b01011000010:
   begin
      ins1_type = EVEN;
      opcode_i1 = FLOATING_COMPARE_GREATER_THAN;
   end
11'b01011001010:
   begin
      ins1_type = EVEN;
      opcode_i1 = FLOATING_COMPARE_MAGNITUDE_GREATER_THAN;
   end
11'b01111000100:
   begin
```

```
        ins1_type = EVEN;
        opcode_i1 = MULTIPLY;
    end
11'b00000000000:
    begin
        ins1_type = EVEN;
        opcode_i1 = STOP;
        ra_addr_i1 = 'dx;
        rb_addr_i1 = 'dx;
        rt_addr_i1 = 'dx;
    end
11'b01010110100:
    begin
        ins1_type = EVEN;
        opcode_i1 = COUNT_ONES_IN_BYTES;
    end
11'b00011010011:
    begin
        ins1_type = EVEN;
        opcode_i1 = AVERAGE_BYTES;
    end
11'b00001010011:
    begin
        ins1_type = EVEN;
        opcode_i1 = ABSOLUTE_DIFFERENCE_OF_BYTES;
    end
11'b01001010011:
    begin
        ins1_type = EVEN;
        opcode_i1 = SUM_BYTES_INTO_HALFWORDS;
    end
11'b00000000001:
    begin
        ins1_type = EVEN;
        opcode_i1 = LNOP;
        ra_addr_i1 = 'dx;
        rb_addr_i1 = 'dx;
        rc_addr_i1 = 'dx;
        rt_addr_i1 = 'dx;
    end
11'b01000000001:
    begin
        ins1_type = ODD;
        opcode_i1 = NOP;
```

```
                ra_addr_i1 = 'dx;
                rb_addr_i1 = 'dx;
                rc_addr_i1 = 'dx;
                rt_addr_i1 = 'dx;
            end
    endcase
end

if(eins2 == 32'hffffffff) begin
    opcode_i2 = LNOP;
    ins2_type = EVEN;
end
else if(eoc2_4b == 4'b1100 || eoc2_4b == 4'b1110 || eoc2_4b == 4'b1111) begin
    ins2_type  = EVEN;
    rt_addr_i2 = eins2[4:10];
    ra_addr_i2 = eins2[11:17];
    rb_addr_i2 = eins2[18:24];
    rc_addr_i2 = eins2[25:31];
    case(eoc2_4b)
        4'b1100:
            begin
                opcode_i2 = MULTIPLY_AND_ADD;
            end
        4'b1110:
            begin
                opcode_i2 = FLOATING_MULTIPLY_AND_ADD;
            end
        4'b1111:
            begin
                opcode_i2 = FLOATING_MULTIPLY_AND_SUBTRACT;
            end
    endcase
end
else if(eoc2_7b == 7'b0100001) begin
    ins2_type = EVEN;
    rt_addr_i2 = eins2[25:31];
    in_I18_i2  = eins2[7:24];
    opcode_i2  = IMMEDIATE_LOAD_ADDRESS;
end
else if(eoc2_8b == 8'b00011101 || eoc2_8b == 8'b00011100 || eoc2_8b == 8'b00001101 || eoc2_8b == 8'b00001100 ||
        eoc2_8b == 8'b00010100 || eoc2_8b == 8'b00010101 || eoc2_8b == 8'b00010110 || eoc2_8b == 8'b00000100 ||
        eoc2_8b == 8'b00000101 || eoc2_8b == 8'b00000110 || eoc2_8b == 8'b01000100 || eoc2_8b == 8'b01000110 ||
        eoc2_8b == 8'b01111100 || eoc2_8b == 8'b01111110 || eoc2_8b == 8'b01001100 || eoc2_8b == 8'b01001101 ||
        eoc2_8b == 8'b00110100 || eoc2_8b == 8'b00100100 || eoc2_8b == 8'b01110100 || eoc2_8b == 8'b01000101) begin
```

```
rt_addr_i2 = eins2[25:31];
ra_addr_i2 = eins2[18:24];
in_I10_i2  = eins2[8:17];
case(eoc2_8b)
  8'b00011101:
    begin
      ins2_type = EVEN;
      opcode_i2  = ADD_HALF_WORD_IMMEDIATE;
    end
  8'b00011100:
    begin
      ins2_type = EVEN;
      opcode_i2  = ADD_WORD_IMMEDIATE;
    end
  8'b00001101:
    begin
      ins2_type = EVEN;
      opcode_i2  = SUBTRACT_FROM_HALFWORD_IMMEDIATE;
    end
  8'b00001100:
    begin
      ins2_type = EVEN;
      opcode_i2  = SUBTRACT_FROM_WORD_IMMEDIATE;
    end
  8'b00010100:
    begin
      ins2_type = EVEN;
      opcode_i2  = AND_WORD_IMMEDIATE;
    end
  8'b00010101:
    begin
      ins2_type = EVEN;
      opcode_i2  = AND_HALFWORD_IMMEDIATE;
    end
  8'b00010110:
    begin
      ins2_type = EVEN;
      opcode_i2  = AND_BYTE_IMMEDIATE;
    end
  8'b00000100:
    begin
      ins2_type = EVEN;
      opcode_i2  = OR_WORD_IMMEDIATE;
    end
```

```
8'b00000101:
  begin
    ins2_type = EVEN;
    opcode_i2  = OR_HALFWORD_IMMEDIATE;
  end
8'b00000110:
  begin
    ins2_type = EVEN;
    opcode_i2  = OR_BYTE_IMMEDIATE;
  end
8'b01000100:
  begin
    ins2_type = EVEN;
    opcode_i2  = EXCLUSIVE_OR_WORD_IMMEDIATE;
  end
8'b01000101:
  begin
    ins2_type = EVEN;
    opcode_i2  = EXCLUSIVE_OR_HALFWORD_IMMEDIATE;
  end
8'b01000110:
  begin
    ins2_type = EVEN;
    opcode_i2  = EXCLUSIVE_OR_BYTE_IMMEDIATE;
  end
8'b01111100:
  begin
    ins2_type = EVEN;
    opcode_i2  = COMPARE_EQUAL_WORD_IMMEDIATE;
  end
8'b01111110:
  begin
    ins2_type = EVEN;
    opcode_i2  = COMPARE_EQUAL_HALFWORD_IMMEDIATE;
  end
8'b01001100:
  begin
    ins2_type = EVEN;
    opcode_i2  = COMPARE_GREATER_THAN_WORD_IMMEDIATE;
  end
8'b01001101:
  begin
    ins2_type = EVEN;
    opcode_i2  = COMPARE_GREATER_THAN_HALFWORD_IMMEDIATE;
```

```
            end
        8'b00110100:
            begin
                ins2_type = ODD;
                opcode_i2  = LOAD_QUADWORD_DFORM;
            end
        8'b00100100:
            begin
                ins2_type = ODD;
                opcode_i2  = STORE_QUADWORD_DFORM;
                rc_addr_i2 = eins2[25:31];
            end
        8'b01110100:
            begin
                ins2_type = EVEN;
                opcode_i2  = MULTIPLY_IMMEDIATE;
            end
    endcase
end
else if(eoc2_9b == 9'b010000011 || eoc2_9b == 9'b010000001 || eoc2_9b == 9'b001100001 ||
        eoc2_9b == 9'b001000001 || eoc2_9b == 9'b001100110 || eoc2_9b == 9'b001100010 ||
        eoc2_9b == 9'b001100100 || eoc2_9b == 9'b001100000 || eoc2_9b == 9'b001000010 ||
        eoc2_9b == 9'b001000000 || eoc2_9b == 9'b001000110 || eoc2_9b == 9'b001000100) begin
    in_I16_i2  = eins2[9:24];
    rt_addr_i2 = eins2[25:31];
    case(eoc2_9b)
        9'b010000011:
            begin
                ins2_type = EVEN;
                opcode_i2 = IMMEDIATE_LOAD_HALFWORD;
            end
        9'b010000001:
            begin
                ins2_type = EVEN;
                opcode_i2 = IMMEDIATE_LOAD_WORD;
            end
        9'b001100001:
            begin
                ins2_type = ODD;
                opcode_i2 = LOAD_QUADWORD_AFORM;
            end
        9'b001000001:
            begin
                ins2_type = ODD;
            opcode_i2 = STORE_QUADWORD_AFORM;
        end
    9'b001100110:
        begin
            ins2_type = ODD;
            opcode_i2 = BRANCH_RELATIVE_AND_SET_LINK;
        end
    9'b001100010:
        begin
            ins2_type = ODD;
            opcode_i2 = BRANCH_ABSOLUTE_AND_SET_LINK;
        end
    9'b001100100:
        begin
            ins2_type = ODD;
            opcode_i2 = BRANCH_RELATIVE;
        end
    9'b001100000:
        begin
            ins2_type = ODD;
            opcode_i2 = BRANCH_ABSOLUTE;
        end
    9'b001000010:
        begin
            ins2_type = ODD;
            opcode_i2 = BRANCH_IF_NOT_ZERO_WORD;
        end
    9'b001000000:
        begin
            ins2_type = ODD;
            opcode_i2 = BRANCH_IF_ZERO_WORD;
        end
    9'b001000110:
        begin
            ins2_type = ODD;
            opcode_i2 = BRANCH_IF_NOT_ZERO_HALFWORD;
        end
    9'b001000100:
        begin
            ins2_type = ODD;
            opcode_i2 = BRANCH_IF_ZERO_HALFWORD;
        end
  endcase
end
```

```
else begin
   rt_addr_i2 = eins2[25:31];
   ra_addr_i2 = eins2[18:24];
   rb_addr_i2 = eins2[11:17];
   case(eoc2_11b)
      11'b00011001000:
         begin
            ins2_type = EVEN;
            opcode_i2 = ADD_HALF_WORD;
         end
      11'b00011000000:
         begin
            ins2_type = EVEN;
            opcode_i2 = ADD_WORD;
         end
      11'b00001001000:
         begin
            ins2_type = EVEN;
            opcode_i2 = SUBTRACT_FROM_HALFWORD;
         end
      11'b00001000000:
         begin
            ins2_type = EVEN;
            opcode_i2 = SUBTRACT_FROM_WORD;
         end
      11'b01101000000:
         begin
            ins2_type = EVEN;
            opcode_i2 = ADD_EXTENDED;
         end
      11'b01101000001:
         begin
            ins2_type = EVEN;
            opcode_i2 = SUBTRACT_FROM_EXTENDED;
         end
      11'b00011000010:
         begin
            ins2_type = EVEN;
            opcode_i2 = CARRY_GENERATE;
         end
      11'b00001000010:
         begin
            ins2_type = EVEN;
            opcode_i2 = BORROW_GENERATE;
```

```
  end
11'b01010100101:
  begin
    ins2_type  = EVEN;
    opcode_i2  = COUNT_LEADING_ZEROS;
    rb_addr_i2 = 'dx;
  end
11'b00011000001:
  begin
    ins2_type = EVEN;
    opcode_i2 = AND;
  end
11'b00001000001:
  begin
    ins2_type = EVEN;
    opcode_i2 = OR;
  end
11'b01001000001:
  begin
    ins2_type = EVEN;
    opcode_i2 = EXCLUSIVE_OR;
  end
11'b00011001001:
  begin
    ins2_type = EVEN;
    opcode_i2 = NAND;
  end
11'b00001001001:
  begin
    ins2_type = EVEN;
    opcode_i2 = NOR;
  end
11'b01001001001:
  begin
    ins2_type = EVEN;
    opcode_i2 = EQUIVALENT;
  end
11'b01111000000:
  begin
    ins2_type = EVEN;
    opcode_i2 = COMPARE_EQUAL_WORD;
  end
11'b01111001000:
  begin
```

```
        ins2_type = EVEN;
        opcode_i2 = COMPARE_EQUAL_HALFWORD;
    end
11'b01001000000:
    begin
        ins2_type = EVEN;
        opcode_i2 = COMPARE_GREATER_THAN_WORD;
    end
11'b01001001000:
    begin
        ins2_type = EVEN;
        opcode_i2 = COMPARE_GREATER_THAN_HALFWORD;
    end
11'b00001011111:
    begin
        ins2_type = EVEN;
        opcode_i2 = SHIFT_LEFT_HALFWORD;
    end
11'b00001111111:
    begin
        ins2_type = EVEN;
        opcode_i2 = SHIFT_LEFT_HALFWORD_IMMEDIATE;
        in_I7_i2  = eins2[11:17];
        rb_addr_i2 = 'dx;
    end
11'b00001011011:
    begin
        ins2_type = EVEN;
        opcode_i2 = SHIFT_LEFT_WORD;
    end
11'b00001111011:
    begin
        ins2_type = EVEN;
        opcode_i2 = SHIFT_LEFT_WORD_IMMEDIATE;
        in_I7_i2  = eins2[11:17];
        rb_addr_i2 = 'dx;
    end
11'b00111011011:
    begin
        ins2_type = ODD;
        opcode_i2 = SHIFT_LEFT_QUADWORD_BY_BITS;
    end
11'b00111111011:
    begin
```

```
        ins2_type = ODD;
        opcode_i2 = SHIFT_LEFT_QUADWORD_BY_BITS_IMMEDIATE;
        in_I7_i2  = eins2[11:17];
        rb_addr_i2 = 'dx;
    end
11'b00111011111:
    begin
        ins2_type = ODD;
        opcode_i2 = SHIFT_LEFT_QUADWORD_BY_BYTES;
    end
11'b00111111111:
    begin
        ins2_type = ODD;
        opcode_i2 = SHIFT_LEFT_QUADWORD_BY_BYTES_IMMEDIATE;
        in_I7_i2  = eins2[11:17];
        rb_addr_i2 = 'dx;
    end
11'b00001011000:
    begin
        ins2_type = EVEN;
        opcode_i2 = ROTATE_WORD;
    end
11'b00001111100:
    begin
        ins2_type = EVEN;
        opcode_i2 = ROTATE_WORD_IMMEDIATE;
        in_I7_i2  = eins2[11:17];
        rb_addr_i2 = 'dx;
    end
11'b00111011000:
    begin
        ins2_type = ODD;
        opcode_i2 = ROTATE_QUADWORD_BY_BITS;
    end
11'b00111111000:
    begin
        ins2_type = ODD;
        opcode_i2 = ROTATE_QUADWORD_BY_BITS_IMMEDIATE;
        in_I7_i2  = eins2[11:17];
        rb_addr_i2 = 'dx;
    end
11'b00111011100:
    begin
        ins2_type = ODD;
        opcode_i2 = ROTATE_QUADWORD_BY_BYTES;
    end
11'b00111111100:
    begin
        ins2_type = ODD;
        opcode_i2 = ROTATE_QUADWORD_BY_BYTES_IMMEDIATE;
        in_I7_i2  = eins2[11:17];
        rb_addr_i2 = 'dx;
    end
11'b00110101000:
    begin
        ins2_type = ODD;
        opcode_i2 = BRANCH_INDIRECT;
        ra_addr_i2 = eins2[18:24];
        rb_addr_i2 = 'dx;
        rt_addr_i2 = 'dx;
    end
11'b01011000100:
    begin
        ins2_type = EVEN;
        opcode_i2 = FLOATING_ADD;
    end
11'b01011000101:
    begin
        ins2_type = EVEN;
        opcode_i2 = FLOATING_SUBTRACT;
    end
11'b01011000110:
    begin
        ins2_type = EVEN;
        opcode_i2 = FLOATING_MULTIPLY;
    end
11'b01111000010:
    begin
        ins2_type = EVEN;
        opcode_i2 = FLOATING_COMPARE_EQUAL;
    end
11'b01111001010:
    begin
        ins2_type = EVEN;
        opcode_i2 = FLOATING_COMPARE_MAGNITUDE_EQUAL;
    end
11'b01011000010:
    begin
```

```
            ins2_type = EVEN;
            opcode_i2 = FLOATING_COMPARE_GREATER_THAN;
        end
    11'b01011001010:
        begin
            ins2_type = EVEN;
            opcode_i2 = FLOATING_COMPARE_MAGNITUDE_GREATER_THAN;
        end
    11'b01111000100:
        begin
            ins2_type = EVEN;
            opcode_i2 = MULTIPLY;
        end
    11'b00000000000:
        begin
            ins2_type = EVEN;
            opcode_i2 = STOP;
        end
    11'b01010110100:
        begin
            ins2_type = EVEN;
            opcode_i2 = COUNT_ONES_IN_BYTES;
        end
    11'b00011010011:
        begin
            ins2_type = EVEN;
            opcode_i2 = AVERAGE_BYTES;
        end
    11'b00001010011:
        begin
            ins2_type = EVEN;
            opcode_i2 = ABSOLUTE_DIFFERENCE_OF_BYTES;
        end
    11'b01001010011:
        begin
            ins2_type = EVEN;
            opcode_i2 = SUM_BYTES_INTO_HALFWORDS;
        end
    11'b00000000001:
        begin
            ins2_type = EVEN;
            opcode_i2 = LNOP;
            ra_addr_i2 = 'dx;
            rb_addr_i2 = 'dx;
```

