VE 370

Project 2

Single cycle & pipelined implementation of MIPS computer in Verilog

Team Member:

Xing Hua	5127169006
Ou Xinyue	5133709045
Qian Junqi	5133709043

Date: Nov 25, 2015

I. Introduction & Requirements

In lab #2 we are to implement one single cycle and one pipeline processor. Both processors should be able to take no more than 32 MIPS instructions in machine code and compute a correct result. Such result should be available from both simulation and FPGA implementation.

Single cycle processor has no special requirement. Pipeline processor should be able to handle EX & MEM data hazard, load-word data hazard and control hazard. Hazard detection and handling should go through specially designed forwarding circuit and hazard detection units.

Simulation and FPGA implementation should be able to output the 31 register content in register file. Simulation result and FPGA implementation result should be identical to each other.

II. Detailed Design

(a) Single Cycle

Single cycle processor design strictly follows textbook Fig. 4.24, as shown below.

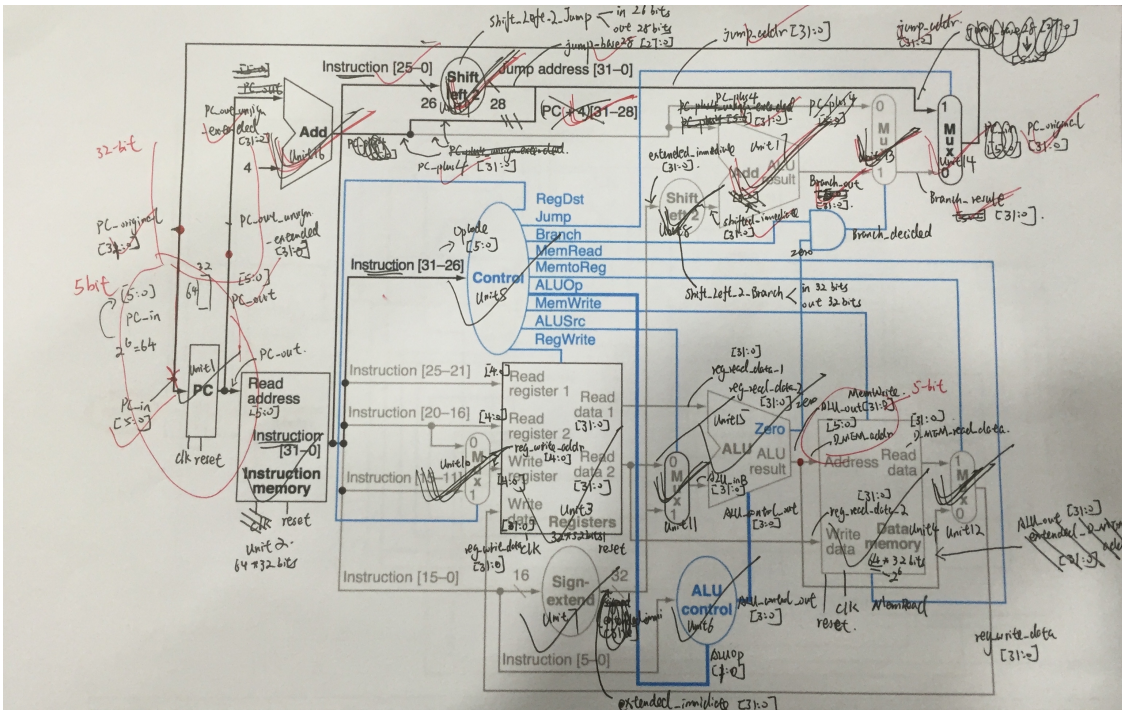

Fig 1. Single Cycle implementation diagram.
[background is Fig 4.24 from David Patterson, *Computer Organization and Design, 4th ed*]

There are a few practical limitations. This has caused minor modifications in single cycle design. The most prominent modification is PC signal width.

Program counter has an input bus width of 32 bits. So PC should be able to store count from 0 to 2^32 - 1. However, such large number is not practically possible in Xilinx FPGA implementation. Besides, out I-MEM will carry less than 64 instructions. So 6 bits would be enough. So right before address signal bus reaches PC, we take the lowest 6 bits and feed to PC. And at PC output, we feed the 6-bit to I-MEM, while unsign-extending the 6-bit PC address to the usual 4 bit and feed that to ALU for the +4 and possible branch and jump operation.

(b) Pipeline

There are no complete pipeline implementation digram at hand. So we are combining several diagrams from textbook as blue print for pipeline implementation.

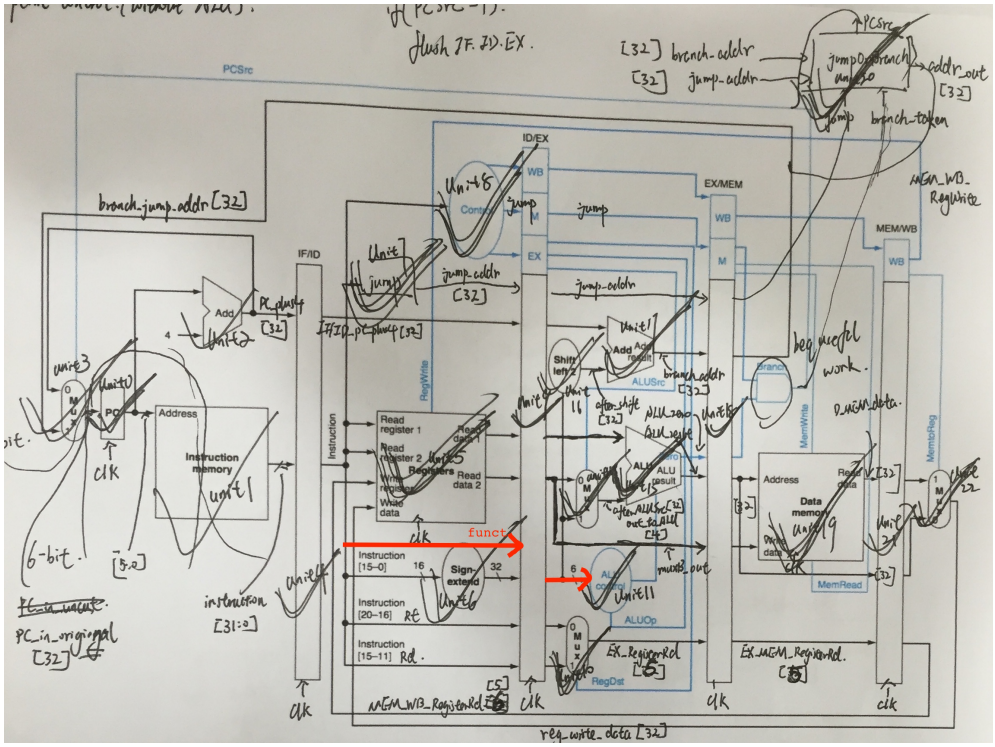

Fig 2. Pipeline basic implementation diagram, with a few discernible modifications.
[background is Fig 4.51 from David Patterson, *Computer Organization and Design, 4th ed*]

A few major modifications are made into Fig 2 layout.