```
                    rc_addr_i2 = 'dx;
                    rt_addr_i2 = 'dx;
                end
            11'b01000000001:
                begin
                    ins2_type = ODD;
                    opcode_i2 = NOP;
                    ra_addr_i2 = 'dx;
                    rb_addr_i2 = 'dx;
                    rc_addr_i2 = 'dx;
                    rt_addr_i2 = 'dx;
                end
        endcase
    end

    if(stall_done || flush) begin
        dec_stall = 0;
    end
    else if((ins1_type == EVEN && ins2_type == EVEN) ||
            (ins1_type == ODD  && ins2_type == ODD))
    begin
        dec_stall = 1;
    end
    else if(((rt_addr_i1 == rt_addr_i2) && rt_addr_i1 !== 7'dx && rt_addr_i2 !== 7'dx) ||
            (rt_addr_i1 == ra_addr_i2 || rt_addr_i2 == ra_addr_i1) ||
            (rt_addr_i1 == rb_addr_i2 || rt_addr_i2 == rb_addr_i1) ||
            (rt_addr_i1 == rc_addr_i2 || rt_addr_i2 == rc_addr_i1)) begin

        dec_stall = 1;
    end
    else begin
        dec_stall = 0;
    end
  end

endmodule
//end of file.
```

## 5.5 Appendix E – Dependency (dependency.sv)

```
import defines_pkg::*;

module dependency
(
input  logic           clk,
input  logic           rst,
input  Opcodes          dec_opcode_ep,
input  Opcodes          dec_opcode_op,
input  logic [0:6]      dec_ra_addr_ep,
input  logic [0:6]      dec_rb_addr_ep,
input  logic [0:6]      dec_rc_addr_ep,
input  logic [0:6]      dec_rt_addr_ep,
input  logic [0:6]      dec_ra_addr_op,
input  logic [0:6]      dec_rb_addr_op,
input  logic [0:6]      dec_rc_addr_op,
input  logic [0:6]      dec_rt_addr_op,
input  logic [0:6]      dec_I7_ep,
input  logic [0:7]      dec_I8_ep,
input  logic [0:9]      dec_I10_ep,
input  logic [0:15]     dec_I16_ep,
input  logic [0:17]     dec_I18_ep,
input  logic [0:6]      dec_I7_op,
input  logic [0:7]      dec_I8_op,
input  logic [0:9]      dec_I10_op,
input  logic [0:15]     dec_I16_op,
input  logic [0:17]     dec_I18_op,
input  logic [0:6]      rf_addr_s1_ep,
input  logic [0:6]      rf_addr_s2_ep,
input  logic [0:6]      rf_addr_s3_ep,
input  logic [0:6]      rf_addr_s4_ep,
input  logic [0:6]      rf_addr_s5_ep,
input  logic [0:6]      rf_addr_s6_ep,
input  logic [0:6]      rf_addr_s1_op,
input  logic [0:6]      rf_addr_s2_op,
input  logic [0:6]      rf_addr_s3_op,
input  logic [0:6]      rf_addr_s4_op,
input  logic [0:6]      rf_addr_s5_op,
input  logic [0:6]      rf_addr_s6_op,
input  logic [0:2]      rf_idx_s1_ep,
input  logic [0:2]      rf_idx_s2_ep,
input  logic [0:2]      rf_idx_s3_ep,
input  logic [0:2]      rf_idx_s4_ep,
input  logic [0:2]      rf_idx_s5_ep,
input  logic [0:2]      rf_idx_s6_ep,
input  logic [0:2]      rf_idx_s1_op,
input  logic [0:2]      rf_idx_s2_op,
input  logic [0:2]      rf_idx_s3_op,
input  logic [0:2]      rf_idx_s4_op,
    input  logic [0:2]       rf_idx_s5_op,
    input  logic [0:2]       rf_idx_s6_op,
    input  logic             flush,
    input  logic [0:31]      pc_in,
    output logic [0:31]      pc_out,
    output Opcodes           opcode_ep,
    output Opcodes           opcode_op,
    output logic [0:6]       ra_addr_ep,
    output logic [0:6]       rb_addr_ep,
    output logic [0:6]       rc_addr_ep,
    output logic [0:6]       rt_addr_ep,
    output logic [0:6]       ra_addr_op,
    output logic [0:6]       rb_addr_op,
    output logic [0:6]       rc_addr_op,
    output logic [0:6]       rt_addr_op,
    output logic [0:6]       I7_ep,
    output logic [0:7]       I8_ep,
    output logic [0:9]       I10_ep,
    output logic [0:15]      I16_ep,
    output logic [0:17]      I18_ep,
    output logic [0:6]       I7_op,
    output logic [0:7]       I8_op,
    output logic [0:9]       I10_op,
    output logic [0:15]      I16_op,
    output logic [0:17]      I18_op,
    output logic             dep_stall
);

    logic is_lat1;
    logic is_lat2;
    logic is_lat3;
    logic is_lat4;
    logic is_lat5;
    logic is_lat6;

    assign is_lat1 = rf_idx_s1_ep == 3'd1 || rf_idx_s1_ep == 3'd2 || rf_idx_s1_ep == 3'd3 ||
                rf_idx_s1_ep == 3'd4 || rf_idx_s1_op == 3'd5 || rf_idx_s1_op == 3'd6 ||
                rf_idx_s1_ep == 3'd7;

    assign is_lat2 = rf_idx_s2_ep == 3'd2 || rf_idx_s2_ep == 3'd3 || rf_idx_s2_ep == 3'd4 ||
                rf_idx_s2_op == 3'd5 || rf_idx_s2_op == 3'd6 || rf_idx_s2_ep == 3'd7;

    assign is_lat3 = rf_idx_s3_ep == 3'd3 || rf_idx_s3_op == 3'd5 || rf_idx_s3_op == 3'd6 ||
                rf_idx_s3_ep == 3'd7;
```

```
assign is_lat4 = rf_idx_s4_ep == 3'd3 || rf_idx_s4_op == 3'd6 || rf_idx_s4_ep == 3'd7;

assign is_lat5 = rf_idx_s5_ep == 3'd3 || rf_idx_s5_op == 3'd6 || rf_idx_s5_ep == 3'd7;

assign is_lat6 = rf_idx_s6_ep == 3'd7;

always_ff @(posedge clk) begin
   if(rst) begin
      opcode_ep  <= LNOP;
      ra_addr_ep <= 'dx;
      rb_addr_ep <= 'dx;
      rc_addr_ep <= 'dx;
      rt_addr_ep <= 'dx;
      I7_ep      <= 'dx;
      I8_ep      <= 'dx;
      I10_ep     <= 'dx;
      I16_ep     <= 'dx;
      I18_ep     <= 'dx;
      opcode_op  <= NOP;
      ra_addr_op <= 'dx;
      rb_addr_op <= 'dx;
      rc_addr_op <= 'dx;
      rt_addr_op <= 'dx;
      I7_op      <= 'dx;
      I8_op      <= 'dx;
      I10_op     <= 'dx;
      I16_op     <= 'dx;
      I18_op     <= 'dx;
      pc_out     <= 'dx;
   end
   else begin
      if(flush || dep_stall) begin
         opcode_ep  <= LNOP;
         ra_addr_ep <= 'dx;
         rb_addr_ep <= 'dx;
         rc_addr_ep <= 'dx;
         rt_addr_ep <= 'dx;
         I7_ep      <= 'dx;
         I8_ep      <= 'dx;
         I10_ep     <= 'dx;
         I16_ep     <= 'dx;
         I18_ep     <= 'dx;
         opcode_op  <= NOP;
```

```
          ra_addr_op <= 'dx;
          rb_addr_op <= 'dx;
          rc_addr_op <= 'dx;
          rt_addr_op <= 'dx;
          I7_op      <= 'dx;
          I8_op      <= 'dx;
          I10_op     <= 'dx;
          I16_op     <= 'dx;
          I18_op     <= 'dx;
        end
        else begin
          opcode_ep  <= dec_opcode_ep;
          ra_addr_ep <= dec_ra_addr_ep;
          rb_addr_ep <= dec_rb_addr_ep;
          rc_addr_ep <= dec_rc_addr_ep;
          rt_addr_ep <= dec_rt_addr_ep;
          I7_ep      <= dec_I7_ep;
          I8_ep      <= dec_I8_ep;
          I10_ep     <= dec_I10_ep;
          I16_ep     <= dec_I16_ep;
          I18_ep     <= dec_I18_ep;
          opcode_op  <= dec_opcode_op;
          ra_addr_op <= dec_ra_addr_op;
          rb_addr_op <= dec_rb_addr_op;
          rc_addr_op <= dec_rc_addr_op;
          rt_addr_op <= dec_rt_addr_op;
          I7_op      <= dec_I7_op;
          I8_op      <= dec_I8_op;
          I10_op     <= dec_I10_op;
          I16_op     <= dec_I16_op;
          I18_op     <= dec_I18_op;
        end
        pc_out     <= pc_in;
      end
    end

    always_comb begin
      dep_stall = 1'b0;

      if(
        (rt_addr_ep == dec_ra_addr_ep || rt_addr_ep == dec_ra_addr_op) ||
        (rt_addr_ep == dec_rb_addr_ep || rt_addr_ep == dec_rb_addr_op) ||
        (rt_addr_ep == dec_rc_addr_ep || rt_addr_ep == dec_rc_addr_op) ||
        (rt_addr_op == dec_ra_addr_ep || rt_addr_op == dec_ra_addr_op) ||
        (rt_addr_op == dec_rb_addr_ep || rt_addr_op == dec_rb_addr_op) ||
        (rt_addr_op == dec_rc_addr_ep || rt_addr_op == dec_rc_addr_op) ||
        ((rf_addr_s1_ep == dec_ra_addr_ep || rf_addr_s1_ep == dec_ra_addr_op) && is_lat1) ||
        ((rf_addr_s2_ep == dec_ra_addr_ep || rf_addr_s2_ep == dec_ra_addr_op) && is_lat2) ||
        ((rf_addr_s3_ep == dec_ra_addr_ep || rf_addr_s3_ep == dec_ra_addr_op) && is_lat3) ||
        ((rf_addr_s4_ep == dec_ra_addr_ep || rf_addr_s4_ep == dec_ra_addr_op) && is_lat4) ||
        ((rf_addr_s5_ep == dec_ra_addr_ep || rf_addr_s5_ep == dec_ra_addr_op) && is_lat5) ||
        ((rf_addr_s6_ep == dec_ra_addr_ep || rf_addr_s6_ep == dec_ra_addr_op) && is_lat6) ||
        ((rf_addr_s1_ep == dec_rb_addr_ep || rf_addr_s1_ep == dec_rb_addr_op) && is_lat1) ||
        ((rf_addr_s2_ep == dec_rb_addr_ep || rf_addr_s2_ep == dec_rb_addr_op) && is_lat2) ||
        ((rf_addr_s3_ep == dec_rb_addr_ep || rf_addr_s3_ep == dec_rb_addr_op) && is_lat3) ||
        ((rf_addr_s4_ep == dec_rb_addr_ep || rf_addr_s4_ep == dec_rb_addr_op) && is_lat4) ||
        ((rf_addr_s5_ep == dec_rb_addr_ep || rf_addr_s5_ep == dec_rb_addr_op) && is_lat5) ||
        ((rf_addr_s6_ep == dec_rb_addr_ep || rf_addr_s6_ep == dec_rb_addr_op) && is_lat6) ||
        ((rf_addr_s1_ep == dec_rc_addr_ep || rf_addr_s1_ep == dec_rc_addr_op) && is_lat1) ||
        ((rf_addr_s2_ep == dec_rc_addr_ep || rf_addr_s2_ep == dec_rc_addr_op) && is_lat2) ||
        ((rf_addr_s3_ep == dec_rc_addr_ep || rf_addr_s3_ep == dec_rc_addr_op) && is_lat3) ||
        ((rf_addr_s4_ep == dec_rc_addr_ep || rf_addr_s4_ep == dec_rc_addr_op) && is_lat4) ||
        ((rf_addr_s5_ep == dec_rc_addr_ep || rf_addr_s5_ep == dec_rc_addr_op) && is_lat5) ||
        ((rf_addr_s6_ep == dec_rc_addr_ep || rf_addr_s6_ep == dec_rc_addr_op) && is_lat6) ||
        ((rf_addr_s1_op == dec_ra_addr_op || rf_addr_s1_op == dec_ra_addr_ep) && is_lat1) ||
        ((rf_addr_s2_op == dec_ra_addr_op || rf_addr_s2_op == dec_ra_addr_ep) && is_lat2) ||
        ((rf_addr_s3_op == dec_ra_addr_op || rf_addr_s3_op == dec_ra_addr_ep) && is_lat3) ||
        ((rf_addr_s4_op == dec_ra_addr_op || rf_addr_s4_op == dec_ra_addr_ep) && is_lat4) ||
        ((rf_addr_s5_op == dec_ra_addr_op || rf_addr_s5_op == dec_ra_addr_ep) && is_lat5) ||
        ((rf_addr_s6_op == dec_ra_addr_op || rf_addr_s6_op == dec_ra_addr_ep) && is_lat6) ||
        ((rf_addr_s1_op == dec_rb_addr_op || rf_addr_s1_op == dec_rb_addr_ep) && is_lat1) ||
        ((rf_addr_s2_op == dec_rb_addr_op || rf_addr_s2_op == dec_rb_addr_ep) && is_lat2) ||
        ((rf_addr_s3_op == dec_rb_addr_op || rf_addr_s3_op == dec_rb_addr_ep) && is_lat3) ||
        ((rf_addr_s4_op == dec_rb_addr_op || rf_addr_s4_op == dec_rb_addr_ep) && is_lat4) ||
        ((rf_addr_s5_op == dec_rb_addr_op || rf_addr_s5_op == dec_rb_addr_ep) && is_lat5) ||
        ((rf_addr_s6_op == dec_rb_addr_op || rf_addr_s6_op == dec_rb_addr_ep) && is_lat6) ||
        ((rf_addr_s1_op == dec_rc_addr_ep || rf_addr_s1_op == dec_rc_addr_op) && is_lat1) ||
        ((rf_addr_s2_op == dec_rc_addr_ep || rf_addr_s2_op == dec_rc_addr_op) && is_lat2) ||
        ((rf_addr_s3_op == dec_rc_addr_ep || rf_addr_s3_op == dec_rc_addr_op) && is_lat3) ||
        ((rf_addr_s4_op == dec_rc_addr_ep || rf_addr_s4_op == dec_rc_addr_op) && is_lat4) ||
        ((rf_addr_s5_op == dec_rc_addr_ep || rf_addr_s5_op == dec_rc_addr_op) && is_lat5) ||
        ((rf_addr_s6_op == dec_rc_addr_ep || rf_addr_s6_op == dec_rc_addr_op) && is_lat6)
      )
    begin
        dep_stall = 1'b1;
    end
end
```

```
endmodule
//end of file.
```

## 5.6 Appendix F – Even Pipe (even_pipe.sv)

```
import defines_pkg::*;

module even_pipe #(parameter OPCODE_LEN  = 11,
                parameter REG_ADDR_WD = 7,
                parameter REG_DATA_WD = 128)
(
   input  logic                 clk,
   input  logic                 rst,
   input  Opcodes               opcode,
   output logic                 rt_wr_en_ep,
   input  logic [0:127]         in_RA,
   input  logic [0:127]         in_RB,
   input  logic [0:127]         in_RC,
   input  logic [0:6]           in_I7,
   input  logic [0:7]           in_I8,
   input  logic [0:9]           in_I10,
   input  logic [0:15]          in_I16,
   input  logic [0:17]          in_I18,
   input  logic                 flush,
   input  logic [0:6]           in_RT_addr,
   output logic [0:6]           rf_addr_s1_ep,
   output logic [0:6]           rf_addr_s2_ep,
   output logic [0:6]           rf_addr_s3_ep,
   output logic [0:6]           rf_addr_s4_ep,
   output logic [0:6]           rf_addr_s5_ep,
   output logic [0:6]           rf_addr_s6_ep,
   output logic [0:6]           rf_addr_s7_ep,
   output logic [0:2]           rf_idx_s1_ep,
   output logic [0:2]           rf_idx_s2_ep,
   output logic [0:2]           rf_idx_s3_ep,
   output logic [0:2]           rf_idx_s4_ep,
   output logic [0:2]           rf_idx_s5_ep,
   output logic [0:2]           rf_idx_s6_ep,
   output logic [0:2]           rf_idx_s7_ep,
   output logic [0:127]         rf_data_s2_ep,
   output logic [0:127]         rf_data_s3_ep,
   output logic [0:127]         rf_data_s4_ep,
   output logic [0:127]         rf_data_s5_ep,
   output logic [0:127]         rf_data_s6_ep,
    output logic [0:127]            rf_data_s7_ep,
    output logic [0:6]              out_RT_addr,
    output logic [0:127]            out_RT
);

    logic           rt_wr_en;
    logic [0:127]   rf_data_s1_ep;
    logic           rf_s1_we;
    logic           rf_s2_we;
    logic           rf_s3_we;
    logic           rf_s4_we;
    logic           rf_s5_we;
    logic           rf_s6_we;
    logic           rf_s7_we;
    logic [0:127]   RT_reg;
    logic [0:32]    temp_reg;
    logic [0:7]     temp_byte_reg;
    logic [0:7]     cnt_reg;
    logic [0:2]     unit_idx;
    real            temp_fp;
    real            temp_op1;
    real            temp_op2;
    real            temp_op3;
    real            temp_fpe;

    logic [0:WORD-1]     rep_lb32_I16;
    logic [0:HALFWORD-1] rep_lb16_I10;
    logic [0:WORD-1]     rep_lb32_I10;

    assign rep_lb32_I16 = {{16{in_I16[0]}}, in_I16};
    assign rep_lb16_I10 = {{6{in_I10[0]}}, in_I10};
    assign rep_lb32_I10 = {{22{in_I10[0]}}, in_I10};

    //Rotate Variables
    logic [0:31]   result;
    logic [0:31]   operand;
    logic [0:4]    rotate;
    logic [0:4]    rotate_temp;

    always_ff @(posedge clk) begin
        if(rst) begin
            rf_addr_s1_ep <= 'd0;
            rf_addr_s2_ep <= 'd0;
            rf_addr_s3_ep <= 'd0;
    rf_addr_s4_ep <= 'd0;
    rf_addr_s5_ep <= 'd0;
    rf_addr_s6_ep <= 'd0;
    rf_addr_s7_ep <= 'd0;
    out_RT_addr   <= 'd0;
    rf_data_s1_ep <= 'd0;
    rf_data_s2_ep <= 'd0;
    rf_data_s3_ep <= 'd0;
    rf_data_s4_ep <= 'd0;
    rf_data_s5_ep <= 'd0;
    rf_data_s6_ep <= 'd0;
    rf_data_s7_ep <= 'd0;
    rf_idx_s1_ep  <= 'd0;
    rf_idx_s2_ep  <= 'd0;
    rf_idx_s3_ep  <= 'd0;
    rf_idx_s4_ep  <= 'd0;
    rf_idx_s5_ep  <= 'd0;
    rf_idx_s6_ep  <= 'd0;
    rf_idx_s7_ep  <= 'd0;
    out_RT        <= 'd0;
    rf_s1_we      <= 'd0;
    rf_s2_we      <= 'd0;
    rf_s3_we      <= 'd0;
    rf_s4_we      <= 'd0;
    rf_s5_we      <= 'd0;
    rf_s6_we      <= 'd0;
    rf_s7_we      <= 'd0;
    rt_wr_en_ep   <= 'd0;
end
else if(flush) begin
    rf_addr_s1_ep <= 'd0;
    rf_addr_s2_ep <= 'd0;
    rf_addr_s3_ep <= 'd0;
    rf_addr_s4_ep <= rf_addr_s3_ep;
    rf_addr_s5_ep <= rf_addr_s4_ep;
    rf_addr_s6_ep <= rf_addr_s5_ep;
    rf_addr_s7_ep <= rf_addr_s6_ep;
    rf_idx_s1_ep  <= 'd0;
    rf_idx_s2_ep  <= 'd0;
    rf_idx_s3_ep  <= 'd0;
    rf_idx_s4_ep  <= rf_idx_s3_ep;
    rf_idx_s5_ep  <= rf_idx_s4_ep;
    rf_idx_s6_ep  <= rf_idx_s5_ep;
    rf_idx_s7_ep  <= rf_idx_s6_ep;
    out_RT_addr   <= rf_addr_s7_ep;
    rf_data_s1_ep <= 'd0;
    rf_data_s2_ep <= 'd0;
    rf_data_s3_ep <= 'd0;
    rf_data_s4_ep <= rf_data_s3_ep;
    rf_data_s5_ep <= rf_data_s4_ep;
    rf_data_s6_ep <= rf_data_s5_ep;
    rf_data_s7_ep <= rf_data_s6_ep;
    out_RT        <= rf_data_s7_ep;
    rf_s1_we      <= 'd0;
    rf_s2_we      <= 'd0;
    rf_s3_we      <= 'd0;
    rf_s4_we      <= rf_s3_we;
    rf_s5_we      <= rf_s4_we;
    rf_s6_we      <= rf_s5_we;
    rf_s7_we      <= rf_s6_we;
    rt_wr_en_ep   <= rf_s7_we;
end
else begin
    rf_addr_s1_ep <= in_RT_addr;
    rf_addr_s2_ep <= rf_addr_s1_ep;
    rf_addr_s3_ep <= rf_addr_s2_ep;
    rf_addr_s4_ep <= rf_addr_s3_ep;
    rf_addr_s5_ep <= rf_addr_s4_ep;
    rf_addr_s6_ep <= rf_addr_s5_ep;
    rf_addr_s7_ep <= rf_addr_s6_ep;
    rf_idx_s1_ep  <= unit_idx;
    rf_idx_s2_ep  <= rf_idx_s1_ep;
    rf_idx_s3_ep  <= rf_idx_s2_ep;
    rf_idx_s4_ep  <= rf_idx_s3_ep;
    rf_idx_s5_ep  <= rf_idx_s4_ep;
    rf_idx_s6_ep  <= rf_idx_s5_ep;
    rf_idx_s7_ep  <= rf_idx_s6_ep;
    out_RT_addr   <= rf_addr_s7_ep;
    rf_data_s1_ep <= RT_reg;
    rf_data_s2_ep <= rf_data_s1_ep;
    rf_data_s3_ep <= rf_data_s2_ep;
    rf_data_s4_ep <= rf_data_s3_ep;
    rf_data_s5_ep <= rf_data_s4_ep;
    rf_data_s6_ep <= rf_data_s5_ep;
    rf_data_s7_ep <= rf_data_s6_ep;
    out_RT        <= rf_data_s7_ep;
    rf_s1_we      <= rt_wr_en;
    rf_s2_we      <= rf_s1_we;
```

```
        rf_s3_we     <= rf_s2_we;
        rf_s4_we     <= rf_s3_we;
        rf_s5_we     <= rf_s4_we;
        rf_s6_we     <= rf_s5_we;
        rf_s7_we     <= rf_s6_we;
        rt_wr_en_ep   <= rf_s7_we;
    end
end

always_comb
begin
    rt_wr_en = 'd0;
    RT_reg = 'd0;
    unit_idx = 'd0;
    cnt_reg = 'd0;

    operand = 'd0;
    result = 'd0;
    rotate = 'd0;
    rotate_temp = 'd0;

    case(opcode)

        //Simple Fixed Unit
        IMMEDIATE_LOAD_HALFWORD:
            begin
                for(int i=0; i < 8; i++) begin
                    RT_reg[i*HALFWORD +: HALFWORD] = in_I16;
                end
                unit_idx = 3'd1;
                rt_wr_en = 1;
            end

        IMMEDIATE_LOAD_WORD:
            begin
                for(int i=0; i < 4; i++) begin
                    RT_reg[i*WORD +: WORD] = rep_lb32_I16;
                end
                unit_idx = 3'd1;
                rt_wr_en = 1;
            end

        IMMEDIATE_LOAD_ADDRESS:
            begin
        for(int i=0; i < 4; i++) begin
            RT_reg[i*WORD +: WORD] = in_I18 & 18'h3ffff;
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
     end

ADD_HALF_WORD:
    begin
        for(int i=0; i < 8; i++) begin
            RT_reg[i*HALFWORD +: HALFWORD] = in_RA[i*HALFWORD +: HALFWORD] + in_RB[i*HALFWORD +: HALFWORD];
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
    end

ADD_HALF_WORD_IMMEDIATE:
    begin
        for(int i=0; i < 8; i++) begin
            RT_reg[i*HALFWORD +: HALFWORD] = in_RA[i*HALFWORD +: HALFWORD] + rep_lb16_I10;
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
    end

ADD_WORD:
    begin
        for(int i=0; i < 4; i++) begin
            RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] + in_RB[i*WORD +: WORD];
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
    end

ADD_WORD_IMMEDIATE:
    begin
        for(int i=0; i < 4; i++) begin
            RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] + rep_lb32_I10;
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
    end

SUBTRACT_FROM_HALFWORD:
    begin
        for(int i=0; i < 8; i++) begin
            RT_reg[i*HALFWORD +: HALFWORD] = in_RA[i*HALFWORD +: HALFWORD] - in_RB[i*HALFWORD +: HALFWORD];
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
    end

SUBTRACT_FROM_HALFWORD_IMMEDIATE:
    begin
        for(int i=0; i < 8; i++) begin
            RT_reg[i*HALFWORD +: HALFWORD] = in_RA[i*HALFWORD +: HALFWORD] - rep_lb16_I10;
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
    end

SUBTRACT_FROM_WORD:
    begin
        for(int i=0; i < 4; i++) begin
            RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] - in_RB[i*WORD +: WORD];
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
    end

SUBTRACT_FROM_WORD_IMMEDIATE:
    begin
        for(int i=0; i < 4; i++) begin
            RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] - rep_lb32_I10;
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
    end

ADD_EXTENDED:
    begin
        for(int i=0; i < 4; i++) begin
            RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] + in_RB[i*WORD +: WORD] + in_RC[i*WORD]; //TODO:Using RC
instead of RT for now. Recheck
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
    end
```

```
SUBTRACT_FROM_EXTENDED:
   begin
      for(int i=0; i < 4; i++) begin
         RT_reg[i*WORD +: WORD] = in_RB[i*WORD +: WORD] - in_RA[i*WORD +: WORD] + in_RC[i*WORD]; //TODO:Using RC
instead of RT for now. Recheck
      end
      unit_idx = 3'd1;
      rt_wr_en = 1;
   end

CARRY_GENERATE:
   begin
      for(int i=0; i < 4; i++) begin
         temp_reg = in_RA[i*WORD +: WORD] + in_RB[i*WORD +: WORD];
         RT_reg[i*WORD +: WORD] = {31'd0, temp_reg[0]};
      end
      unit_idx = 3'd1;
      rt_wr_en = 1;
   end

BORROW_GENERATE:
   begin
      for(int i=0; i < 4; i++) begin
         RT_reg[i*WORD +: WORD] = (in_RB[i*WORD +: WORD] >= in_RA[i*WORD +: WORD]) ? 32'd1 : 32'd0;
      end
      unit_idx = 3'd1;
      rt_wr_en = 1;
   end

COUNT_LEADING_ZEROS:
   begin
      for(int i=0; i < 4; i++) begin
         cnt_reg = 'd0;
         temp_reg = in_RA[i*WORD +: WORD];
         for(int j=0; j < WORD; j++) begin
            if(temp_reg[j] && cnt_reg == 'd0) begin
               cnt_reg = 'd1;
               RT_reg[i*WORD +: WORD] = j;
            end
         end
         if(cnt_reg == 'd0) RT_reg[i*WORD +: WORD] = 32;
      end
      unit_idx = 3'd1;
```

```
        rt_wr_en = 1;
    end

AND:
    begin
        for(int i=0; i < 4; i++) begin
            RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] & in_RB[i*WORD +: WORD];
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
    end

AND_WORD_IMMEDIATE:
    begin
        for(int i=0; i < 4; i++) begin
            RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] & rep_lb32_I10;
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
    end

AND_HALFWORD_IMMEDIATE:
    begin
        for(int i=0; i < 8; i++) begin
            RT_reg[i*HALFWORD +: HALFWORD] = in_RA[i*HALFWORD +: HALFWORD] & rep_lb16_I10;
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
    end

AND_BYTE_IMMEDIATE:
    begin
        for(int i=0; i < 4; i++) begin
            temp_reg[0:7] = in_I10 & 32'h00ff;
            temp_reg[0:31] = {4{temp_reg[0:7]}};
            RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] & temp_reg[0:31];
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
    end

OR:
    begin
        for(int i=0; i < 4; i++) begin
```

```
            RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] | in_RB[i*WORD +: WORD];
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
    end

OR_WORD_IMMEDIATE:
    begin
        for(int i=0; i < 4; i++) begin
            RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] | rep_lb32_I10;
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
    end

OR_HALFWORD_IMMEDIATE:
    begin
        for(int i=0; i < 8; i++) begin
            RT_reg[i*HALFWORD +: HALFWORD] = in_RA[i*HALFWORD +: HALFWORD] | rep_lb16_I10;
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
    end

OR_BYTE_IMMEDIATE:
    begin
        for(int i=0; i < 4; i++) begin
            temp_reg[0:7] = in_I10 & 32'h00ff;
            temp_reg[0:31] = {4{temp_reg[0:7]}};
            RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] | temp_reg[0:31];
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
    end

EXCLUSIVE_OR:
    begin
        for(int i=0; i < 4; i++) begin
            RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] ^ in_RB[i*WORD +: WORD];
        end
        unit_idx = 3'd1;
        rt_wr_en = 1;
    end
```

```
EXCLUSIVE_OR_WORD_IMMEDIATE:
   begin
      for(int i=0; i < 4; i++) begin
         RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] ^ rep_lb32_I10;
      end
      unit_idx = 3'd1;
      rt_wr_en = 1;
   end

EXCLUSIVE_OR_HALFWORD_IMMEDIATE:
   begin
      for(int i=0; i < 8; i++) begin
         RT_reg[i*HALFWORD +: HALFWORD] = in_RA[i*HALFWORD +: HALFWORD] ^ rep_lb16_I10;
      end
      unit_idx = 3'd1;
      rt_wr_en = 1;
   end

EXCLUSIVE_OR_BYTE_IMMEDIATE:
   begin
      for(int i=0; i < 4; i++) begin
         temp_reg[0:7] = in_I10 & 32'h00ff;
         temp_reg[0:31] = {4{temp_reg[0:7]}};
         RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] ^ temp_reg[0:31];
      end
      unit_idx = 3'd1;
      rt_wr_en = 1;
   end

NAND:
   begin
      for(int i=0; i < 4; i++) begin
         RT_reg[i*WORD +: WORD] = ~(in_RA[i*WORD +: WORD] & in_RB[i*WORD +: WORD]);
      end
      unit_idx = 3'd1;
      rt_wr_en = 1;
   end

NOR:
   begin
      for(int i=0; i < 4; i++) begin
         RT_reg[i*WORD +: WORD] = ~(in_RA[i*WORD +: WORD] | in_RB[i*WORD +: WORD]);
      end
      unit_idx = 3'd1;
```

```
                rt_wr_en = 1;
            end

        EQUIVALENT:
            begin
                for(int i=0; i < 4; i++) begin
                    RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] ^ (~in_RB[i*WORD +: WORD]);
                end
                unit_idx = 3'd1;
                rt_wr_en = 1;
            end

        COMPARE_EQUAL_WORD:
            begin
                for(int i=0; i < 4; i++) begin
                    RT_reg[i*WORD +: WORD] = (in_RA[i*WORD +: WORD] == in_RB[i*WORD +: WORD])? 32'hffffffff : 32'h0;
                end
                unit_idx = 3'd1;
                rt_wr_en = 1;
            end

        COMPARE_EQUAL_WORD_IMMEDIATE:
            begin
                for(int i=0; i < 4; i++) begin
                    RT_reg[i*WORD +: WORD] = (in_RA[i*WORD +: WORD] == rep_lb32_I10)? 32'hffffffff : 32'h0;
                end
                unit_idx = 3'd1;
                rt_wr_en = 1;
            end

        COMPARE_EQUAL_HALFWORD:
            begin
                for(int i=0; i < 8; i++) begin
                    RT_reg[i*HALFWORD +: HALFWORD] = (in_RA[i*HALFWORD +: HALFWORD] == in_RB[i*HALFWORD +: HALFWORD])?
32'hffff : 32'h0;
                end
                unit_idx = 3'd1;
                rt_wr_en = 1;
            end

        COMPARE_EQUAL_HALFWORD_IMMEDIATE:
            begin
                for(int i=0; i < 8; i++) begin
                    RT_reg[i*HALFWORD +: HALFWORD] = (in_RA[i*HALFWORD +: HALFWORD] == rep_lb16_I10)? 32'hffff : 32'h0;
                end
                unit_idx = 3'd1;
                rt_wr_en = 1;
            end

        COMPARE_GREATER_THAN_WORD:
            begin
                for(int i=0; i < 4; i++) begin
                    RT_reg[i*WORD +: WORD] = (in_RA[i*WORD +: WORD] > in_RB[i*WORD +: WORD])? 32'hffffffff : 32'h0;
                end
                unit_idx = 3'd1;
                rt_wr_en = 1;
            end

        COMPARE_GREATER_THAN_WORD_IMMEDIATE:
            begin
                for(int i=0; i < 4; i++) begin
                    RT_reg[i*WORD +: WORD] = (in_RA[i*WORD +: WORD] > rep_lb32_I10)? 32'hffffffff : 32'h0;
                end
                unit_idx = 3'd1;
                rt_wr_en = 1;
            end

        COMPARE_GREATER_THAN_HALFWORD:
            begin
                for(int i=0; i < 8; i++) begin
                    RT_reg[i*HALFWORD +: HALFWORD] = (in_RA[i*HALFWORD +: HALFWORD] > in_RB[i*HALFWORD +: HALFWORD])?
32'hffff : 32'h0;
                end
                unit_idx = 3'd1;
                rt_wr_en = 1;
            end

        COMPARE_GREATER_THAN_HALFWORD_IMMEDIATE:
            begin
                for(int i=0; i < 8; i++) begin
                    RT_reg[i*HALFWORD +: HALFWORD] = (in_RA[i*HALFWORD +: HALFWORD] > rep_lb16_I10)? 32'hffff : 32'h0;
                end
                unit_idx = 3'd1;
                rt_wr_en = 1;
            end

        //Single Precision Unit
        MULTIPLY:
            begin
                for(int i=0; i < 4; i++) begin
                    RT_reg[i*WORD +: WORD] = in_RA[((i*WORD)+HALFWORD) +: HALFWORD] * in_RB[((i*WORD)+HALFWORD) +:
HALFWORD];
                end
                unit_idx = 3'd3;
                rt_wr_en = 1;
            end