1. ALU control in EX stage requires the last 6 bits of instruction as input. There are no mentioning of this wire in ID/EX stage register. So we adding an additional input into ID/EX register, which is named funct. funct contains 6 bits, and is assigned the value of last 6 bits from IF/ID instruction signal.
2. Jump instruction is not handled in this diagram. So we calculated jump address as old_PC[31:28] concatenated with 26 bits jump field [27:2] in jump instruction and lastly concatenated with 00.

Such jump address is calculated in ID stage. Jump control signal and jump address will be propagated through ID/EX stage reg, EX/MEM stage reg into MEM stage.

3. In memory stage, a newly designed module "JumpOrBranch" will discern if there is a branch taken or if there is a jump instruction arriving at MEM stage. If either of the two is true, JumpOrBranch will set PCSrc to 1, diverting PC input to the branch destination address or jump destination address that is readily available at MEM stage.

Such is the basic pipeline implementation structure. Then we will construct forwarding unit and hazard detection units based on Fig 3.

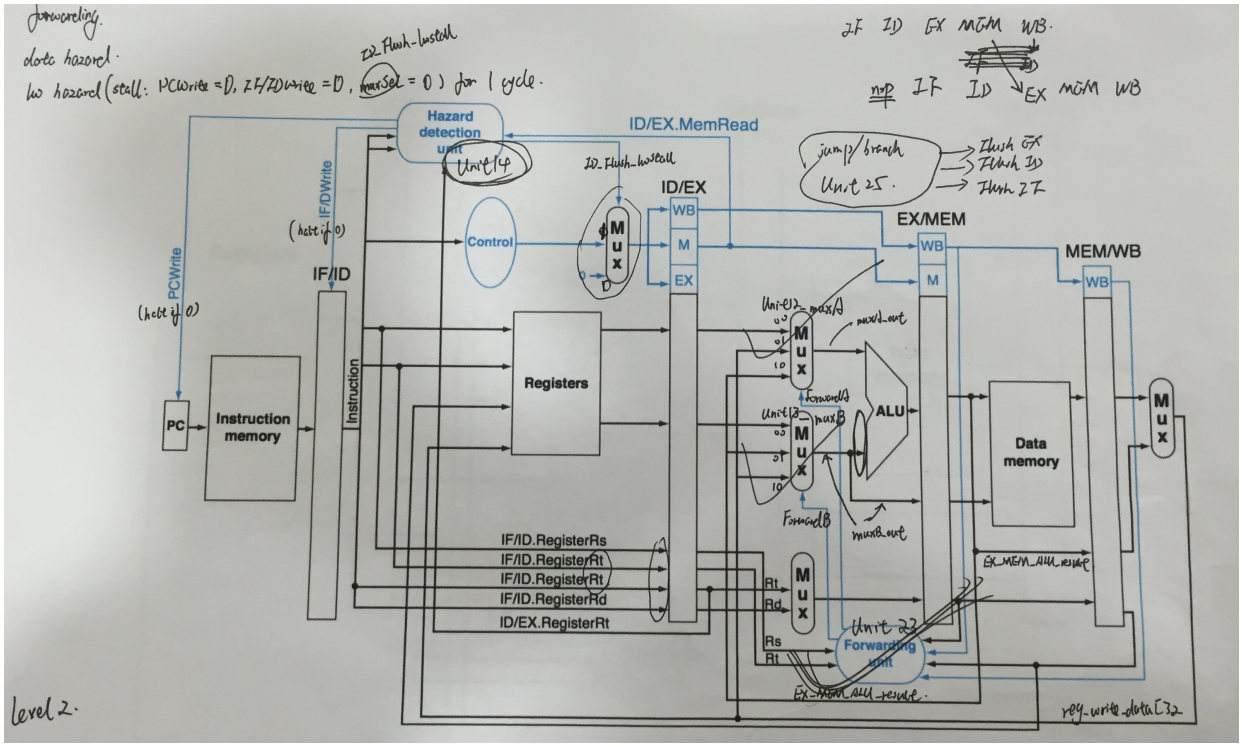

Fig 3. Pipeline forwarding & hazard detection design.
[background is Fig 4.60 from David Patterson, *Computer Organization and Design, 4th ed*]

In this digram, "Forwarding Unit" resolves

- EX data hazard
- MEM data hazard
- double data hazard (EX and MEM forwarding at the same time)

"Hazard Detection Unit" resolves

- load-word data hazard: stall one cycle if such hazard is detected.

We have added an additional jump_branch_hazard_detection unit to deal with the flush of IF/ID, ID/EX and EX/MEM stage registered when either a branch-taken or jump instruction pops up at MEM stage.

That concludes the top-level pipeline design and basic hazard detection & forwarding schematics. Now Fig 3 to 6 shows the detailed I/O design of all 4 stage registers.