        MULTIPLY_IMMEDIATE:
            begin
                for(int i=0; i < 4; i++) begin
                    RT_reg[i*WORD +: WORD] = in_RA[((i*WORD)+HALFWORD) +: HALFWORD] * rep_lb16_I10;
                end
                unit_idx = 3'd3;
                rt_wr_en = 1;
            end

        MULTIPLY_AND_ADD:
            begin
                for(int i=0; i < 4; i++) begin
                    RT_reg[i*WORD +: WORD] = (in_RA[((i*WORD)+HALFWORD) +: HALFWORD] * in_RB[((i*WORD)+HALFWORD) +:
HALFWORD]) + in_RC[i*WORD +: WORD];
                end
                unit_idx = 3'd7;
                rt_wr_en = 1;
            end

        FLOATING_ADD:
            begin
                for(int i=0; i < 4; i++) begin
                    temp_op1 = $bitstoshortreal(in_RA[i*WORD +: WORD]);
                    temp_op2 = $bitstoshortreal(in_RB[i*WORD +: WORD]);
                    temp_fp = temp_op1 + temp_op2;
                    if (temp_fp < -S_MAX)                      RT_reg[i*WORD +: WORD] = -$shortrealtobits(S_MAX);
                    else if (temp_fp > S_MAX)                  RT_reg[i*WORD +: WORD] =  $shortrealtobits(S_MAX);
                    else if (temp_fp > -S_MIN && temp_fp < S_MIN) RT_reg[i*WORD +: WORD] =  0;
                    else                                       RT_reg[i*WORD +: WORD] =  $shortrealtobits(temp_fp);
                end
                unit_idx = 3'd3;
                rt_wr_en = 1;
            end

        FLOATING_SUBTRACT:
```

```
    begin
        for(int i=0; i < 4; i++) begin
            temp_op1 = $bitstoshortreal(in_RA[i*WORD +: WORD]);
            temp_op2 = $bitstoshortreal(in_RB[i*WORD +: WORD]);
            temp_fp = temp_op1 - temp_op2;
            if (temp_fp < -S_MAX)                       RT_reg[i*WORD +: WORD] = -$shortrealtobits(S_MAX);
            else if (temp_fp > S_MAX)                    RT_reg[i*WORD +: WORD] =  $shortrealtobits(S_MAX);
            else if (temp_fp > -S_MIN && temp_fp < S_MIN) RT_reg[i*WORD +: WORD] =  0;
            else                                    RT_reg[i*WORD +: WORD] =  $shortrealtobits(temp_fp);
        end
        unit_idx = 3'd3;
        rt_wr_en = 1;
    end

FLOATING_MULTIPLY:
    begin
        for(int i=0; i < 4; i++) begin
            temp_op1 = $bitstoshortreal(in_RA[i*WORD +: WORD]);
            temp_op2 = $bitstoshortreal(in_RB[i*WORD +: WORD]);
            temp_fp = temp_op1 * temp_op2;
            if (temp_fp < -S_MAX)                       RT_reg[i*WORD +: WORD] = -$shortrealtobits(S_MAX);
            else if (temp_fp > S_MAX)                    RT_reg[i*WORD +: WORD] =  $shortrealtobits(S_MAX);
            else if (temp_fp > -S_MIN && temp_fp < S_MIN) RT_reg[i*WORD +: WORD] =  0;
            else                                    RT_reg[i*WORD +: WORD] =  $shortrealtobits(temp_fp);
        end
        unit_idx = 3'd3;
        rt_wr_en = 1;
    end

FLOATING_MULTIPLY_AND_ADD:
    begin
        for(int i=0; i < 4; i++) begin
            temp_op1 = $bitstoshortreal(in_RA[i*WORD +: WORD]);
            temp_op2 = $bitstoshortreal(in_RB[i*WORD +: WORD]);
            temp_op3 = $bitstoshortreal(in_RC[i*WORD +: WORD]);
            temp_fp = temp_op1 * temp_op2 + temp_op3;
            if (temp_fp < -S_MAX)                       RT_reg[i*WORD +: WORD] = -$shortrealtobits(S_MAX);
            else if (temp_fp > S_MAX)                    RT_reg[i*WORD +: WORD] =  $shortrealtobits(S_MAX);
            else if (temp_fp > -S_MIN && temp_fp < S_MIN) RT_reg[i*WORD +: WORD] =  0;
            else                                    RT_reg[i*WORD +: WORD] =  $shortrealtobits(temp_fp);
        end
        unit_idx = 3'd3;
        rt_wr_en = 1;
    end
```

```
FLOATING_MULTIPLY_AND_SUBTRACT:
    begin
        for(int i=0; i < 4; i++) begin
            temp_op1 = $bitstoshortreal(in_RA[i*WORD +: WORD]);
            temp_op2 = $bitstoshortreal(in_RB[i*WORD +: WORD]);
            temp_op3 = $bitstoshortreal(in_RC[i*WORD +: WORD]);
            temp_fp = temp_op1 * temp_op2 - temp_op3;
            if (temp_fp < -S_MAX)                        RT_reg[i*WORD +: WORD] = -$shortrealtobits(S_MAX);
            else if (temp_fp > S_MAX)                    RT_reg[i*WORD +: WORD] =  $shortrealtobits(S_MAX);
            else if (temp_fp > -S_MIN && temp_fp < S_MIN) RT_reg[i*WORD +: WORD] =  0;
            else                                         RT_reg[i*WORD +: WORD] =  $shortrealtobits(temp_fp);
        end
        unit_idx = 3'd3;
        rt_wr_en = 1;
    end

FLOATING_COMPARE_EQUAL:
    begin
        for(int i=0; i < 4; i++) begin
            temp_fp  = $bitstoshortreal(in_RA[i*WORD +: WORD]);
            temp_fpe = $bitstoshortreal(in_RB[i*WORD +: WORD]);
            if (temp_fp == temp_fpe || (temp_fp < 1 && temp_fpe < 1)) begin
                RT_reg[i*WORD +: WORD] = 32'hffffffff;
            end
            else begin
                RT_reg[i*WORD +: WORD] = 32'h00000000;
            end
        end
        unit_idx = 3'd3;
        rt_wr_en = 1;
    end

FLOATING_COMPARE_MAGNITUDE_EQUAL:
    begin
        for(int i=0; i < 4; i++) begin
            temp_fp  = $bitstoshortreal(in_RA[i*WORD +: WORD]);
            temp_fpe = $bitstoshortreal(in_RB[i*WORD +: WORD]);
            temp_fp  = (temp_fp  >= 0) ? temp_fp  : -temp_fp;
            temp_fpe = (temp_fpe >= 0) ? temp_fpe : -temp_fpe;
            if (temp_fp == temp_fpe || (temp_fp < 1 && temp_fpe < 1)) begin
                RT_reg[i*WORD +: WORD] = 32'hffffffff;
            end
            else begin
```

```
                RT_reg[i*WORD +: WORD] = 32'h00000000;
            end
        end
        unit_idx = 3'd3;
        rt_wr_en = 1;
    end

FLOATING_COMPARE_GREATER_THAN:
    begin
        for(int i=0; i < 4; i++) begin
            temp_fp  = $bitstoshortreal(in_RA[i*WORD +: WORD]);
            temp_fpe = $bitstoshortreal(in_RB[i*WORD +: WORD]);
            if (temp_fp > temp_fpe && !(temp_fp < 1 && temp_fpe < 1)) begin
                RT_reg[i*WORD +: WORD] = 32'hffffffff;
            end
            else begin
                RT_reg[i*WORD +: WORD] = 32'h00000000;
            end
        end
        unit_idx = 3'd3;
        rt_wr_en = 1;
    end

FLOATING_COMPARE_MAGNITUDE_GREATER_THAN:
    begin
        for(int i=0; i < 4; i++) begin
            temp_fp  = $bitstoshortreal(in_RA[i*WORD +: WORD]);
            temp_fpe = $bitstoshortreal(in_RB[i*WORD +: WORD]);
            temp_fp  = (temp_fp  >= 0) ? temp_fp  : -temp_fp;
            temp_fpe = (temp_fpe >= 0) ? temp_fpe : -temp_fpe;
            if (temp_fp > temp_fpe && !(temp_fp < 1 && temp_fpe < 1)) begin
                RT_reg[i*WORD +: WORD] = 32'hffffffff;
            end
            else begin
                RT_reg[i*WORD +: WORD] = 32'h00000000;
            end
        end
        unit_idx = 3'd3;
        rt_wr_en = 1;
    end

//Byte Unit
COUNT_ONES_IN_BYTES:
    begin
```

```
            for(int i=0; i < 16; i++) begin
                cnt_reg = 0;
                temp_byte_reg = in_RA[i*BYTE +: BYTE];
                for(int i=0; i < 7; i++) begin
                    if(temp_byte_reg[i]) cnt_reg = cnt_reg + 1;
                end
                RT_reg[i*BYTE +: BYTE] = cnt_reg;
            end
            unit_idx = 3'd4;
            rt_wr_en = 1;
        end

    AVERAGE_BYTES:
        begin
            for(int i=0; i < 16; i++) begin
                RT_reg[i*BYTE +: BYTE] = ({8'd0, in_RA[i*BYTE +: BYTE]} + {8'd0, in_RB[i*BYTE +: BYTE]} + 1) >> 1;
            end
            unit_idx = 3'd4;
            rt_wr_en = 1;
        end

    ABSOLUTE_DIFFERENCE_OF_BYTES:
        begin
            for(int i=0; i < 16; i++) begin
                if(in_RB[i*BYTE +: BYTE] > in_RA[i*BYTE +: BYTE]) begin
                    RT_reg[i*BYTE +: BYTE] = in_RB[i*BYTE +: BYTE] - in_RA[i*BYTE +: BYTE];
                end
                else begin
                    RT_reg[i*BYTE +: BYTE] = in_RA[i*BYTE +: BYTE] - in_RB[i*BYTE +: BYTE];
                end
            end
            unit_idx = 3'd4;
            rt_wr_en = 1;
        end

    SUM_BYTES_INTO_HALFWORDS:
        begin
            for(int i=0; i < 4; i++) begin
                    RT_reg[(2*i+0)*HALFWORD +: HALFWORD] = in_RB[(4*i+0)*BYTE +: BYTE] + in_RB[(4*i+1)*BYTE +: BYTE] +
in_RB[(4*i+2)*BYTE +: BYTE] + in_RB[(4*i+3)*BYTE +: BYTE];
                    RT_reg[(2*i+1)*HALFWORD +: HALFWORD] = in_RB[(4*i+0)*BYTE +: BYTE] + in_RB[(4*i+1)*BYTE +: BYTE] +
in_RB[(4*i+2)*BYTE +: BYTE] + in_RB[(4*i+3)*BYTE +: BYTE];
            end
            unit_idx = 3'd4;
        rt_wr_en = 1;
    end

//Shift Unit
SHIFT_LEFT_HALFWORD:
    begin
            for(int i=0; i < 8; i++) begin
              if(in_RB[i*HALFWORD+26 +: 6] < 16) begin
                RT_reg[i*HALFWORD +: HALFWORD] = in_RA[i*HALFWORD +: HALFWORD] << in_RB[i*HALFWORD+26 +: 6];
              end
              else begin
                RT_reg[i*HALFWORD +: HALFWORD] = 'd0;
              end
        end

        unit_idx = 3'd2;
        rt_wr_en = 1;
    end

SHIFT_LEFT_HALFWORD_IMMEDIATE:
    begin
        if(in_I7[0:4] < 16) begin
            for(int i=0; i < 8; i++) begin
                RT_reg[i*HALFWORD +: HALFWORD] = in_RA[i*HALFWORD +: HALFWORD] << in_I7[0:4];
            end
        end
        else begin
            RT_reg = 'd0;
        end
        unit_idx = 3'd2;
        rt_wr_en = 1;
    end

SHIFT_LEFT_WORD:
    begin
        for (int i=0; i < 4; i++) begin
            if (in_RB[i*WORD+26 +: 6] < 32) begin
                RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] << in_RB[i*WORD+26 +: 6];
            end
            else begin
                RT_reg[i*WORD +: WORD] = 'd0;
            end
        end
        unit_idx = 3'd2;
        rt_wr_en = 1;
    end

SHIFT_LEFT_WORD_IMMEDIATE:
    begin
        if(in_I7[0:4] < 32) begin
            for(int i=0; i < 4; i++) begin
                RT_reg[i*WORD +: WORD] = in_RA[i*WORD +: WORD] << in_I7[0:4];
            end
        end
        else begin
            RT_reg = 'd0;
        end
        unit_idx = 3'd2;
        rt_wr_en = 1;
    end

ROTATE_WORD:
    begin
        for (int i=0; i < 4; i++) begin
            operand = in_RA[i*WORD +: WORD];
            rotate = in_RB[i*WORD +: 4];
            while(rotate > 32) begin
                rotate_temp = rotate - 32;
                rotate = rotate_temp;
            end
            for ( int j=0; j < rotate ; j++) begin
                result =  {operand[0:30],operand[0]};
                operand = result;
            end
            RT_reg[i*WORD +: WORD] = operand;
        end
        unit_idx = 3'd2;
        rt_wr_en = 1;
    end

ROTATE_WORD_IMMEDIATE:
    begin
        for (int i=0; i < 4; i++) begin
            operand = in_RA[i*WORD +: WORD];
            rotate = in_I7[0:4];
            while(rotate > 32) begin
                rotate_temp = rotate - 32;
                rotate = rotate_temp;
                end
                for ( int j=0; j < rotate ; j++) begin
                    result =  {operand[0:30],operand[0]};
                    operand = result;
                end
                RT_reg[i*WORD +: WORD] = operand;
            end
            unit_idx = 3'd2;
            rt_wr_en = 1;
        end

    LNOP:
        begin
            //No Operation
            rt_wr_en = 0;
        end

    STOP:
        begin
            $finish;
        end

    endcase
  end

endmodule
//end of file.
```

## 5.7 Appendix G - Odd Pipe (odd_pipe.sv)

```
import defines_pkg::*;

module odd_pipe #(parameter OPCODE_LEN  = 11,
              parameter REG_ADDR_WD = 7,
              parameter REG_DATA_WD = 128)
(

  input  logic                clk,
  input  logic                rst,
  input  Opcodes                opcode,
  output logic                 rt_wr_en_op,
  input  logic [0:127]          in_RA,
  input  logic [0:127]          in_RB,
  input  logic [0:127]          in_RC,
    input  logic [0:6]          in_I7,
    input  logic [0:7]          in_I8,
    input  logic [0:9]          in_I10,
    input  logic [0:15]         in_I16,
    input  logic [0:17]         in_I18,
    input  logic [0:6]          in_RT_addr,
    output logic [0:6]          rf_addr_s1_op,
    output logic [0:6]          rf_addr_s2_op,
    output logic [0:6]          rf_addr_s3_op,
    output logic [0:6]          rf_addr_s4_op,
    output logic [0:6]          rf_addr_s5_op,
    output logic [0:6]          rf_addr_s6_op,
    output logic [0:6]          rf_addr_s7_op,
    output logic [0:2]          rf_idx_s1_op,
    output logic [0:2]          rf_idx_s2_op,
    output logic [0:2]          rf_idx_s3_op,
    output logic [0:2]          rf_idx_s4_op,
    output logic [0:2]          rf_idx_s5_op,
    output logic [0:2]          rf_idx_s6_op,
    output logic [0:2]          rf_idx_s7_op,
    output logic [0:127]        rf_data_s2_op,
    output logic [0:127]        rf_data_s3_op,
    output logic [0:127]        rf_data_s4_op,
    output logic [0:127]        rf_data_s5_op,
    output logic [0:127]        rf_data_s6_op,
    output logic [0:127]        rf_data_s7_op,
    output logic [0:6]          out_RT_addr,
    output logic [0:127]        out_RT,
    input  logic [0:127]        in_ls_data,
    input  logic [0:31]         PC_in,
    output logic [0:31]         PC_out,
    output logic                cache_wr,
    output logic                out_ls_wr_en,
    output logic [0:31]         out_ls_addr,
    output logic [0:127]        out_ls_data,
    output logic                flush
  );

    localparam MASK1 = 128'hFFFFFFF0;
    localparam MASK2 = 128'hFFFFFFFC;

    logic           rt_wr_en;
    logic [0:127]   rf_data_s1_op;
    logic [0:31]    pc_s1;
    logic [0:31]     pc_s2;
    logic [0:31]     pc_s3;
    logic [0:31]     pc_s4;
    logic [0:31]     pc_s5;
    logic            flush_s1;
    logic            flush_s2;
    logic            flush_s3;
    logic            flush_s4;
    logic            flush_s5;
    logic            rf_s1_we;
    logic            rf_s2_we;
    logic            rf_s3_we;
    logic            rf_s4_we;
    logic            rf_s5_we;
    logic            rf_s6_we;
    logic            rf_s7_we;
    logic            flush_reg;
    logic [0:31]     PC_reg;
    logic [0:127]    RT_reg;
    logic [0:2]      unit_idx;

    logic [0:WORD-1] repc_lb32_I10;
    logic [0:WORD-1] repc_lb32_I16;