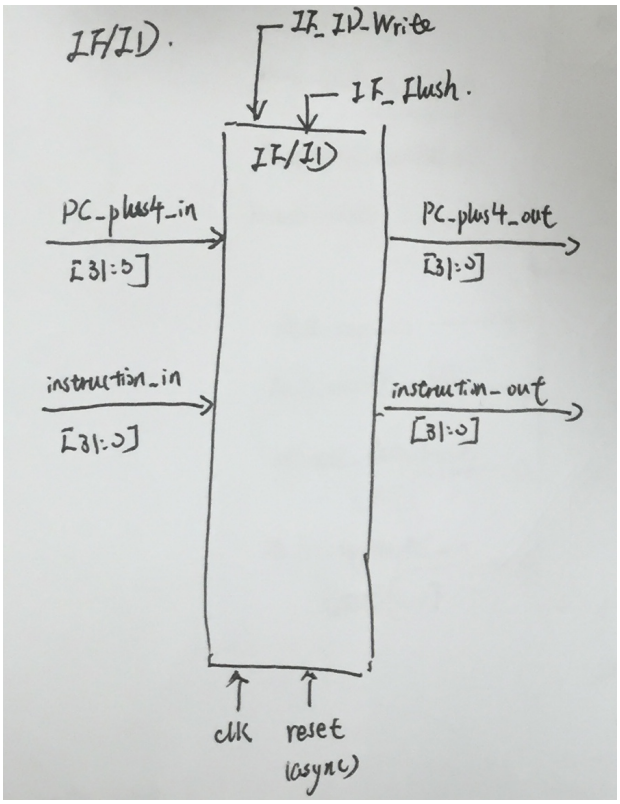

Fig 4. IF/ID stage register I/O design

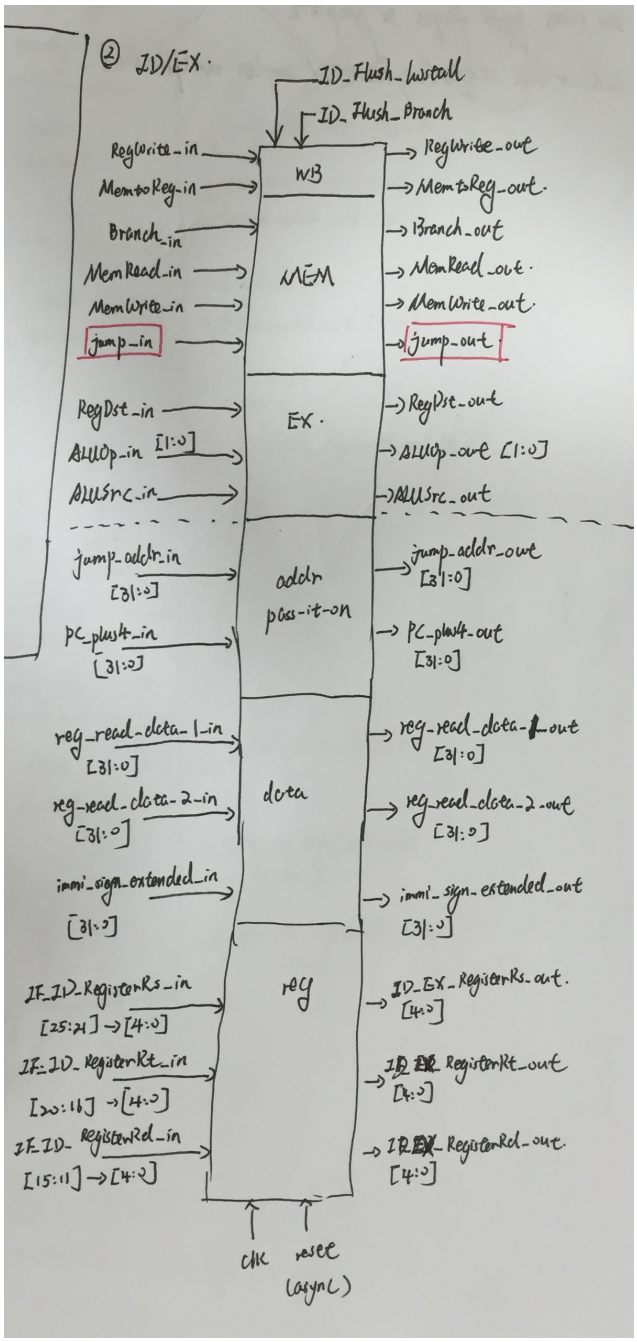

Fig 5. ID/EX stage register I/O design

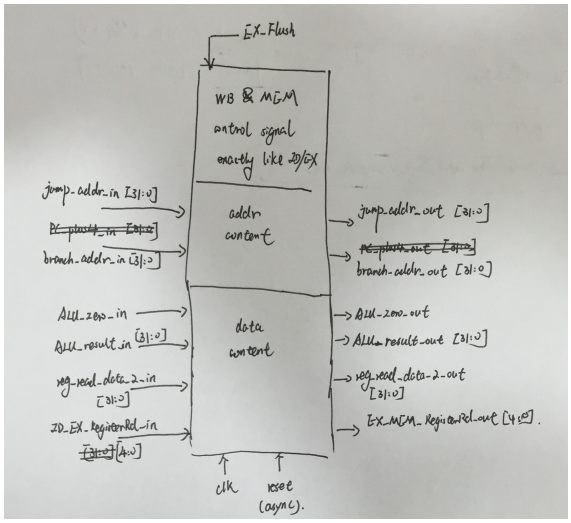

Fig 6. EX/MEM stage register I/O design

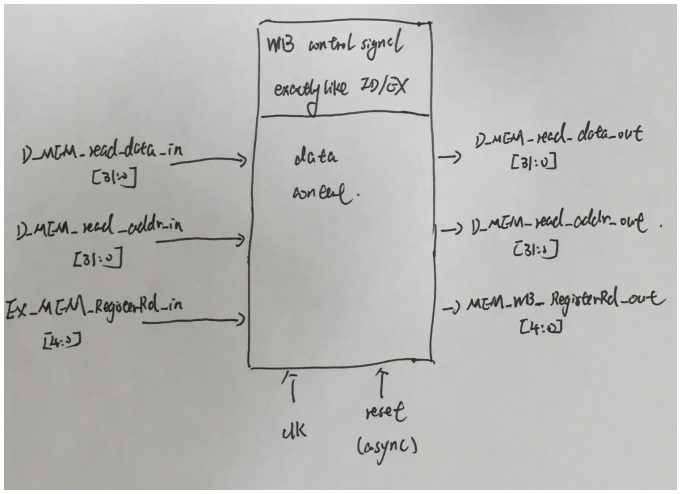

Fig 7. MEM/WB stage register I/O design

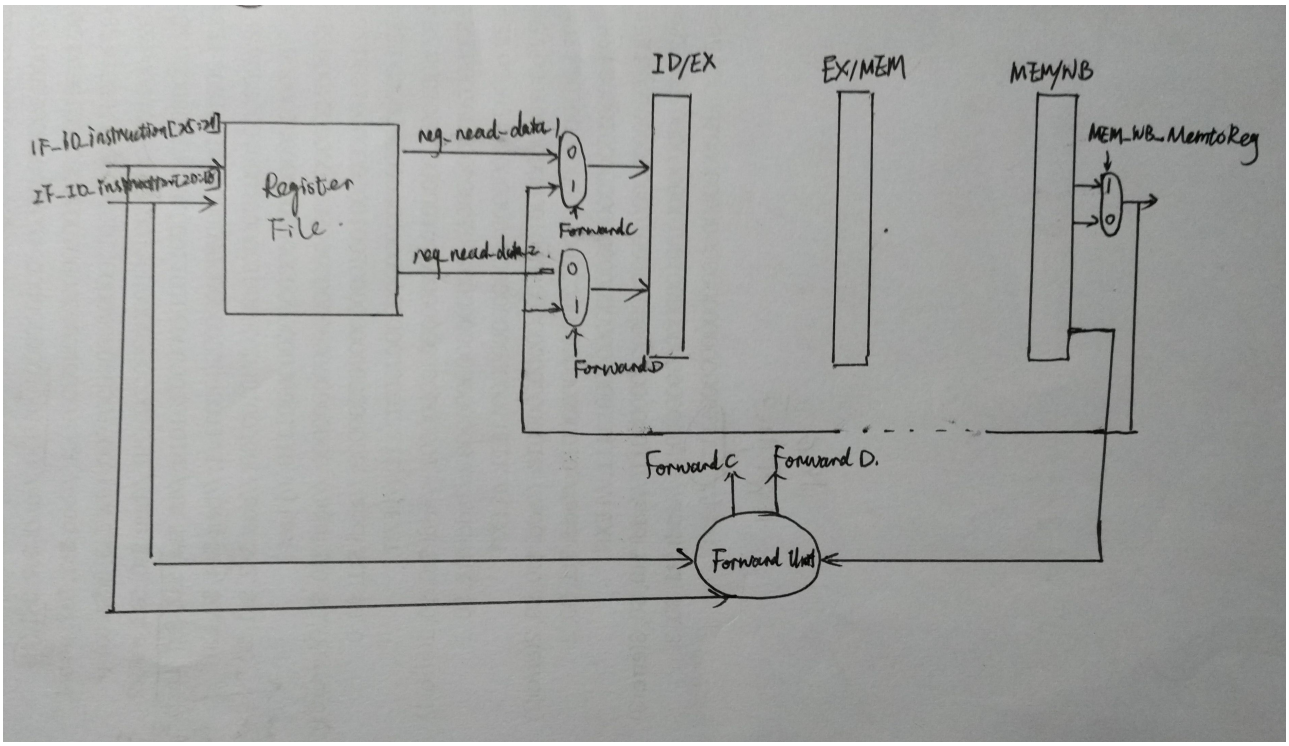

Fig 8. Extra Forwarding Unit in ID stage

The additional MUXes in Fig 7 are mainly to resolve the problem when the register is read before it is written back into the register file. When an instruction need to write back has gone into the WB stage, it computed the value to write back but it still doesn't write into the register file until the next posedge. Without the extra forwarding units we can forward it to EX stage, but we can't deal with the case when the ID stage requires the data. We can't feed the data to the EX stage because it is one instruction before the requiring instruction in the ID stage. Nor can we feed to the EX stage in the next cycle because the value to write back has gone out of the pipeline in the next cycle. With the

additional forwarding unit, we can resolve the conflicts of requesting the data in ID stage and the writing back in WB stage.

We have designed a special mechanism to stop the clock for us to change register file address input in order to "peek into" register content at register file data read port. To achieve such objective, we are giving different clock signals into register file and all other time-dependent modules (at 1 Hz). Two clocks function the same during normal processor operations. But when we flip a switch on FPGA to indicate we intend to read register content, all other clock will freeze to zero while register file clock speeds up to 500 Hz. This enables us to read register file content (read on falling-edge) almost instantaneously as we select which register to look at with a 5-switch combination.

The 32-bit register file content is truncated into 16 bits, assigned to 4 HEX numbers. Each HEX number correspond to one SSD output. Borrowing from the clock-divider and the technique to display different numbers on 4 FPGA SSDs, we are now able to output any register file content of our choosing (with a total of 32 registers to choose from).

And when processor is running in normal operating mode, FPGA SSD outputs PC content. So we'd know which instruction is processor currently working on.

III. Test & Result

For single cycle:

Single cycle is not difficult nor error-prone. So we are only running the program given in instruction manual [credit to VE370 FA2015 TA Group].

```
     addi $t0, $zero, 32
     addi $t1, $zero, 55
     add $s0, $t0, $t1
     sub $s1, $t0, $t1
     and $s2, $t0, $t1
     or $s3, $t0, $t1
LOOP:slt $s4, $t0, $t1
     beq $s4, $zero, EXIT
     add $t2, $t0, $t1
```

```
     add $t2, $t2, $t1
     add $t3, $t2, $t0
     lw $s5, 4($zero)
     add $s6, $t2, $s5
     add $s7, $s5, $t3
EXIT:sw $s5, 8($zero)
     j LOOP
```

Theoretically, we should have

```
[$s0] = 00000057        [$t1] = 00000037
[$s1] = fffffe9         [$t2] = 0000008e
[$s2] = 00000020        [$t3] = 000000ae
[$s3] = 00000037        [$t4] = 00000000
[$s4] = 00000001        [$t5] = 00000000
[$s5] = 00000000        [$t6] = 00000000
[$s6] = 0000008e        [$t7] = 00000000
[$s7] = 000000ae        [$t8] = 00000000
[$t0] = 00000020        [$t9] = 00000000
```