// Shift Rotate variables
    logic [0:127]    result;
    logic [0:127]    result_temp;

    assign repc_lb32_I10 = {{18{in_I10[0]}}, in_I10, 4'b0000};
    assign repc_lb32_I16 = {{14{in_I16[0]}}, in_I16, 2'b00};

    always_ff @(posedge clk) begin
        if(rst) begin
            rf_addr_s1_op <= 'd0;
            rf_addr_s2_op <= 'd0;
            rf_addr_s3_op <= 'd0;
            rf_addr_s4_op <= 'd0;
            rf_addr_s5_op <= 'd0;
            rf_addr_s6_op <= 'd0;
            rf_addr_s7_op <= 'd0;
            out_RT_addr   <= 'd0;
            rf_data_s1_op <= 'd0;
            rf_data_s2_op <= 'd0;
            rf_data_s3_op <= 'd0;
    rf_data_s4_op <= 'd0;
    rf_data_s5_op <= 'd0;
    rf_data_s6_op <= 'd0;
    rf_data_s7_op <= 'd0;
    rf_idx_s1_op  <= 'd0;
    rf_idx_s2_op  <= 'd0;
    rf_idx_s3_op  <= 'd0;
    rf_idx_s4_op  <= 'd0;
    rf_idx_s5_op  <= 'd0;
    rf_idx_s6_op  <= 'd0;
    rf_idx_s7_op  <= 'd0;
    out_RT        <= 'd0;
    rf_s1_we      <= 'd0;
    rf_s2_we      <= 'd0;
    rf_s3_we      <= 'd0;
    rf_s4_we      <= 'd0;
    rf_s5_we      <= 'd0;
    rf_s6_we      <= 'd0;
    rf_s7_we      <= 'd0;
    rt_wr_en_op   <= 'd0;
    pc_s1         <= 'd0;
    pc_s2         <= 'd0;
    pc_s3         <= 'd0;
    pc_s4         <= 'd0;
    pc_s5         <= 'd0;
    PC_out        <= 'd0;
    flush_s1      <= 'd0;
    flush_s2      <= 'd0;
    flush_s3      <= 'd0;
    flush_s4      <= 'd0;
    flush_s5      <= 'd0;
end
else if(flush) begin
    rf_addr_s1_op <= 'd0;
    rf_addr_s2_op <= 'd0;
    rf_addr_s3_op <= 'd0;
    rf_addr_s4_op <= rf_addr_s3_op;
    rf_addr_s5_op <= rf_addr_s4_op;
    rf_addr_s6_op <= rf_addr_s5_op;
    rf_addr_s7_op <= rf_addr_s6_op;
    out_RT_addr   <= rf_addr_s7_op;
    rf_idx_s1_op  <= 'd0;
    rf_idx_s2_op  <= 'd0;
    rf_idx_s3_op  <= 'd0;
    rf_idx_s4_op  <= rf_idx_s3_op;
    rf_idx_s5_op  <= rf_idx_s4_op;
    rf_idx_s6_op  <= rf_idx_s5_op;
    rf_idx_s7_op  <= rf_idx_s6_op;
    rf_data_s1_op <= 'd0;
    rf_data_s2_op <= 'd0;
    rf_data_s3_op <= 'd0;
    rf_data_s4_op <= rf_data_s3_op;
    rf_data_s5_op <= rf_data_s4_op;
    rf_data_s6_op <= rf_data_s5_op;
    rf_data_s7_op <= rf_data_s6_op;
    out_RT        <= rf_data_s7_op;
    rf_s1_we      <= 'd0;
    rf_s2_we      <= 'd0;
    rf_s3_we      <= 'd0;
    rf_s4_we      <= rf_s3_we;
    rf_s5_we      <= rf_s4_we;
    rf_s6_we      <= rf_s5_we;
    rf_s7_we      <= rf_s6_we;
    rt_wr_en_op   <= rf_s7_we;
    pc_s1         <= PC_reg;
    pc_s2         <= pc_s1;
    pc_s3         <= pc_s2;
    pc_s4         <= pc_s3;
    pc_s5         <= pc_s4;
    PC_out        <= pc_s5;
    flush_s1      <= 'd0;
    flush_s2      <= 'd0;
    flush_s3      <= 'd0;
    flush_s4      <= 'd0;
    flush_s5      <= 'd0;
    flush         <= 'd0;
  end
  else begin
    rf_addr_s1_op <= in_RT_addr;
    rf_addr_s2_op <= rf_addr_s1_op;
    rf_addr_s3_op <= rf_addr_s2_op;
    rf_addr_s4_op <= rf_addr_s3_op;
    rf_addr_s5_op <= rf_addr_s4_op;
    rf_addr_s6_op <= rf_addr_s5_op;
    rf_addr_s7_op <= rf_addr_s6_op;
    out_RT_addr   <= rf_addr_s7_op;
    rf_idx_s1_op  <= unit_idx;
    rf_idx_s2_op  <= rf_idx_s1_op;
```

```
        rf_idx_s3_op  <= rf_idx_s2_op;
        rf_idx_s4_op  <= rf_idx_s3_op;
        rf_idx_s5_op  <= rf_idx_s4_op;
        rf_idx_s6_op  <= rf_idx_s5_op;
        rf_idx_s7_op  <= rf_idx_s6_op;
        rf_data_s1_op <= RT_reg;
        rf_data_s2_op <= rf_data_s1_op;
        rf_data_s3_op <= rf_data_s2_op;
        rf_data_s4_op <= rf_data_s3_op;
        rf_data_s5_op <= rf_data_s4_op;
        rf_data_s6_op <= rf_data_s5_op;
        rf_data_s7_op <= rf_data_s6_op;
        out_RT        <= rf_data_s7_op;
        rf_s1_we      <= rt_wr_en;
        rf_s2_we      <= rf_s1_we;
        rf_s3_we      <= rf_s2_we;
        rf_s4_we      <= rf_s3_we;
        rf_s5_we      <= rf_s4_we;
        rf_s6_we      <= rf_s5_we;
        rf_s7_we      <= rf_s6_we;
        rt_wr_en_op   <= rf_s7_we;
        pc_s1         <= PC_reg;
        pc_s2         <= pc_s1;
        pc_s3         <= pc_s2;
        pc_s4         <= pc_s3;
        pc_s5         <= pc_s4;
        PC_out        <= pc_s5;
        flush_s1      <= flush_reg;
        flush_s2      <= flush_s1;
        flush_s3      <= flush_s2;
        flush_s4      <= flush_s3;
        flush_s5      <= flush_s4;
        flush         <= flush_s5;
    end
 end

always_comb
 begin
    rt_wr_en = 'd0;
    RT_reg = 'd0;
    PC_reg = 'd0;
    unit_idx = 'd0;
    flush_reg = 'd0;
    out_ls_wr_en = 'd0;
```

```
cache_wr = 'd0;

case(opcode)

    LOAD_QUADWORD_DFORM:
        begin
            out_ls_addr = $signed($signed(repc_lb32_I10) + $signed(in_RA[0:31])) & MASK1;
            RT_reg = in_ls_data;
            rt_wr_en = 1'b1;
            unit_idx = 3'd6;
        end

    LOAD_QUADWORD_AFORM:
        begin
            out_ls_addr = repc_lb32_I16 & MASK1;
            RT_reg = in_ls_data;
            rt_wr_en = 1'b1;
            unit_idx = 3'd6;
        end

    STORE_QUADWORD_DFORM:
        begin
            out_ls_addr = $signed($signed(repc_lb32_I10) + $signed(in_RA[0:31])) & MASK1;
            out_ls_data = in_RC;
            out_ls_wr_en = 1'b1;
            rt_wr_en = 1'b0;
            unit_idx = 3'd6;
        end

    STORE_QUADWORD_AFORM:
        begin
            out_ls_addr = repc_lb32_I16 & MASK1;
            out_ls_data = in_RC;
            out_ls_wr_en = 1'b1;
            rt_wr_en = 1'b0;
            unit_idx = 3'd6;
        end

    BRANCH_RELATIVE_AND_SET_LINK:
        begin
            RT_reg[0:31] = (PC_in + 32'd4);
            RT_reg[32:127] = 'd0;
            PC_reg = ($signed(PC_in) + $signed(repc_lb32_I16));
            rt_wr_en = 1'b1;
        flush_reg = 1;
        unit_idx = 3'd7;
    end

BRANCH_ABSOLUTE_AND_SET_LINK:
    begin
        RT_reg[0:31] = (PC_in + 32'd4);
        RT_reg[32:127] = 'd0;
        PC_reg = repc_lb32_I16;
        rt_wr_en = 1'b1;
        flush_reg = 1;
        unit_idx = 3'd7;
    end

BRANCH_INDIRECT:
    begin
        PC_reg = in_RA[0:31] & MASK2;
        rt_wr_en = 1'b0;
        flush_reg = 1;
        unit_idx = 3'd7;
    end

BRANCH_RELATIVE:
    begin
        PC_reg = ($signed(PC_in) + $signed(repc_lb32_I16));
        rt_wr_en = 1'b0;
        flush_reg = 1;
        unit_idx = 3'd7;
    end

BRANCH_ABSOLUTE:
    begin
        PC_reg = repc_lb32_I16;
        rt_wr_en = 1'b0;
        flush_reg = 1;
        unit_idx = 3'd7;
    end

BRANCH_IF_NOT_ZERO_WORD:
    begin
        if(in_RC[0:31] != 32'd0) begin
            PC_reg = (PC_in + repc_lb32_I16) & MASK2;
            flush_reg = 1;
        end
        else begin
            PC_reg = PC_in + 4;
            flush_reg = 0;
        end
        unit_idx = 3'd7;
        rt_wr_en = 1'b0;
    end

BRANCH_IF_ZERO_WORD:
    begin
        if(in_RC[0:31] == 32'd0) begin
            PC_reg = (PC_in + repc_lb32_I16) & MASK2;
            flush_reg = 1;
        end
        else begin
            PC_reg = PC_in + 4;
            flush_reg = 0;
        end
        unit_idx = 3'd7;
        rt_wr_en = 1'b0;
    end

BRANCH_IF_NOT_ZERO_HALFWORD:
    begin
        if(in_RC[16:31] != 16'd0) begin
            PC_reg = (PC_in + repc_lb32_I16) & MASK2;
            flush_reg = 1;
        end
        else begin
            PC_reg = PC_reg + 4;
            flush_reg = 0;
        end
        unit_idx = 3'd7;
        rt_wr_en = 1'b0;
    end

BRANCH_IF_ZERO_HALFWORD:
    begin
        if(in_RC[16:31] == 16'd0) begin
            PC_reg = (PC_in + repc_lb32_I16) & MASK2;
            flush_reg = 1;
        end
        else begin
            PC_reg = PC_in + 4;
            flush_reg = 0;
        end
        unit_idx = 3'd7;
        rt_wr_en = 1'b0;
    end

SHIFT_LEFT_QUADWORD_BY_BITS :
    begin
        RT_reg = in_RA << in_RB[29:31];
        rt_wr_en = 1'b1;
        unit_idx = 3'd5;
    end

SHIFT_LEFT_QUADWORD_BY_BITS_IMMEDIATE :
    begin
        RT_reg = in_RA << in_I7[4:6];
        rt_wr_en = 1'b1;
        unit_idx = 3'd5;
    end

SHIFT_LEFT_QUADWORD_BY_BYTES :
    begin
        if (in_RB[27:31] < 16) begin
            RT_reg = in_RA << in_RB[27:31] * 8;
        end
        else begin
            RT_reg = 'd0;
        end
        rt_wr_en = 1'b1;
        unit_idx = 3'd5;
    end

SHIFT_LEFT_QUADWORD_BY_BYTES_IMMEDIATE:
    begin
        if (in_I7[2:6] < 16) begin
            RT_reg = in_RA << in_I7[2:6] * 8;
        end
        else begin
            RT_reg = 'd0;
        end
        rt_wr_en = 1'b1;
        unit_idx = 3'd5;
    end
```

```
ROTATE_QUADWORD_BY_BITS:
  begin
    if (in_RB[29:31] < 8) begin
      result_temp = in_RA;
      for (int i = 0; i < in_RB[29:31]; i++) begin
        result = {result_temp[1:127],result_temp[0]};
        result_temp = result;
      end
      RT_reg = result_temp;
    end
    else begin
      RT_reg = 'd0;
    end
    unit_idx = 3'd5;
    rt_wr_en = 1'b1;
   end

 ROTATE_QUADWORD_BY_BITS_IMMEDIATE:
  begin
    if (in_I7[4:6] < 8) begin
      result_temp = in_RA;
      for (int i = 0; i < in_I7[4:6]; i++) begin
        result = {result_temp[1:127],result_temp[0]};
        result_temp = result;
      end
      RT_reg = result_temp;
    end
    else begin
      RT_reg = 'd0;
    end
    unit_idx = 3'd5;
    rt_wr_en = 1'b1;
  end

 ROTATE_QUADWORD_BY_BYTES:
  begin
    if (in_RB[27:31] < 16) begin
      result_temp = in_RA;
      for (int i = 0; i < in_RB[27:31]; i++) begin
        result = {result_temp[8:127],result_temp[0:7]};
        result_temp = result;
      end
      RT_reg = result_temp;
    end
```

```
                else begin
                    RT_reg = 'd0;
                end
                unit_idx = 3'd5;
                rt_wr_en = 1'b1;
            end

        ROTATE_QUADWORD_BY_BYTES_IMMEDIATE:
            begin
                if (in_I7[2:6] < 16) begin
                    result_temp = in_RA;
                    for (int i = 0; i < in_I7[2:6]; i++) begin
                        result = {result_temp[8:127],result_temp[0:7]};
                        result_temp = result;
                    end
                    RT_reg = result_temp;
                end
                else begin
                    RT_reg = 'd0;
                end
                unit_idx = 3'd5;
                rt_wr_en = 1'b1;
            end

        CMISS:
            begin
                cache_wr =  'd1;
                out_ls_addr = PC_in;
                rt_wr_en = 1'b0;
            end

        NOP:
            begin
                rt_wr_en = 1'b0;
            end

      endcase
   end
endmodule
//end of file.
```

## 5.8 Appendix H - Register File (reg_file.sv)

```
import defines_pkg::*;
```

```
module reg_file #(parameter NUM_REGS = 128,
              parameter RADDR_WD = $clog2(NUM_REGS))
(
   input  logic              clk,
   input  logic              rst,
   input  logic [6:0]          ra_addr_ep,
   input  logic [6:0]          rb_addr_ep,
   input  logic [6:0]          rc_addr_ep,
   input  logic [6:0]          rt_addr_ep,
   input  logic [6:0]          ra_addr_op,
   input  logic [6:0]          rb_addr_op,
   input  logic [6:0]          rc_addr_op,
   input  logic [6:0]          rt_addr_op,
   input  logic              rt_wr_en_ep,
   input  logic              rt_wr_en_op,
   input  logic [127:0]        rt_wr_ep,
   input  logic [127:0]        rt_wr_op,
   output logic [127:0]        ra_rd_ep,
   output logic [127:0]        rb_rd_ep,
   output logic [127:0]        rc_rd_ep,
   output logic [127:0]        ra_rd_op,
   output logic [127:0]        rb_rd_op,
   output logic [127:0]        rc_rd_op
);

   logic [127:0] reg_array[128];

   always_comb
   begin
      ra_rd_ep = reg_array[ra_addr_ep];
      rb_rd_ep = reg_array[rb_addr_ep];
      rc_rd_ep = reg_array[rc_addr_ep];
      ra_rd_op = reg_array[ra_addr_op];
      rb_rd_op = reg_array[rb_addr_op];
      rc_rd_op = reg_array[rc_addr_op];
   end

   always_ff @(posedge clk)
   begin
      if(rst) begin
         for(int i = 0; i < 128; i++) begin
            reg_array[i] <= 'd0;
         end
```

```
        end
        else begin
            if(rt_wr_en_ep) begin
                reg_array[rt_addr_ep] <= rt_wr_ep;
            end

            if(rt_wr_en_op) begin
                reg_array[rt_addr_op] <= rt_wr_op;
            end
        end
    end

endmodule
//end of file.
```

## 5.9 Appendix I - Forwarding Macro (fw_macro.sv)