And the simulation, as shown below, gives identical results. FPGA implementation also shows identical calculation result. **And please note all following results are taken at `clock == 0`.**

```
Time:            0
[$s0] = 00000000        [$s6] = 00000000        [$t4] = 00000000
[$s1] = 00000000        [$s7] = 00000000        [$t5] = 00000000
[$s2] = 00000000        [$t0] = 00000000        [$t6] = 00000000
[$s3] = 00000000        [$t1] = 00000000        [$t7] = 00000000
[$s4] = 00000000        [$t2] = 00000000        [$t8] = 00000000
[$s5] = 00000000        [$t3] = 00000000        [$t9] = 00000000
Time:            1
[$s0] = 00000000        [$s4] = 00000000        [$t0] = 00000020
[$s1] = 00000000        [$s5] = 00000000        [$t1] = 00000000
[$s2] = 00000000        [$s6] = 00000000        [$t2] = 00000000
[$s3] = 00000000        [$s7] = 00000000        [$t3] = 00000000
```

```
[$t4] = 00000000        [$t6] = 00000000        [$t8] = 00000000
[$t5] = 00000000        [$t7] = 00000000        [$t9] = 00000000
Time:           2
[$s0] = 00000000        [$s6] = 00000000        [$t4] = 00000000
[$s1] = 00000000        [$s7] = 00000000        [$t5] = 00000000
[$s2] = 00000000        [$t0] = 00000020        [$t6] = 00000000
[$s3] = 00000000        [$t1] = 00000037        [$t7] = 00000000
[$s4] = 00000000        [$t2] = 00000000        [$t8] = 00000000
[$s5] = 00000000        [$t3] = 00000000        [$t9] = 00000000
Time:           3
[$s0] = 00000057        [$s6] = 00000000        [$t4] = 00000000
[$s1] = 00000000        [$s7] = 00000000        [$t5] = 00000000
[$s2] = 00000000        [$t0] = 00000020        [$t6] = 00000000
[$s3] = 00000000        [$t1] = 00000037        [$t7] = 00000000
[$s4] = 00000000        [$t2] = 00000000        [$t8] = 00000000
[$s5] = 00000000        [$t3] = 00000000        [$t9] = 00000000
Time:           4
[$s0] = 00000057        [$s6] = 00000000        [$t4] = 00000000
[$s1] = ffffffe9        [$s7] = 00000000        [$t5] = 00000000
[$s2] = 00000000        [$t0] = 00000020        [$t6] = 00000000
[$s3] = 00000000        [$t1] = 00000037        [$t7] = 00000000
[$s4] = 00000000        [$t2] = 00000000        [$t8] = 00000000
[$s5] = 00000000        [$t3] = 00000000        [$t9] = 00000000
Time:           5
[$s0] = 00000057        [$s6] = 00000000        [$t4] = 00000000
[$s1] = ffffffe9        [$s7] = 00000000        [$t5] = 00000000
[$s2] = 00000020        [$t0] = 00000020        [$t6] = 00000000
[$s3] = 00000000        [$t1] = 00000037        [$t7] = 00000000
[$s4] = 00000000        [$t2] = 00000000        [$t8] = 00000000
[$s5] = 00000000        [$t3] = 00000000        [$t9] = 00000000
Time:           6
[$s0] = 00000057        [$s4] = 00000000        [$t0] = 00000020
[$s1] = ffffffe9        [$s5] = 00000000        [$t1] = 00000037
[$s2] = 00000020        [$s6] = 00000000        [$t2] = 00000000
[$s3] = 00000037        [$s7] = 00000000        [$t3] = 00000000
```

```
[$t4] = 00000000        [$t6] = 00000000        [$t8] = 00000000
[$t5] = 00000000        [$t7] = 00000000        [$t9] = 00000000
Time:           7
[$s0] = 00000057        [$s6] = 00000000        [$t4] = 00000000
[$s1] = fffffe9         [$s7] = 00000000        [$t5] = 00000000
[$s2] = 00000020        [$t0] = 00000020        [$t6] = 00000000
[$s3] = 00000037        [$t1] = 00000037        [$t7] = 00000000
[$s4] = 00000001        [$t2] = 00000000        [$t8] = 00000000
[$s5] = 00000000        [$t3] = 00000000        [$t9] = 00000000
Time:           8
[$s0] = 00000057        [$s6] = 00000000        [$t4] = 00000000
[$s1] = fffffe9         [$s7] = 00000000        [$t5] = 00000000
[$s2] = 00000020        [$t0] = 00000020        [$t6] = 00000000
[$s3] = 00000037        [$t1] = 00000037        [$t7] = 00000000
[$s4] = 00000001        [$t2] = 00000000        [$t8] = 00000000
[$s5] = 00000000        [$t3] = 00000000        [$t9] = 00000000
Time:           9
[$s0] = 00000057        [$s6] = 00000000        [$t4] = 00000000
[$s1] = fffffe9         [$s7] = 00000000        [$t5] = 00000000
[$s2] = 00000020        [$t0] = 00000020        [$t6] = 00000000
[$s3] = 00000037        [$t1] = 00000037        [$t7] = 00000000
[$s4] = 00000001        [$t2] = 00000057        [$t8] = 00000000
[$s5] = 00000000        [$t3] = 00000000        [$t9] = 00000000
Time:          10
[$s0] = 00000057        [$s6] = 00000000        [$t4] = 00000000
[$s1] = fffffe9         [$s7] = 00000000        [$t5] = 00000000
[$s2] = 00000020        [$t0] = 00000020        [$t6] = 00000000
[$s3] = 00000037        [$t1] = 00000037        [$t7] = 00000000
[$s4] = 00000001        [$t2] = 0000008e        [$t8] = 00000000
[$s5] = 00000000        [$t3] = 00000000        [$t9] = 00000000
Time:          11
[$s0] = 00000057        [$s4] = 00000001        [$t0] = 00000020
[$s1] = fffffe9         [$s5] = 00000000        [$t1] = 00000037
[$s2] = 00000020        [$s6] = 00000000        [$t2] = 0000008e
[$s3] = 00000037        [$s7] = 00000000        [$t3] = 000000ae
```

```
[$t4] = 00000000        [$t6] = 00000000        [$t8] = 00000000
[$t5] = 00000000        [$t7] = 00000000        [$t9] = 00000000
Time:           12
[$s0] = 00000057        [$s6] = 00000000        [$t4] = 00000000
[$s1] = ffffffe9        [$s7] = 00000000        [$t5] = 00000000
[$s2] = 00000020        [$t0] = 00000020        [$t6] = 00000000
[$s3] = 00000037        [$t1] = 00000037        [$t7] = 00000000
[$s4] = 00000001        [$t2] = 0000008e        [$t8] = 00000000
[$s5] = 00000000        [$t3] = 000000ae        [$t9] = 00000000
Time:           13
[$s0] = 00000057        [$s6] = 0000008e        [$t4] = 00000000
[$s1] = ffffffe9        [$s7] = 00000000        [$t5] = 00000000
[$s2] = 00000020        [$t0] = 00000020        [$t6] = 00000000
[$s3] = 00000037        [$t1] = 00000037        [$t7] = 00000000
[$s4] = 00000001        [$t2] = 0000008e        [$t8] = 00000000
[$s5] = 00000000        [$t3] = 000000ae        [$t9] = 00000000
Time:           14
[$s0] = 00000057        [$s6] = 0000008e        [$t4] = 00000000
[$s1] = ffffffe9        [$s7] = 000000ae        [$t5] = 00000000
[$s2] = 00000020        [$t0] = 00000020        [$t6] = 00000000
[$s3] = 00000037        [$t1] = 00000037        [$t7] = 00000000
[$s4] = 00000001        [$t2] = 0000008e        [$t8] = 00000000
[$s5] = 00000000        [$t3] = 000000ae        [$t9] = 00000000
Time:           15
[$s0] = 00000057        [$s6] = 0000008e        [$t4] = 00000000
[$s1] = ffffffe9        [$s7] = 000000ae        [$t5] = 00000000
[$s2] = 00000020        [$t0] = 00000020        [$t6] = 00000000
[$s3] = 00000037        [$t1] = 00000037        [$t7] = 00000000
[$s4] = 00000001        [$t2] = 0000008e        [$t8] = 00000000
[$s5] = 00000000        [$t3] = 000000ae        [$t9] = 00000000
Time:           16
[$s0] = 00000057        [$s4] = 00000001        [$t0] = 00000020
[$s1] = ffffffe9        [$s5] = 00000000        [$t1] = 00000037
[$s2] = 00000020        [$s6] = 0000008e        [$t2] = 0000008e
[$s3] = 00000037        [$s7] = 000000ae        [$t3] = 000000ae
```

```
[$t4] = 00000000        [$t6] = 00000000        [$t8] = 00000000
[$t5] = 00000000        [$t7] = 00000000        [$t9] = 00000000
```

For pipeline:

Identical code from single cycle is also run on the pipeline simulation and FPGA implementation. **Final** result is shown below.

```
Time:           22
[$s0] = 00000057        [$s6] = 0000008e        [$t4] = 00000000
[$s1] = fffffe9         [$s7] = 000000ae        [$t5] = 00000000
[$s2] = 00000020        [$t0] = 00000020        [$t6] = 00000000
[$s3] = 00000037        [$t1] = 00000037        [$t7] = 00000000
[$s4] = 00000001        [$t2] = 0000008e        [$t8] = 00000000
[$s5] = 00000000        [$t3] = 000000ae        [$t9] = 00000000
```

Since pipeline has special requirements on hazard detection & elimination, we will raise one test case for each hazard and see if our implementation gives the correct result.

- **EX data hazard**

```
MIPS code:
addi $t0, $zero, 32
addi $t1,$t0, 16    #forward to rs
add $t2,$zero,$t1   #forward to rt
add $t3,$t2,$t1     #forward to rt and rs

Translated to machine code:
0x20080020
0x21090010
0x00095020
0x01495820

And we can expect result:
[$t0] = 0x20;     [$t1] = 0x30;     [$t2] = 0x30;     [$t3] = 0x60;
```

And the simulation gives us:

```
Time:           19
[$s0] = 00000000        [$s6] = 00000000        [$t4] = 00000000
[$s1] = 00000000        [$s7] = 00000000        [$t5] = 00000000
[$s2] = 00000000        [$t0] = 00000020        [$t6] = 00000000
[$s3] = 00000000        [$t1] = 00000030        [$t7] = 00000000
[$s4] = 00000000        [$t2] = 00000030        [$t8] = 00000000
[$s5] = 00000000        [$t3] = 00000060        [$t9] = 00000000
```

Theoretical and simulation results are a match. So we are sure the pipeline processor can handle EX data hazard.