```
import defines_pkg::*;

module fw_macro #(parameter OPCODE_LEN  = 11,
                  parameter REG_DATA_WD = 128)
(
    input  logic                clk,
    input  logic                rst,
    input  logic [6:0]          ra_addr_ep,
    input  logic [6:0]          rb_addr_ep,
    input  logic [6:0]          rc_addr_ep,
    input  logic [6:0]          rt_addr_ep,
    input  logic [6:0]          ra_addr_op,
    input  logic [6:0]          rb_addr_op,
    input  logic [6:0]          rc_addr_op,
    input  logic [6:0]          rt_addr_op,
    input  logic [127:0]        ra_data_ep,
    input  logic [127:0]        rb_data_ep,
    input  logic [127:0]        rc_data_ep,
    input  logic [127:0]        ra_data_op,
    input  logic [127:0]        rb_data_op,
    input  logic [127:0]        rc_data_op,
    input  logic [6:0]          rf_addr_s2_ep,
    input  logic [6:0]          rf_addr_s3_ep,
    input  logic [6:0]          rf_addr_s4_ep,
    input  logic [6:0]          rf_addr_s5_ep,
    input  logic [6:0]          rf_addr_s6_ep,
    input  logic [6:0]          rf_addr_s7_ep,
    input  logic [6:0]              rf_addr_ep,
    input  logic [6:0]              rf_addr_s2_op,
    input  logic [6:0]              rf_addr_s3_op,
    input  logic [6:0]              rf_addr_s4_op,
    input  logic [6:0]              rf_addr_s5_op,
    input  logic [6:0]              rf_addr_s6_op,
    input  logic [6:0]              rf_addr_s7_op,
    input  logic [6:0]              rf_addr_op,
    input  logic [127:0]            rf_data_s2_ep,
    input  logic [127:0]            rf_data_s3_ep,
    input  logic [127:0]            rf_data_s4_ep,
    input  logic [127:0]            rf_data_s5_ep,
    input  logic [127:0]            rf_data_s6_ep,
    input  logic [127:0]            rf_data_s7_ep,
    input  logic [127:0]            rf_data_ep,
    input  logic [127:0]            rf_data_s2_op,
    input  logic [127:0]            rf_data_s3_op,
    input  logic [127:0]            rf_data_s4_op,
    input  logic [127:0]            rf_data_s5_op,
    input  logic [127:0]            rf_data_s6_op,
    input  logic [127:0]            rf_data_s7_op,
    input  logic [127:0]            rf_data_op,
    input  logic [0:2]              rf_idx_s2_ep,
    input  logic [0:2]              rf_idx_s3_ep,
    input  logic [0:2]              rf_idx_s4_ep,
    input  logic [0:2]              rf_idx_s5_ep,
    input  logic [0:2]              rf_idx_s6_ep,
    input  logic [0:2]              rf_idx_s7_ep,
    input  logic [0:2]              rf_idx_s2_op,
    input  logic [0:2]              rf_idx_s3_op,
    input  logic [0:2]              rf_idx_s4_op,
    input  logic [0:2]              rf_idx_s5_op,
    input  logic [0:2]              rf_idx_s6_op,
    input  logic [0:2]              rf_idx_s7_op,
    output logic [127:0]            ra_fw_ep,
    output logic [127:0]            rb_fw_ep,
    output logic [127:0]            rc_fw_ep,
    output logic [127:0]            ra_fw_op,
    output logic [127:0]            rb_fw_op,
    output logic [127:0]            rc_fw_op
);

    assign ra_fw_ep = (rf_addr_s2_ep === ra_addr_ep && (rf_idx_s2_ep == 3'd1)) ? rf_data_s2_ep :
                (rf_addr_s3_ep === ra_addr_ep && (rf_idx_s3_ep == 3'd1 || rf_idx_s3_ep == 3'd2 || rf_idx_s3_ep == 3'd4)) ? rf_data_s3_ep :
                (rf_addr_s4_ep === ra_addr_ep && (rf_idx_s4_ep == 3'd1 || rf_idx_s4_ep == 3'd2 || rf_idx_s4_ep == 3'd4)) ? rf_data_s4_ep :
                (rf_addr_s5_ep === ra_addr_ep && (rf_idx_s5_ep == 3'd1 || rf_idx_s5_ep == 3'd2 || rf_idx_s5_ep == 3'd4)) ? rf_data_s5_ep :
                (rf_addr_s6_ep === ra_addr_ep && (rf_idx_s6_ep == 3'd1 || rf_idx_s6_ep == 3'd2 || rf_idx_s6_ep == 3'd4 || rf_idx_s6_ep ==
3'd3)) ? rf_data_s6_ep :
                (rf_addr_s7_ep === ra_addr_ep && (rf_idx_s7_ep == 3'd1 || rf_idx_s7_ep == 3'd2 || rf_idx_s7_ep == 3'd4 || rf_idx_s7_ep == 3'd3
|| rf_addr_s7_ep == 3'd7)) ? rf_data_s7_ep :
                (rf_addr_s4_op === ra_addr_ep && (rf_idx_s4_op == 3'd5)) ? rf_data_s4_op :
                (rf_addr_s5_op === ra_addr_ep && (rf_idx_s5_op == 3'd5)) ? rf_data_s5_op :
                (rf_addr_s6_op === ra_addr_ep && (rf_idx_s6_op == 3'd5 || rf_idx_s6_op == 3'd6)) ? rf_data_s6_op :
                (rf_addr_s7_op === ra_addr_ep && (rf_idx_s7_op == 3'd5 || rf_idx_s7_op == 3'd6)) ? rf_data_s7_op :
                (rf_addr_ep    === ra_addr_ep) ? rf_data_ep :
                (rf_addr_op    === ra_addr_ep) ? rf_data_op : ra_data_ep;

    assign rb_fw_ep = (rf_addr_s2_ep === rb_addr_ep && (rf_idx_s2_ep == 3'd1)) ? rf_data_s2_ep :
                (rf_addr_s3_ep === rb_addr_ep && (rf_idx_s3_ep == 3'd1 || rf_idx_s3_ep == 3'd2 || rf_idx_s3_ep == 3'd4)) ? rf_data_s3_ep :
                (rf_addr_s4_ep === rb_addr_ep && (rf_idx_s4_ep == 3'd1 || rf_idx_s4_ep == 3'd2 || rf_idx_s4_ep == 3'd4)) ? rf_data_s4_ep :
                (rf_addr_s5_ep === rb_addr_ep && (rf_idx_s5_ep == 3'd1 || rf_idx_s5_ep == 3'd2 || rf_idx_s5_ep == 3'd4)) ? rf_data_s5_ep :
                (rf_addr_s6_ep === rb_addr_ep && (rf_idx_s6_ep == 3'd1 || rf_idx_s6_ep == 3'd2 || rf_idx_s6_ep == 3'd4 || rf_idx_s6_ep ==
3'd3)) ? rf_data_s6_ep :
                (rf_addr_s7_ep === rb_addr_ep && (rf_idx_s7_ep == 3'd1 || rf_idx_s7_ep == 3'd2 || rf_idx_s7_ep == 3'd4 || rf_idx_s7_ep == 3'd3
|| rf_addr_s7_ep == 3'd7)) ? rf_data_s7_ep :
                (rf_addr_s4_op === rb_addr_ep && (rf_idx_s4_op == 3'd5)) ? rf_data_s4_op :
                (rf_addr_s5_op === rb_addr_ep && (rf_idx_s5_op == 3'd5)) ? rf_data_s5_op :
                (rf_addr_s6_op === rb_addr_ep && (rf_idx_s6_op == 3'd5 || rf_idx_s6_op == 3'd6)) ? rf_data_s6_op :
                (rf_addr_s7_op === rb_addr_ep && (rf_idx_s7_op == 3'd5 || rf_idx_s7_op == 3'd6)) ? rf_data_s7_op :
                (rf_addr_ep    === rb_addr_ep) ? rf_data_ep :
                (rf_addr_op    === rb_addr_ep) ? rf_data_op : rb_data_ep;

    assign rc_fw_ep = (rf_addr_s2_ep === rc_addr_ep && (rf_idx_s2_ep == 3'd1)) ? rf_data_s2_ep :
                (rf_addr_s3_ep === rc_addr_ep && (rf_idx_s3_ep == 3'd1 || rf_idx_s3_ep == 3'd2 || rf_idx_s3_ep == 3'd4)) ? rf_data_s3_ep :
                (rf_addr_s4_ep === rc_addr_ep && (rf_idx_s4_ep == 3'd1 || rf_idx_s4_ep == 3'd2 || rf_idx_s4_ep == 3'd4)) ? rf_data_s4_ep :
                (rf_addr_s5_ep === rc_addr_ep && (rf_idx_s5_ep == 3'd1 || rf_idx_s5_ep == 3'd2 || rf_idx_s5_ep == 3'd4)) ? rf_data_s5_ep :
                (rf_addr_s6_ep === rc_addr_ep && (rf_idx_s6_ep == 3'd1 || rf_idx_s6_ep == 3'd2 || rf_idx_s6_ep == 3'd4 || rf_idx_s6_ep ==
3'd3)) ? rf_data_s6_ep :
                (rf_addr_s7_ep === rc_addr_ep && (rf_idx_s7_ep == 3'd1 || rf_idx_s7_ep == 3'd2 || rf_idx_s7_ep == 3'd4 || rf_idx_s7_ep == 3'd3
|| rf_addr_s7_ep == 3'd7)) ? rf_data_s7_ep :
                (rf_addr_s4_op === rc_addr_ep && (rf_idx_s4_op == 3'd5)) ? rf_data_s4_op :
                (rf_addr_s5_op === rc_addr_ep && (rf_idx_s5_op == 3'd5)) ? rf_data_s5_op :
                (rf_addr_s6_op === rc_addr_ep && (rf_idx_s6_op == 3'd5 || rf_idx_s6_op == 3'd6)) ? rf_data_s6_op :
                (rf_addr_s7_op === rc_addr_ep && (rf_idx_s7_op == 3'd5 || rf_idx_s7_op == 3'd6)) ? rf_data_s7_op :
                (rf_addr_ep    === rc_addr_ep) ? rf_data_ep :
                (rf_addr_op    === rc_addr_ep) ? rf_data_op : rc_data_ep;

    assign ra_fw_op = (rf_addr_s2_ep === ra_addr_op && (rf_idx_s2_ep == 3'd1)) ? rf_data_s2_ep :
```

```
                    (rf_addr_s3_ep === ra_addr_op && (rf_idx_s3_ep == 3'd1 || rf_idx_s3_ep == 3'd2 || rf_idx_s3_ep == 3'd4)) ? rf_data_s3_ep :
                    (rf_addr_s4_ep === ra_addr_op && (rf_idx_s4_ep == 3'd1 || rf_idx_s4_ep == 3'd2 || rf_idx_s4_ep == 3'd4)) ? rf_data_s4_ep :
                    (rf_addr_s5_ep === ra_addr_op && (rf_idx_s5_ep == 3'd1 || rf_idx_s5_ep == 3'd2 || rf_idx_s5_ep == 3'd4)) ? rf_data_s5_ep :
                    (rf_addr_s6_ep === ra_addr_op && (rf_idx_s6_ep == 3'd1 || rf_idx_s6_ep == 3'd2 || rf_idx_s6_ep == 3'd4 || rf_idx_s6_ep ==
3'd3)) ? rf_data_s6_ep :
                    (rf_addr_s7_ep === ra_addr_op && (rf_idx_s7_ep == 3'd1 || rf_idx_s7_ep == 3'd2 || rf_idx_s7_ep == 3'd4 || rf_idx_s7_ep == 3'd3
|| rf_addr_s7_ep == 3'd7)) ? rf_data_s7_ep :
                    (rf_addr_s4_op === ra_addr_op && (rf_idx_s4_op == 3'd5)) ? rf_data_s4_op :
                    (rf_addr_s5_op === ra_addr_op && (rf_idx_s5_op == 3'd5)) ? rf_data_s5_op :
                    (rf_addr_s6_op === ra_addr_op && (rf_idx_s6_op == 3'd5 || rf_idx_s6_op == 3'd6)) ? rf_data_s6_op :
                    (rf_addr_s7_op === ra_addr_op && (rf_idx_s7_op == 3'd5 || rf_idx_s7_op == 3'd6)) ? rf_data_s7_op :
                    (rf_addr_ep    === ra_addr_op) ? rf_data_ep :
                    (rf_addr_op    === ra_addr_op) ? rf_data_op : ra_data_op;

   assign rb_fw_op = (rf_addr_s2_ep === rb_addr_op && (rf_idx_s2_ep == 3'd1)) ? rf_data_s2_ep :
                    (rf_addr_s3_ep === rb_addr_op && (rf_idx_s3_ep == 3'd1 || rf_idx_s3_ep == 3'd2 || rf_idx_s3_ep == 3'd4)) ? rf_data_s3_ep :
                    (rf_addr_s4_ep === rb_addr_op && (rf_idx_s4_ep == 3'd1 || rf_idx_s4_ep == 3'd2 || rf_idx_s4_ep == 3'd4)) ? rf_data_s4_ep :
                    (rf_addr_s5_ep === rb_addr_op && (rf_idx_s5_ep == 3'd1 || rf_idx_s5_ep == 3'd2 || rf_idx_s5_ep == 3'd4)) ? rf_data_s5_ep :
                    (rf_addr_s6_ep === rb_addr_op && (rf_idx_s6_ep == 3'd1 || rf_idx_s6_ep == 3'd2 || rf_idx_s6_ep == 3'd4 || rf_idx_s6_ep ==
3'd3)) ? rf_data_s6_ep :
                    (rf_addr_s7_ep === rb_addr_op && (rf_idx_s7_ep == 3'd1 || rf_idx_s7_ep == 3'd2 || rf_idx_s7_ep == 3'd4 || rf_idx_s7_ep == 3'd3
|| rf_addr_s7_ep == 3'd7)) ? rf_data_s7_ep :
                    (rf_addr_s4_op === rb_addr_op && (rf_idx_s4_op == 3'd5)) ? rf_data_s4_op :
                    (rf_addr_s5_op === rb_addr_op && (rf_idx_s5_op == 3'd5)) ? rf_data_s5_op :
                    (rf_addr_s6_op === rb_addr_op && (rf_idx_s6_op == 3'd5 || rf_idx_s6_op == 3'd6)) ? rf_data_s6_op :
                    (rf_addr_s7_op === rb_addr_op && (rf_idx_s7_op == 3'd5 || rf_idx_s7_op == 3'd6)) ? rf_data_s7_op :
                    (rf_addr_ep    === rb_addr_op) ? rf_data_ep :
                    (rf_addr_op    === rb_addr_op) ? rf_data_op : rb_data_op;

   assign rc_fw_op = (rf_addr_s2_ep === rc_addr_op && (rf_idx_s2_ep == 3'd1)) ? rf_data_s2_ep :
                    (rf_addr_s3_ep === rc_addr_op && (rf_idx_s3_ep == 3'd1 || rf_idx_s3_ep == 3'd2 || rf_idx_s3_ep == 3'd4)) ? rf_data_s3_ep :
                    (rf_addr_s4_ep === rc_addr_op && (rf_idx_s4_ep == 3'd1 || rf_idx_s4_ep == 3'd2 || rf_idx_s4_ep == 3'd4)) ? rf_data_s4_ep :
                    (rf_addr_s5_ep === rc_addr_op && (rf_idx_s5_ep == 3'd1 || rf_idx_s5_ep == 3'd2 || rf_idx_s5_ep == 3'd4)) ? rf_data_s5_ep :
                    (rf_addr_s6_ep === rc_addr_op && (rf_idx_s6_ep == 3'd1 || rf_idx_s6_ep == 3'd2 || rf_idx_s6_ep == 3'd4 || rf_idx_s6_ep ==
3'd3)) ? rf_data_s6_ep :
                    (rf_addr_s7_ep === rc_addr_op && (rf_idx_s7_ep == 3'd1 || rf_idx_s7_ep == 3'd2 || rf_idx_s7_ep == 3'd4 || rf_idx_s7_ep == 3'd3
|| rf_addr_s7_ep == 3'd7)) ? rf_data_s7_ep :
                    (rf_addr_s4_op === rc_addr_op && (rf_idx_s4_op == 3'd5)) ? rf_data_s4_op :
                    (rf_addr_s5_op === rc_addr_op && (rf_idx_s5_op == 3'd5)) ? rf_data_s5_op :
                    (rf_addr_s6_op === rc_addr_op && (rf_idx_s6_op == 3'd5 || rf_idx_s6_op == 3'd6)) ? rf_data_s6_op :
                    (rf_addr_s7_op === rc_addr_op && (rf_idx_s7_op == 3'd5 || rf_idx_s7_op == 3'd6)) ? rf_data_s7_op :
                    (rf_addr_ep    === rc_addr_op) ? rf_data_ep :
                    (rf_addr_op    === rc_addr_op) ? rf_data_op : rc_data_op;
```

```
endmodule
//end of file.
```

## 5.10 Appendix J – Defines Package (defines_pkg.sv)