- **MEM data hazard**

MIPS code:

```
addi $t0,$zero, 8
addi $t1,$zero, 4
addi  $t2,$t0,2     # forward to rs
add $t3,$t0,$t1     # forward to rs and rt, wb and ex
add $t4,$t2,$t3;    # forward to rs and rt, ex and mem
```

Translated to machine code:

```
0x20080008
0x20090004
0x210a0002
0x01095820
0x014b6020
```

And we can expect result:

```
[$t0] = 0x8;      [$t2] = 0xa;      [$t4] = 0x16;
[$t1] = 0x4;      [$t3] = 0xc;
```

And the simulation gives us:

```
Time:           10
[$s0] = 00000000        [$s3] = 00000000        [$s6] = 00000000
[$s1] = 00000000        [$s4] = 00000000        [$s7] = 00000000
[$s2] = 00000000        [$s5] = 00000000        [$t0] = 00000008
```

[$t1] = 00000004 [$t4] = 00000016 [$t7] = 00000000

[$t2] = 0000000a [$t5] = 00000000 [$t8] = 00000000

[$t3] = 0000000c [$t6] = 00000000 [$t9] = 00000000

Theoretical and simulation results are a match. So we are sure the pipeline processor can handle MEM data hazard.

- **Double data hazard**

MIPS code:

```
addi $t0, $zero, 64
addi $t0, $t0, 8
add $t1, $t0, $zero
addi $t2,$zero,48
addi $t2,$t2, 48
add $t3,$zero,$t2
```

Translated to machine code:

```
0x20080040
0x21080008
0x01004820
0x200a0030
0x214a0030
0x000a5820
```

And we can expect result:
[$t0] = 0x48; [$t1] = 0x48; [$t2] = 0x60; [$t3] = 0x60;

And the simulation gives us:
Time: 10
[$s0] = 00000000 [$s3] = 00000000 [$s6] = 00000000

[$s1] = 00000000 [$s4] = 00000000 [$s7] = 00000000

[$s2] = 00000000 [$s5] = 00000000 [$t0] = 00000048

```
[$t1] = 00000048        [$t4] = 00000000        [$t7] = 00000000
[$t2] = 00000060        [$t5] = 00000000        [$t8] = 00000000
[$t3] = 00000060        [$t6] = 00000000        [$t9] = 00000000
```

Theoretical and simulation results are a match. So we are sure the pipeline processor can handle double data hazard.

- **Load-use data hazard**

MIPS code:

```
# change D-MEM[0] to 6
lw $t0,0($zero)
add $t1, $t0,$zero
addi $t2, $t1,8
lw $t3,0($zero)
lw $t4,0($zero)
```

Translated to machine code:

```
0x8c080000
0x01004820
0x212a0008
0x8c0b0000
0x8c0c0000
```

And we can expect result:

```
[$t0] = 0x6;            [$t2] = 0xe;            [$t6] = 0x6;
[$t1] = 0x6;            [$t3] = 0x6;
```

And the simulation gives us:

```
Time:           10
[$s0] = 00000000        [$s3] = 00000000        [$s6] = 00000000
[$s1] = 00000000        [$s4] = 00000000        [$s7] = 00000000
[$s2] = 00000000        [$s5] = 00000000        [$t0] = 00000006
```

```
[$t1] = 00000006        [$t4] = 00000006        [$t7] = 00000000
[$t2] = 0000000e        [$t5] = 00000000        [$t8] = 00000000
[$t3] = 00000006        [$t6] = 00000000        [$t9] = 00000000
```

Theoretical and simulation results are a match. So we are sure the pipeline processor can handle lw data hazard.

- **Control hazard (Branches & Jumps)**

MIPS code:

```
beq $s0,$zero, Branch1
addi $s1, $zero,8
Branch1:
beq $s1,$zero, Branch2
add $s1,$s1,$s1
Branch2:
addi $s1,$zero,4
j Jump1
addi $s1,$s1,4
addi $s1,$s1,4
addi $s1,$s1,4
addi $s1,$s1,4
Jump1:
addi $s1,$zero,1
```

Translated to machine code:

```
0x12000001
0x20110008
0x12200001
0x02318820
0x0800000f
0x00400060
0x22310004
0x22310004
```

0x22310004

0x22310004

0x20110001

And we can expect result:
[$s1] = 0x1;

And the simulation gives us:

Time: 22

[$s0] = 00000000	[$s6] = 00000000	[$t4] = 00000000
[$s1] = 00000001	[$s7] = 00000000	[$t5] = 00000000
[$s2] = 00000000	[$t0] = 00000000	[$t6] = 00000000
[$s3] = 00000000	[$t1] = 00000000	[$t7] = 00000000
[$s4] = 00000000	[$t2] = 00000000	[$t8] = 00000000
[$s5] = 00000000	[$t3] = 00000000	[$t9] = 00000000

Theoretical and simulation results are a match. So we are sure the pipeline processor can handle lw data hazard.

IV. Conclusion

We have successfully implemented and run our single cycle processor and pipeline processor. Our simulation correctly characterized the expected performance of the processors, and our implementation on the FPGA board exhibited correct results including the stall and flush mechanism.

The only problem we encountered during the simulation is that the register data cannot be read in time. The reason is that the data will not be read until the falling edge of clock. Therefore we replaced the falling-edge synchronized data reading with unsynchronized data reading, and then the data can be read from the register in time, which enables the simulation to perform normally.

During the period of this project, we have employed a coding style that is efficient and error-prone for coding in teams. The coding style is encapsulation. One coder defines a top level structure and input/output of every second level module it employs, while another coder programs the second level module based on input/ output requirements. Such encapsulation methods we borrowed from C ++ coding experiences ensure everyone know exactly where the other coder takes over. Encapsulation has helped facilitate team coding efforts and help us reduce bug rate.

One more important lesson we learned from the project is the necessity to program buffer into our processor structures, rather than only doing what is minimally necessary to run the sample MIPS code. For example, the sample code had 16 instructions, so we restricted program counter I/O to 6 bits. With 2 bit byte offset, we only have enough room to read 16 instructions. But the final demo code had 32 MIPS instructions. That has led to a last-minute change of all wires related to PC width. The ethos of the story is to always leave extra room during the initial programming so we wouldn’t be surprised when specification changes.

V. Appendix

V.I Single-Cycle code (loaded with demo instructions; for simulation)

```
`timescale 1ns / 1ps
// p2_single_cycle.v
// ver 9.0
// VE370 Project #2

// Due:
// Report & Source code due Nov 27 23:00 through Sakai
// Demonstration is due by 4:00pm, November 27, 2015

// Created by Hua Xing, Junqi Qian, Xinyue Ou on 11/12/15
// Copyright (c) 2015. All rights reserved.

// Version History
// ver 1.0: single cycle highest-level program; module I/O defined
// ver 2.0: added clock-divider and SSD output. SSD outputs PC or reg file content.
// ver 3.0: all module implementation complete. Passed Xilinx synthesis. No error. Tons of warnings.
// ver 4.0: debug complete. FPGA implementation complete.
// ver 5.0: simulation test bench complete
// ver 6.0: improve test bench
// ver 7.0: test bench running time lifted
// ver 8.0: expanded PC width to 7 bits (5 bit + 2 bit offset). I-MEM now supports 2^5 = 32 lines.
// ver 9.0: I-MEM loads TA demo code
// ver 11.1: code for simulation
module single_cycle (clkFast, reset, SwitchSelector, switchRun, Cathode, AN, LEDIndicator, reg_read_data_1);
      input clkFast, reset;         // clkFast (5m[6] Hz) feeds clock divider
      input switchRun;              // decide run(1) or check reg file(0)
      input [4:0] SwitchSelector;   // select $0 to $31 for output
      output [6:0] Cathode;         // SSD
      output [3:0] AN;
      output LEDIndicator;          // output clkNormal for user reference
      output [31:0] reg_read_data_1; // moved here for simulation

      // PC signals
      wire [7:0] PC_in, PC_out;
      wire [31:0] PC_original, PC_out_unsign_extended, PC_plus4;
      // I-MEM signals
      wire [31:0] instruction;
      // Register File signals
      wire [4:0] reg_write_addr;
      wire [31:0] reg_write_data, reg_read_data_2; // reg_read_data_1 moved to output for simulation
      // D-MEM signals
      wire [7:0] D_MEM_addr;
      wire [31:0] D_MEM_read_data;
      // control signals
      wire RegDst, Jump, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite;
      wire [1:0] ALUOp;
      wire [3:0] ALU_control_out;
      // branch
      wire [31:0] extended_immidiate;
      wire [31:0] shifted_immidiate;
```

```
      wire [31:0] Branch_out;
      wire [31:0] Branch_result;
      wire Branch_decided, zero;
      // jump
      wire [27:0] jump_base28;
      wire [31:0] jump_addr;
      // ALU
      wire [31:0] ALU_inB, ALU_out;
      // SSD display & clock slow-down
      wire clkSSD, clkNormal, clkRF, clk;
            // clkFast: 5m Hz
            // clkSSD: 500 Hz for ring counter
            // clkNormal: 1 Hz
    wire  [3:0] tho; // Binary-Coded-Decimal 0-15
      wire  [3:0] hun;
      wire  [3:0] ten;
      wire  [3:0] one;
    wire  [6:0] thossd;
      wire  [6:0] hunssd;
      wire  [6:0] tenssd;
      wire  [6:0] onessd;
      // multi-purpose I-MEM read_addr_1
      wire [4:0] multi_purpose_read_addr;
      wire multi_purpose_RegWrite;

      // reg to resolve always block technicals
      reg clkRF_reg, clk_reg, multi_purpose_RegWrite_reg;
      reg [4:0] multi_purpose_read_addr_reg;
      reg [3:0] tho_reg, hun_reg, ten_reg, one_reg;

      assign D_MEM_addr = ALU_out[7:0];
      assign PC_in = PC_original[7:0];
      assign    PC_out_unsign_extended    =    {26'b0000_0000_0000_0000_0000_0000,
PC_out}; // from 7 bits to 32 bits
      assign jump_addr = {PC_plus4[31:28], jump_base28}; // jump_addr = (PC+4)
[31:28] joined with jump_base28[27:0]
      // output processor clock (1 Hz or freeze) to a LED
      assign LEDIndicator = clk;

      Program_Counter                                                    Unit1
(.clk(clk), .reset(reset), .PC_in(PC_in), .PC_out(PC_out));
      Instruction_Memory                                                 Unit2
(.read_addr(PC_out), .instruction(instruction), .reset(reset));
      Register_File                                                      Unit3
(.read_addr_1(multi_purpose_read_addr), .read_addr_2(instruction[20:16]), .write
_addr(reg_write_addr), .read_data_1(reg_read_data_1), .read_data_2(reg_read_data
_2), .write_data(reg_write_data), .RegWrite(multi_purpose_RegWrite), .clk(clkRF)
, .reset(reset));
      Data_Memory                                                        Unit4
(.addr(D_MEM_addr), .write_data(reg_read_data_2), .read_data(D_MEM_read_data), .
clk(clk), .reset(reset), .MemRead(MemRead), .MemWrite(MemWrite));
      Control                                                            Unit5
(.OpCode(instruction[31:26]), .RegDst(RegDst), .Jump(Jump), .Branch(Branch), .Me
mRead(MemRead), .MemtoReg(MemtoReg), .ALUOp(ALUOp), .MemWrite(MemWrite), .ALUSrc
(ALUSrc), .RegWrite(RegWrite));
      ALUControl                                                         Unit6
(.ALUOp(ALUOp), .funct(instruction[5:0]), .out_to_ALU(ALU_control_out));
      Sign_Extension                                                     Unit7
(.sign_in(instruction[15:0]), .sign_out(extended_immidiate));
```

```
      Shift_Left_2_Branch                                                    Unit8
(.shift_in(extended_immidiate), .shift_out(shifted_immidiate));
      Shift_Left_2_Jump                                                      Unit9
(.shift_in(instruction[25:0]), .shift_out(jump_base28));
      Mux_N_bit #(5) Unit10 (.in0(instruction[20:16]), .in1(instruction[15:11]),
.mux_out(reg_write_addr), .control(RegDst));
      Mux_N_bit                          #(32)                               Unit11
(.in0(reg_read_data_2), .in1(extended_immidiate), .mux_out(ALU_inB), .control(AL
USrc));
      Mux_N_bit                          #(32)                               Unit12
(.in0(ALU_out), .in1(D_MEM_read_data), .mux_out(reg_write_data), .control(MemtoR
eg));
      Mux_N_bit                          #(32)                               Unit13
(.in0(PC_plus4), .in1(Branch_out), .mux_out(Branch_result), .control(Branch_deci
ded));
      Mux_N_bit                          #(32)                               Unit14
(.in0(Branch_result), .in1(jump_addr), .mux_out(PC_original), .control(Jump));
      ALU                                                                    Unit15
(.inA(reg_read_data_1), .inB(ALU_inB), .alu_out(ALU_out), .zero(zero), .control(
ALU_control_out));
      ALU_add_only                                                           Unit16
(.inA(PC_out_unsign_extended), .inB(32'b0100), .add_out(PC_plus4)); // PC + 4
      ALU_add_only                                                           Unit17
(.inA(PC_plus4), .inB(shifted_immidiate), .add_out(Branch_out));
      and (Branch_decided, zero, Branch);

      // SSD Display
      divide_by_100k                                          Unit_Clock500HZ