```
package defines_pkg;

   parameter NUM_INS = 100;
   parameter EVENINSFILE = "even_ins_file.txt";
   parameter ODDINSFILE  = "odd_ins_file.txt";
   parameter LSLOADFILE  = "ls_load_file.txt";

   parameter BYTE       =   8;
   parameter HALFWORD   =  16;
   parameter WORD       =  32;
   parameter DOUBLEWORD =  64;
   parameter QUADWORD   = 128;

   parameter LS_SIZE    = 32000;

   shortreal S_MAX = $bitstoshortreal(32'h7fffffff);
   shortreal S_MIN = $bitstoshortreal(32'h00800000);

   typedef enum logic [10:0] {  //Simple Fixed Instructions
                        IMMEDIATE_LOAD_HALFWORD                = 11'b00010000011,
                        IMMEDIATE_LOAD_WORD                    = 11'b00010000001,
                        IMMEDIATE_LOAD_ADDRESS                 = 11'b00000100001,
                        ADD_HALF_WORD                          = 11'b00011001000,
                        ADD_HALF_WORD_IMMEDIATE                = 11'b00000011101,
                        ADD_WORD                               = 11'b00011000000,
                        ADD_WORD_IMMEDIATE                     = 11'b00000011100,
                        SUBTRACT_FROM_HALFWORD                 = 11'b00001001000,
                        SUBTRACT_FROM_HALFWORD_IMMEDIATE       = 11'b00000001101,
                        SUBTRACT_FROM_WORD                     = 11'b00001000000,
                        SUBTRACT_FROM_WORD_IMMEDIATE           = 11'b00000001100,
                        ADD_EXTENDED                           = 11'b01101000000,
                        SUBTRACT_FROM_EXTENDED                 = 11'b01101000001,
                        CARRY_GENERATE                         = 11'b00011000010,
                        BORROW_GENERATE                        = 11'b00001000010,
                        COUNT_LEADING_ZEROS                    = 11'b01010100101,
                        AND                                    = 11'b00011000001,
                        AND_WORD_IMMEDIATE                     = 11'b00000010100,
                        AND_HALFWORD_IMMEDIATE                 = 11'b00000010101,
                        AND_BYTE_IMMEDIATE                     = 11'b00000010110,
                        OR                                  = 11'b00001000001,
                        OR_WORD_IMMEDIATE                   = 11'b00000000100,
                        OR_HALFWORD_IMMEDIATE               = 11'b00000000101,
                        OR_BYTE_IMMEDIATE                   = 11'b00000000110,
                        EXCLUSIVE_OR                        = 11'b01001000001,
                        EXCLUSIVE_OR_WORD_IMMEDIATE         = 11'b00001000100,
                        EXCLUSIVE_OR_HALFWORD_IMMEDIATE     = 11'b00001000101,
                        EXCLUSIVE_OR_BYTE_IMMEDIATE         = 11'b00001000110,
                        NAND                                = 11'b00011001001,
                        NOR                                 = 11'b00001001001,
                        EQUIVALENT                          = 11'b01001001001,
                        COMPARE_EQUAL_WORD                  = 11'b01111000000,
                        COMPARE_EQUAL_WORD_IMMEDIATE        = 11'b00001111100,
                        COMPARE_EQUAL_HALFWORD              = 11'b01111001000,
                        COMPARE_EQUAL_HALFWORD_IMMEDIATE    = 11'b00001111101,
                        COMPARE_GREATER_THAN_WORD           = 11'b01001000000,
                        COMPARE_GREATER_THAN_WORD_IMMEDIATE = 11'b00001001100,
                        COMPARE_GREATER_THAN_HALFWORD       = 11'b01001001000,
                        COMPARE_GREATER_THAN_HALFWORD_IMMEDIATE = 11'b00001001101,
                        //Single Precision Unit
                        MULTIPLY                            = 11'b01111000100,
                        MULTIPLY_IMMEDIATE                  = 11'b00001110100,
                        MULTIPLY_AND_ADD                    = 11'b10000001100,
                        FLOATING_ADD                        = 11'b01011000100,
                        FLOATING_SUBTRACT                   = 11'b01011000101,
                        FLOATING_MULTIPLY                   = 11'b01011000110,
                        FLOATING_MULTIPLY_AND_ADD           = 11'b00000001110,
                        FLOATING_MULTIPLY_AND_SUBTRACT      = 11'b00000001111,
                        FLOATING_COMPARE_EQUAL              = 11'b01111000010,
                        FLOATING_COMPARE_MAGNITUDE_EQUAL    = 11'b01111001010,
                        FLOATING_COMPARE_GREATER_THAN       = 11'b01011000010,
                        FLOATING_COMPARE_MAGNITUDE_GREATER_THAN = 11'b01011001010,
                        //Byte Unit
                        COUNT_ONES_IN_BYTES                 = 11'b01010110100,
                        AVERAGE_BYTES                       = 11'b00011010011,
                        ABSOLUTE_DIFFERENCE_OF_BYTES        = 11'b00001010011,
                        SUM_BYTES_INTO_HALFWORDS            = 11'b01001010011,
                        //Shift and Rotate
                        SHIFT_LEFT_HALFWORD                 = 11'b00001011111,
                        SHIFT_LEFT_HALFWORD_IMMEDIATE       = 11'b00001111111,
                        SHIFT_LEFT_WORD                     = 11'b00001011011,
                        SHIFT_LEFT_WORD_IMMEDIATE           = 11'b00001111011,
                        SHIFT_LEFT_QUADWORD_BY_BITS         = 11'b00111011011,
                        SHIFT_LEFT_QUADWORD_BY_BITS_IMMEDIATE = 11'b00111111011,
                                        SHIFT_LEFT_QUADWORD_BY_BYTES           = 11'b00111011111,
                                        SHIFT_LEFT_QUADWORD_BY_BYTES_IMMEDIATE   = 11'b00111111111,
                                        ROTATE_WORD                          = 11'b00001011000,
                                        ROTATE_WORD_IMMEDIATE                  = 11'b00001111000,
                                        ROTATE_QUADWORD_BY_BITS                = 11'b00111011000,
                                        ROTATE_QUADWORD_BY_BITS_IMMEDIATE        = 11'b00111111000,
                                        ROTATE_QUADWORD_BY_BYTES               = 11'b00111011100,
                                        ROTATE_QUADWORD_BY_BYTES_IMMEDIATE       = 11'b00111111100,
                                        //Load Store Instructions
                                        LOAD_QUADWORD_DFORM                   = 11'b00000110100,
                                        LOAD_QUADWORD_AFORM                   = 11'b00001100001,
                                        STORE_QUADWORD_DFORM                  = 11'b00000100100,
                                        STORE_QUADWORD_AFORM                  = 11'b10001000001,
                                        BRANCH_RELATIVE_AND_SET_LINK           = 11'b00001100110,
                                        BRANCH_ABSOLUTE_AND_SET_LINK           = 11'b00001100010,
                                        BRANCH_INDIRECT                       = 11'b00110101000,
                                        BRANCH_RELATIVE                       = 11'b00001100100,
                                        BRANCH_ABSOLUTE                       = 11'b00001100000,
                                        BRANCH_IF_NOT_ZERO_WORD                = 11'b10001000010,
                                        BRANCH_IF_ZERO_WORD                    = 11'b10001000000,
                                        BRANCH_IF_NOT_ZERO_HALFWORD            = 11'b10001000110,
                                        BRANCH_IF_ZERO_HALFWORD                = 11'b10001000100,
                                        //Control
                                        CMISS                                = 11'b11111111111,
                                        LNOP                                 = 11'b00000000001,
                                        NOP                                  = 11'b01000000001,
                                        STOP                                 = 11'b00000000000
                                      } Opcodes;

                        endpackage
```

## 5.11 Appendix K – SPU pipes top (spu_pipes_top.sv)

```
import defines_pkg::*;

module spu_pipes_top #(parameter OPCODE_LEN  = 11,
                  parameter REG_DATA_WD = 128)
(
   input  logic                   clk,
   input  logic                   rst,
   input  Opcodes                  opcode_ep,
   input  Opcodes                  opcode_op,
   input  logic [0:6]              ra_addr_ep,
   input  logic [0:6]              rb_addr_ep,
input  logic [0:6]              rc_addr_ep,
input  logic [0:6]              rt_addr_ep,
input  logic [0:6]              ra_addr_op,
input  logic [0:6]              rb_addr_op,
input  logic [0:6]              rc_addr_op,
input  logic [0:6]              rt_addr_op,
input  logic [0:6]              in_I7e,
input  logic [0:7]              in_I8e,
input  logic [0:9]              in_I10e,
input  logic [0:15]              in_I16e,
input  logic [0:17]              in_I18e,
input  logic [0:6]              in_I7o,
input  logic [0:7]              in_I8o,
input  logic [0:9]              in_I10o,
input  logic [0:15]              in_I16o,
input  logic [0:17]              in_I18o,
output logic [0:6]              rf_addr_s1_ep,
output logic [0:6]              rf_addr_s2_ep,
output logic [0:6]              rf_addr_s3_ep,
output logic [0:6]              rf_addr_s4_ep,
output logic [0:6]              rf_addr_s5_ep,
output logic [0:6]              rf_addr_s6_ep,
output logic [0:6]              rf_addr_s1_op,
output logic [0:6]              rf_addr_s2_op,
output logic [0:6]              rf_addr_s3_op,
output logic [0:6]              rf_addr_s4_op,
output logic [0:6]              rf_addr_s5_op,
output logic [0:6]              rf_addr_s6_op,
output logic [0:2]              rf_idx_s1_ep,
output logic [0:2]              rf_idx_s2_ep,
output logic [0:2]              rf_idx_s3_ep,
output logic [0:2]              rf_idx_s4_ep,
output logic [0:2]              rf_idx_s5_ep,
output logic [0:2]              rf_idx_s6_ep,
output logic [0:2]              rf_idx_s1_op,
output logic [0:2]              rf_idx_s2_op,
output logic [0:2]              rf_idx_s3_op,
output logic [0:2]              rf_idx_s4_op,
output logic [0:2]              rf_idx_s5_op,
output logic [0:2]              rf_idx_s6_op,
output logic                   flush,
input  logic [0:31]              PC_in,
input  logic [0:127]              ls_data_rd,
output logic [0:127]              ls_data_wr,
    output logic [0:31]          ls_addr,
    output logic                 ls_wr_en,
    output logic                 cache_wr,
    output logic [0:31]          PC_out
);

    logic              rt_wr_en_ep;
    logic              rt_wr_en_op;
    logic [0:127]        rt_wr_ep;
    logic [0:127]        rt_wr_op;
    logic [0:127]        ra_rd_ep;
    logic [0:127]        rb_rd_ep;
    logic [0:127]        rc_rd_ep;
    logic [0:127]        ra_rd_op;
    logic [0:127]        rb_rd_op;
    logic [0:127]        rc_rd_op;
    logic [0:6]          rf_addr_s7_ep;
    logic [0:6]          rf_addr_s7_op;
    logic [0:127]        rf_data_s2_ep;
    logic [0:127]        rf_data_s3_ep;
    logic [0:127]        rf_data_s4_ep;
    logic [0:127]        rf_data_s5_ep;
    logic [0:127]        rf_data_s6_ep;
    logic [0:127]        rf_data_s7_ep;
    logic [0:127]        rf_data_s2_op;
    logic [0:127]        rf_data_s3_op;
    logic [0:127]        rf_data_s4_op;
    logic [0:127]        rf_data_s5_op;
    logic [0:127]        rf_data_s6_op;
    logic [0:127]        rf_data_s7_op;
    logic [0:127]        ra_fw_ep;
    logic [0:127]        rb_fw_ep;
    logic [0:127]        rc_fw_ep;
    logic [0:127]        ra_fw_op;
    logic [0:127]        rb_fw_op;
    logic [0:127]        rc_fw_op;
    logic [0:6]          rt_fw_addr_ep;
    logic [0:6]          rt_fw_addr_op;
    logic [0:2]          rf_idx_s7_ep;
    logic [0:2]          rf_idx_s7_op;

    reg_file u_reg_file (
        .clk        ( clk        ),
        .rst        ( rst        ),
        .ra_addr_ep   ( ra_addr_ep  ),
        .rb_addr_ep   ( rb_addr_ep  ),
        .rc_addr_ep   ( rc_addr_ep  ),
        .rt_addr_ep   ( rt_fw_addr_ep  ),
        .ra_addr_op   ( ra_addr_op  ),
        .rb_addr_op   ( rb_addr_op  ),
        .rc_addr_op   ( rc_addr_op  ),
        .rt_addr_op   ( rt_fw_addr_op  ),
        .rt_wr_en_ep  ( rt_wr_en_ep ),
        .rt_wr_en_op  ( rt_wr_en_op ),
        .rt_wr_ep     ( rt_wr_ep    ),
        .rt_wr_op     ( rt_wr_op    ),
        .ra_rd_ep     ( ra_rd_ep    ),
        .rb_rd_ep     ( rb_rd_ep    ),
        .rc_rd_ep     ( rc_rd_ep    ),
        .ra_rd_op     ( ra_rd_op    ),
        .rb_rd_op     ( rb_rd_op    ),
        .rc_rd_op     ( rc_rd_op    )
    );

    fw_macro u_fw_macro (
        .clk          ( clk         ),
        .rst          ( rst         ),
        .ra_addr_ep   ( ra_addr_ep  ),
        .rb_addr_ep   ( rb_addr_ep  ),
        .rc_addr_ep   ( rc_addr_ep  ),
        .rt_addr_ep   ( rt_addr_ep  ),
        .ra_addr_op   ( ra_addr_op  ),
        .rb_addr_op   ( rb_addr_op  ),
        .rc_addr_op   ( rc_addr_op  ),
        .rt_addr_op   ( rt_addr_op  ),
        .ra_data_ep   ( ra_rd_ep    ),
        .rb_data_ep   ( rb_rd_ep    ),
        .rc_data_ep   ( rc_rd_ep    ),
        .ra_data_op   ( ra_rd_op    ),
        .rb_data_op   ( rb_rd_op    ),
        .rc_data_op   ( rc_rd_op    ),
        .rf_addr_s2_ep( rf_addr_s2_ep ),
        .rf_addr_s3_ep( rf_addr_s3_ep ),
        .rf_addr_s4_ep( rf_addr_s4_ep ),
        .rf_addr_s5_ep( rf_addr_s5_ep ),
        .rf_addr_s6_ep( rf_addr_s6_ep ),
        .rf_addr_s7_ep( rf_addr_s7_ep ),
        .rf_addr_ep   ( rt_fw_addr_ep ),
        .rf_addr_s2_op( rf_addr_s2_op ),
        .rf_addr_s3_op( rf_addr_s3_op ),
        .rf_addr_s4_op( rf_addr_s4_op ),
        .rf_addr_s5_op( rf_addr_s5_op ),
        .rf_addr_s6_op( rf_addr_s6_op ),
        .rf_addr_s7_op( rf_addr_s7_op ),
        .rf_addr_op   ( rt_fw_addr_op ),
        .rf_data_s2_ep( rf_data_s2_ep ),
        .rf_data_s3_ep( rf_data_s3_ep ),
        .rf_data_s4_ep( rf_data_s4_ep ),
        .rf_data_s5_ep( rf_data_s5_ep ),
        .rf_data_s6_ep( rf_data_s6_ep ),
        .rf_data_s7_ep( rf_data_s7_ep ),
        .rf_data_ep   ( rt_wr_ep ),
        .rf_data_s2_op( rf_data_s2_op ),
        .rf_data_s3_op( rf_data_s3_op ),
        .rf_data_s4_op( rf_data_s4_op ),
        .rf_data_s5_op( rf_data_s5_op ),
        .rf_data_s6_op( rf_data_s6_op ),
        .rf_data_s7_op( rf_data_s7_op ),
        .rf_data_op   ( rt_wr_op ),
        .rf_idx_s2_op ( rf_idx_s2_op  ),
        .rf_idx_s3_op ( rf_idx_s3_op  ),
        .rf_idx_s4_op ( rf_idx_s4_op  ),
        .rf_idx_s5_op ( rf_idx_s5_op  ),
        .rf_idx_s6_op ( rf_idx_s6_op  ),
        .rf_idx_s7_op ( rf_idx_s7_op  ),
        .rf_idx_s2_ep ( rf_idx_s2_ep  ),
        .rf_idx_s3_ep ( rf_idx_s3_ep  ),
        .rf_idx_s4_ep ( rf_idx_s4_ep  ),
        .rf_idx_s5_ep ( rf_idx_s5_ep  ),
        .rf_idx_s6_ep ( rf_idx_s6_ep  ),
        .rf_idx_s7_ep ( rf_idx_s7_ep  ),
        .ra_fw_ep     ( ra_fw_ep      ),
        .rb_fw_ep     ( rb_fw_ep      ),
        .rc_fw_ep     ( rc_fw_ep      ),
        .ra_fw_op     ( ra_fw_op      ),
        .rb_fw_op     ( rb_fw_op      ),
        .rc_fw_op     ( rc_fw_op      )
    );

    even_pipe u_even_pipe (
        .clk          (clk),
        .rst          (rst),
        .opcode       (opcode_ep),
        .rt_wr_en_ep  (rt_wr_en_ep),
        .in_RA        (ra_fw_ep),
        .in_RB        (rb_fw_ep),
        .in_RC        (rc_fw_ep),
        .in_I7        (in_I7e),
        .in_I8        (in_I8e),
        .in_I10       (in_I10e),
        .in_I16       (in_I16e),
        .in_I18       (in_I18e),
        .in_RT_addr   (rt_addr_ep),
        .flush        ( flush  ),
        .rf_addr_s1_ep ( rf_addr_s1_ep ),
        .rf_addr_s2_ep ( rf_addr_s2_ep ),
        .rf_addr_s3_ep ( rf_addr_s3_ep ),
        .rf_addr_s4_ep ( rf_addr_s4_ep ),
        .rf_addr_s5_ep ( rf_addr_s5_ep ),
        .rf_addr_s6_ep ( rf_addr_s6_ep ),
        .rf_addr_s7_ep ( rf_addr_s7_ep ),
        .rf_data_s2_ep ( rf_data_s2_ep ),
        .rf_data_s3_ep ( rf_data_s3_ep ),
        .rf_data_s4_ep ( rf_data_s4_ep ),
        .rf_data_s5_ep ( rf_data_s5_ep ),
        .rf_data_s6_ep ( rf_data_s6_ep ),
        .rf_data_s7_ep ( rf_data_s7_ep ),
        .rf_idx_s1_ep  ( rf_idx_s1_ep  ),
        .rf_idx_s2_ep  ( rf_idx_s2_ep  ),
        .rf_idx_s3_ep  ( rf_idx_s3_ep  ),
        .rf_idx_s4_ep  ( rf_idx_s4_ep  ),
        .rf_idx_s5_ep  ( rf_idx_s5_ep  ),
        .rf_idx_s6_ep  ( rf_idx_s6_ep  ),
        .rf_idx_s7_ep  ( rf_idx_s7_ep  ),
        .out_RT_addr   (rt_fw_addr_ep),
        .out_RT        (rt_wr_ep)
    );

    odd_pipe u_odd_pipe (
        .clk          (clk),
        .rst          (rst),
        .opcode       (opcode_op),
        .rt_wr_en_op  (rt_wr_en_op),
        .in_RA        (ra_fw_op),
        .in_RB        (rb_fw_op),
        .in_RC        (rc_fw_op),
        .in_I7         (in_I7o),
        .in_I8         (in_I8o),
        .in_I10        (in_I10o),
        .in_I16        (in_I16o),
        .in_I18        (in_I18o),
        .flush         ( flush  ),
        .in_RT_addr    (rt_addr_op),
        .rf_addr_s1_op ( rf_addr_s1_op ),
        .rf_addr_s2_op ( rf_addr_s2_op ),
        .rf_addr_s3_op ( rf_addr_s3_op ),
        .rf_addr_s4_op ( rf_addr_s4_op ),
        .rf_addr_s5_op ( rf_addr_s5_op ),
        .rf_addr_s6_op ( rf_addr_s6_op ),
        .rf_addr_s7_op ( rf_addr_s7_op ),
        .rf_data_s2_op ( rf_data_s2_op ),
        .rf_data_s3_op ( rf_data_s3_op ),
        .rf_data_s4_op ( rf_data_s4_op ),
        .rf_data_s5_op ( rf_data_s5_op ),
        .rf_data_s6_op ( rf_data_s6_op ),
        .rf_data_s7_op ( rf_data_s7_op ),
        .rf_idx_s1_op  ( rf_idx_s1_op  ),
        .rf_idx_s2_op  ( rf_idx_s2_op  ),
        .rf_idx_s3_op  ( rf_idx_s3_op  ),
        .rf_idx_s4_op  ( rf_idx_s4_op  ),
        .rf_idx_s5_op  ( rf_idx_s5_op  ),
        .rf_idx_s6_op  ( rf_idx_s6_op  ),
        .rf_idx_s7_op  ( rf_idx_s7_op  ),
        .in_ls_data    ( ls_data_rd    ),
        .out_ls_addr   ( ls_addr       ),
        .out_ls_data   ( ls_data_wr    ),
        .out_ls_wr_en  ( ls_wr_en      ),
        .PC_in         ( PC_in         ),
        .PC_out        ( PC_out        ),
        .cache_wr      ( cache_wr      ),
        .out_RT_addr   ( rt_fw_addr_op ),
        .out_RT        ( rt_wr_op      )
    );

endmodule
//end of file.
```

## 5.12 Appendix L – SPU top (spu_top.sv)

```
import defines_pkg::*;
```

```
module spu_top
(
  input  logic              clk,
  input  logic              rst
);

  Opcodes        dec_opcode_ep;
  Opcodes        dec_opcode_op;
  logic [0:6]    dec_ra_addr_ep;
  logic [0:6]    dec_rb_addr_ep;
  logic [0:6]    dec_rc_addr_ep;
  logic [0:6]    dec_rt_addr_ep;
  logic [0:6]    dec_ra_addr_op;
  logic [0:6]    dec_rb_addr_op;
  logic [0:6]    dec_rc_addr_op;
  logic [0:6]    dec_rt_addr_op;
  logic [0:6]    dec_I7_ep;
  logic [0:7]    dec_I8_ep;
  logic [0:9]    dec_I10_ep;
  logic [0:15]   dec_I16_ep;
  logic [0:17]   dec_I18_ep;
  logic [0:6]    dec_I7_op;
  logic [0:7]    dec_I8_op;
  logic [0:9]    dec_I10_op;
  logic [0:15]   dec_I16_op;
  logic [0:17]   dec_I18_op;
  logic [0:6]    rf_addr_s1_ep;
  logic [0:6]    rf_addr_s2_ep;
  logic [0:6]    rf_addr_s3_ep;
  logic [0:6]    rf_addr_s4_ep;
  logic [0:6]    rf_addr_s5_ep;
  logic [0:6]    rf_addr_s6_ep;
  logic [0:6]    rf_addr_s1_op;
  logic [0:6]    rf_addr_s2_op;
  logic [0:6]    rf_addr_s3_op;
  logic [0:6]    rf_addr_s4_op;
  logic [0:6]    rf_addr_s5_op;
  logic [0:6]    rf_addr_s6_op;
  logic [0:2]    rf_idx_s1_ep;
  logic [0:2]    rf_idx_s2_ep;
  logic [0:2]    rf_idx_s3_ep;
  logic [0:2]    rf_idx_s4_ep;
  logic [0:2]    rf_idx_s5_ep;
logic [0:2]       rf_idx_s6_ep;
logic [0:2]       rf_idx_s1_op;
logic [0:2]       rf_idx_s2_op;
logic [0:2]       rf_idx_s3_op;
logic [0:2]       rf_idx_s4_op;
logic [0:2]       rf_idx_s5_op;
logic [0:2]       rf_idx_s6_op;
logic             flush;
Opcodes           opcode_ep;
Opcodes           opcode_op;
logic [0:6]       ra_addr_ep;
logic [0:6]       rb_addr_ep;
logic [0:6]       rc_addr_ep;
logic [0:6]       rt_addr_ep;
logic [0:6]       ra_addr_op;
logic [0:6]       rb_addr_op;
logic [0:6]       rc_addr_op;
logic [0:6]       rt_addr_op;
logic [0:6]       I7_ep;
logic [0:7]       I8_ep;
logic [0:9]       I10_ep;
logic [0:15]      I16_ep;
logic [0:17]      I18_ep;
logic [0:6]       I7_op;
logic [0:7]       I8_op;
logic [0:9]       I10_op;
logic [0:15]      I16_op;
logic [0:17]      I18_op;
logic [0:31]      eins1;
logic [0:31]      eins2;
logic [0:127]     ls_data_rd;
logic [0:127]     ls_data_wr;
logic [0:31]      ls_addr;
logic             ls_wr_en;
logic [0:31]      pc_dtof;
logic [0:31]      pc_ftod;
logic [0:1023]    cache_line;
logic             ins_wr_en;
logic             cache_wr;
logic             dec_stall;
logic             dep_stall;
logic [0:31]      pc_dec;
logic [0:31]      pc_dep;
```

```
local_store u_local_store (
    .clk(clk),
    .rst(rst),
    .ls_addr(ls_addr),
    .ls_data_wr(ls_data_wr),
    .ls_data_rd(ls_data_rd),
    .cache_wr(cache_wr),
    .cache_out(cache_line),
    .ls_wr_en(ls_wr_en)
);

fetch u_fetch (
    .clk(clk),
    .rst(rst),
    .cache_line(cache_line),
    .cache_wr(cache_wr),
    .dec_stall(dec_stall),
    .dep_stall(dep_stall),
    .branch_taken(flush),
    .pc_in(pc_dtof),
    .pc_out(pc_ftod),
    .eins1(eins1),
    .eins2(eins2)
);

decode u_decode (
    .clk(clk),
    .rst(rst),
    .dec_stall(dec_stall),
    .dep_stall(dep_stall),
    .eins1(eins1),
    .eins2(eins2),
    .flush(flush),
    .pc_in(pc_ftod),
    .pc_out(pc_dec),
    .opcode_ep(dec_opcode_ep),
    .opcode_op(dec_opcode_op),
    .ra_addr_ep(dec_ra_addr_ep),
    .rb_addr_ep(dec_rb_addr_ep),
    .rc_addr_ep(dec_rc_addr_ep),
    .rt_addr_ep(dec_rt_addr_ep),
    .ra_addr_op(dec_ra_addr_op),
    .rb_addr_op(dec_rb_addr_op),
    .rc_addr_op(dec_rc_addr_op),
    .rt_addr_op(dec_rt_addr_op),
    .in_I7e(dec_I7_ep),
    .in_I8e(dec_I8_ep),
    .in_I10e(dec_I10_ep),
    .in_I16e(dec_I16_ep),
    .in_I18e(dec_I18_ep),
    .in_I7o(dec_I7_op),
    .in_I8o(dec_I8_op),
    .in_I10o(dec_I10_op),
    .in_I16o(dec_I16_op),
    .in_I18o(dec_I18_op)
);

dependency u_dependency (
    .clk(clk),
    .rst(rst),
    .pc_in(pc_dec),
    .pc_out(pc_dep),
    .dec_opcode_ep(dec_opcode_ep),
    .dec_opcode_op(dec_opcode_op),
    .dec_ra_addr_ep(dec_ra_addr_ep),
    .dec_rb_addr_ep(dec_rb_addr_ep),
    .dec_rc_addr_ep(dec_rc_addr_ep),
    .dec_rt_addr_ep(dec_rt_addr_ep),
    .dec_ra_addr_op(dec_ra_addr_op),
    .dec_rb_addr_op(dec_rb_addr_op),
    .dec_rc_addr_op(dec_rc_addr_op),
    .dec_rt_addr_op(dec_rt_addr_op),
    .dec_I7_ep(dec_I7_ep),
    .dec_I8_ep(dec_I8_ep),
    .dec_I10_ep(dec_I10_ep),
    .dec_I16_ep(dec_I16_ep),
    .dec_I18_ep(dec_I18_ep),
    .dec_I7_op(dec_I7_op),
    .dec_I8_op(dec_I8_op),
    .dec_I10_op(dec_I10_op),
    .dec_I16_op(dec_I16_op),
    .dec_I18_op(dec_I18_op),
    .rf_addr_s1_ep(rf_addr_s1_ep),
    .rf_addr_s2_ep(rf_addr_s2_ep),
    .rf_addr_s3_ep(rf_addr_s3_ep),
    .rf_addr_s4_ep(rf_addr_s4_ep),
    .rf_addr_s5_ep(rf_addr_s5_ep),
    .rf_addr_s6_ep(rf_addr_s6_ep),
    .rf_addr_s1_op(rf_addr_s1_op),
    .rf_addr_s2_op(rf_addr_s2_op),
    .rf_addr_s3_op(rf_addr_s3_op),
    .rf_addr_s4_op(rf_addr_s4_op),
    .rf_addr_s5_op(rf_addr_s5_op),
    .rf_addr_s6_op(rf_addr_s6_op),
    .rf_idx_s1_ep(rf_idx_s1_ep),
    .rf_idx_s2_ep(rf_idx_s2_ep),
    .rf_idx_s3_ep(rf_idx_s3_ep),
    .rf_idx_s4_ep(rf_idx_s4_ep),
    .rf_idx_s5_ep(rf_idx_s5_ep),
    .rf_idx_s6_ep(rf_idx_s6_ep),
    .rf_idx_s1_op(rf_idx_s1_op),
    .rf_idx_s2_op(rf_idx_s2_op),
    .rf_idx_s3_op(rf_idx_s3_op),
    .rf_idx_s4_op(rf_idx_s4_op),
    .rf_idx_s5_op(rf_idx_s5_op),
    .rf_idx_s6_op(rf_idx_s6_op),
    .flush(flush),
    .opcode_ep(opcode_ep),
    .opcode_op(opcode_op),
    .ra_addr_ep(ra_addr_ep),
    .rb_addr_ep(rb_addr_ep),
    .rc_addr_ep(rc_addr_ep),
    .rt_addr_ep(rt_addr_ep),
    .ra_addr_op(ra_addr_op),
    .rb_addr_op(rb_addr_op),
    .rc_addr_op(rc_addr_op),
    .rt_addr_op(rt_addr_op),
    .I7_ep(I7_ep),
    .I8_ep(I8_ep),
    .I10_ep(I10_ep),
    .I16_ep(I16_ep),
    .I18_ep(I18_ep),
    .I7_op(I7_op),
    .I8_op(I8_op),
    .I10_op(I10_op),
    .I16_op(I16_op),
    .I18_op(I18_op),
    .dep_stall(dep_stall)
);

spu_pipes_top u_spu_pipes_top (
    .clk(clk),
.rst(rst),
.opcode_ep(opcode_ep),
.opcode_op(opcode_op),
.ra_addr_ep(ra_addr_ep),
.rb_addr_ep(rb_addr_ep),
.rc_addr_ep(rc_addr_ep),
.rt_addr_ep(rt_addr_ep),
.ra_addr_op(ra_addr_op),
.rb_addr_op(rb_addr_op),
.rc_addr_op(rc_addr_op),
.rt_addr_op(rt_addr_op),
.in_I7e(I7_ep),
.in_I8e(I8_ep),
.in_I10e(I10_ep),
.in_I16e(I16_ep),
.in_I18e(I18_ep),
.in_I7o(I7_op),
.in_I8o(I8_op),
.in_I10o(I10_op),
.in_I16o(I16_op),
.in_I18o(I18_op),
.rf_addr_s1_ep(rf_addr_s1_ep),
.rf_addr_s2_ep(rf_addr_s2_ep),
.rf_addr_s3_ep(rf_addr_s3_ep),
.rf_addr_s4_ep(rf_addr_s4_ep),
.rf_addr_s5_ep(rf_addr_s5_ep),
.rf_addr_s6_ep(rf_addr_s6_ep),
.rf_addr_s1_op(rf_addr_s1_op),
.rf_addr_s2_op(rf_addr_s2_op),
.rf_addr_s3_op(rf_addr_s3_op),
.rf_addr_s4_op(rf_addr_s4_op),
.rf_addr_s5_op(rf_addr_s5_op),
.rf_addr_s6_op(rf_addr_s6_op),
.rf_idx_s1_ep(rf_idx_s1_ep),
.rf_idx_s2_ep(rf_idx_s2_ep),
.rf_idx_s3_ep(rf_idx_s3_ep),
.rf_idx_s4_ep(rf_idx_s4_ep),
.rf_idx_s5_ep(rf_idx_s5_ep),
.rf_idx_s6_ep(rf_idx_s6_ep),
.rf_idx_s1_op(rf_idx_s1_op),
.rf_idx_s2_op(rf_idx_s2_op),
.rf_idx_s3_op(rf_idx_s3_op),
.rf_idx_s4_op(rf_idx_s4_op),
.rf_idx_s5_op(rf_idx_s5_op),
        .rf_idx_s6_op(rf_idx_s6_op),
        .flush(flush),
        .ls_data_rd(ls_data_rd),
        .ls_data_wr(ls_data_wr),
        .ls_addr(ls_addr),
        .ls_wr_en(ls_wr_en),
        .cache_wr(cache_wr),
        .PC_in(pc_dep),
        .PC_out(pc_dtof)
    );

endmodule
//end of file.
```

## 5.13 Appendix M - Simulation Script (run.py)

```
#!/usr/bin/env python2

import os
import sys
import subprocess

from subprocess import call

def gen_syn_script(name, clk):
    with open('cfg/runsynth_template.tcl', 'r') as file :
        filedata = file.read()

    filedata = filedata.replace('{ARG0}', name)
    filedata = filedata.replace('{CLK}' , str(clk))

    with open('synth/runsynth.tcl', 'w') as file:
        file.write(filedata)

    return "synth/" + name + ".txt"

def gen_sim_script(name):
    with open('cfg/runsim_template', 'r') as file :
        filedata = file.read()

    filedata = filedata.replace('{ARG0}', name)

    with open('sim/runsim', 'w') as file:
        file.write(filedata)
    os.chmod('sim/runsim', 0777)

    return "sim/" + name + ".txt"

def run_syn(name, clk):
    lname = gen_syn_script(name, clk)
    os.chdir("synth")
    subprocess.call(["dc_shell", "-f", "runsynth.tcl"])
    os.chdir("..")

def run_sim(mode, ins_file):
    lname = gen_sim_script(name)
    os.chdir("cfg")
    if(mode == "1"):
        subprocess.call(["./sasm.py", "even_ins.sasm", "even_ins_file.txt", mode])
        subprocess.call(["./sasm.py", "odd_ins.sasm", "odd_ins_file.txt", mode])
    else:
        subprocess.call(["./sasm.py", ins_file, "spu_mcode.bin", mode])
    os.chdir("..")
    os.chdir("sim")
    if(mode == "1"):
        subprocess.call(["cp", "-rf", "../cfg/even_ins_file.txt", "."])
        subprocess.call(["cp", "-rf", "../cfg/odd_ins_file.txt", "."])
    subprocess.call(["cp", "-rf", "../cfg/ls_load_file.txt", "."])
    subprocess.call(["./runsim"])
    os.chdir("..")

def print_usage():
    print sys.argv[0] + " flow [Top Module] [Clock]"
    print "flow:  sim, syn"
    print "<sim> target expects top module name only"
    print "<syn> target requires both top module and clock"

print "\n!!Welcome to SPU Mini Simulation Script!!"

if(len(sys.argv) == 1):
    print "Invalid number of arguments"
    print_usage()
    sys.exit(0)
else:
    flow = sys.argv[1]

if flow == "syn":
    if(len(sys.argv) < 4):
        print "Invalid number of arguments"
        print_usage()
        sys.exit(0)
    name = sys.argv[2]
    clk  = float(sys.argv[3])
    print "Running Synthesis on module " + name + "with clock period: " + str(clk)
    run_syn(name, clk)
elif flow == "sim":
    if(len(sys.argv) < 4):
        print "Invalid number of arguments"
        print_usage()
        sys.exit(0)
    name = sys.argv[2]
    ins_file = sys.argv[3]
    if(name == "sp_pipes_tb"):
        mode = "1"
    else:
        mode = "0"
    print "Running Simulation on module " + name
    run_sim(mode, ins_file)
elif flow == "clean":
    print "Cleaning Temp Files!"
    subprocess.call(["rm", "-rf", "synth/*", "sim/*"])
else:
    print "Invalid Argument: " + flow

print "!!SPU Mini Simulation Script finished sucessfully!!"
```