(.clock(clkFast), .reset(reset), .clock_out(clkSSD));
      divide_by_500                                             Unit_Clock1HZ
(.clock(clkSSD), .reset(reset), .clock_out(clkNormal));
      Ring_4_counter Unit_Ring_Counter (.clock(clkSSD), .reset(reset), .Q(AN));
      ssd_driver  Unit_SSDTHO (.in_BCD(tho), .out_SSD(thossd));
      ssd_driver  Unit_SSDHUN (.in_BCD(hun), .out_SSD(hunssd));
      ssd_driver  Unit_SSDTEN (.in_BCD(ten), .out_SSD(tenssd));
      ssd_driver  Unit_SSDONE (.in_BCD(one), .out_SSD(onessd));
      choose_chathode                                           Unit_CHOOSE
(.tho(thossd), .hun(hunssd), .ten(tenssd), .one(onessd), .AN(AN), .CA(Cathode));

      assign clkRF = clkRF_reg;
      assign clk = clk_reg;
      assign multi_purpose_read_addr = multi_purpose_read_addr_reg;
      assign multi_purpose_RegWrite = multi_purpose_RegWrite_reg;
      assign tho = tho_reg;
      assign hun = hun_reg;
      assign ten = ten_reg;
      assign one = one_reg;

      always @(switchRun or clkSSD) begin
            if (switchRun) begin
                  // sys status 1: run single-cycle processor
                  clkRF_reg <= clkNormal; // 1 Hz
                  clk_reg <= clkNormal;         // 1 Hz
                  multi_purpose_read_addr_reg  <=  instruction[25:21];  //  reg-
file-port1 reads from instruction
                  // reg-file protection measure; explained in "else"
                  multi_purpose_RegWrite_reg <= RegWrite;
                  // output PC to SSD, but since PC only has 6 bits
                  tho_reg <= PC_out_unsign_extended[15:12]; // always 0
```

```
                  hun_reg <= PC_out_unsign_extended[11:8];  // always 0
                  ten_reg <= PC_out_unsign_extended[7:4];
                  one_reg <= PC_out_unsign_extended[3:0];
            end
            else begin
                  // sys status 2: pause processor; inspect Reg File content
                  clkRF_reg <= clkSSD;    // 500 Hz
                  clk_reg <= 1'b0;        // freeze at 0
                  multi_purpose_read_addr_reg <= SwitchSelector; // reg-file-
port1 reads from SwitchSelector
                  // Reg-file is not freezed in time, this protects against RF-
data-overwrite
                  multi_purpose_RegWrite_reg <= 1'b0;
                  // output reg file content to SSD, but only the lower 16 bits
(we only have 4 SSD)
                  tho_reg <= reg_read_data_1[15:12];
                  hun_reg <= reg_read_data_1[11:8];
                  ten_reg <= reg_read_data_1[7:4];
                  one_reg <= reg_read_data_1[3:0];
            end
      end
endmodule

// rising-edge synchronous program counter
// output range: decimal 0 to 63 (== I-MEM height)
// data I/O width: 64 = 2^6. Actually, 32 = 2^5; 5+2 offset = 7 bits
// async reset: set program counter to 0 asynchronously
module Program_Counter (clk, reset, PC_in, PC_out);
      input clk, reset;
      input [7:0] PC_in;
      output [7:0] PC_out;
      reg [7:0] PC_out;
      always @ (posedge clk or posedge reset)
      begin
            if(reset==1'b1)
                  PC_out<=0;
            else
                  PC_out<=PC_in;
      end
endmodule

// async read I-MEM
// height: 64, width: 32 bits (as required by TA)
// PC input width: 32 = 2^5 + 2 offset = 7 bits
// instruction output width: 32 bits (== I-MEM width)
// async reset: as specified in document "Project Two Specification (V3)",
//                     first reset all to 0, then hard-code instructions
module Instruction_Memory (read_addr, instruction, reset);
      input reset;
      input [7:0] read_addr;
      output [31:0] instruction;
      reg [31:0] Imemory [63:0];
      integer k;
      // I-MEM in this case is addressed by word, not by byte
      wire [5:0] shifted_read_addr;
      assign shifted_read_addr=read_addr[7:2];
      assign instruction = Imemory[shifted_read_addr];

      always @(posedge reset)
```

```
      begin

            for (k=30; k<64; k=k+1) begin// here Ou changes k=0 to k=16
                  Imemory[k] = 32'b0;
            end

            Imemory[0]   =   32'b00100000000010000000000000100000;   //addi   $t0,
$zero, 32
            Imemory[1]   =   32'b00100000000010010000000000110111;   //addi   $t1,
$zero, 55
            Imemory[2] = 32'b00000001000010011000000000100100; //and $s0, $t0,
$t1
            Imemory[3] = 32'b00000001000010011000000000100101; //or  $s0, $t0,
$t1
            Imemory[4]   =   32'b10101100000100000000000000000100;   //sw   $s0,
4($zero)
            Imemory[5]   =   32'b10101100000100000000000000001000;   //sw   $t0,
8($zero)
            Imemory[6] = 32'b00000001000010011000100000100000; //add $s1, $t0,
$t1
            Imemory[7] = 32'b00000001000010011001000000100010; //sub $s2, $t0,
$t1
            Imemory[8] = 32'b00010010001100100000000000001001; //beq $s1, $s2,
error0
            Imemory[9]   =   32'b10001100000100010000000000000100;   //lw   $s1,
4($zero)
            Imemory[10]= 32'b00110010001100100000000001001000; //andi $s2, $s1,
48
            Imemory[11] =32'b00010010001100100000000000001001; //beq $s1, $s2,
error1
            Imemory[12]    =32'b10001100000100110000000000001000;    //lw    $s3,
8($zero)
            Imemory[13] =32'b00010010000100110000000000001010; //beq $s0, $s3,
error2
            Imemory[14] =32'b00000010010100011010000000101010; //slt $s4, $s2,
$s1 (Last)
            Imemory[15] =32'b00010010100000000000000000001111; //beq $s4, $0,
EXIT
            Imemory[16] =32'b00000010001000001001000000100000; //add $s2, $s1,
$0
            Imemory[17] =32'b00001000000000000000000000001110; //j Last
            Imemory[18] =32'b00100000000010000000000000000000; //addi $t0, $0,
0(error0)
            Imemory[19] =32'b00100000000010010000000000000000; //addi $t1, $0, 0
            Imemory[20] =32'b00001000000000000000000000011111; //j EXIT
            Imemory[21] =32'b00100000000010000000000000000001; //addi $t0, $0,
1(error1)
            Imemory[22] =32'b00100000000010010000000000000001; //addi $t1, $0, 1
            Imemory[23] =32'b00001000000000000000000000011111; //j EXIT
            Imemory[24] =32'b00100000000010000000000000000010; //addi $t0, $0,
2(error2)
            Imemory[25] =32'b00100000000010010000000000000010; //addi $t1, $0, 2
            Imemory[26] =32'b00001000000000000000000000011111; //j EXIT
            Imemory[27] =32'b00100000000010000000000000000011; //addi $t0, $0,
3(error3)
            Imemory[28] =32'b00100000000010010000000000000011; //addi $t1, $0, 3
            Imemory[29] =32'b00001000000000000000000000011111; //j EXIT

      end
```

```
endmodule

// sync register file (write/read occupy half cycle each)
// height: 32 (from $0 to $ra), width: 32 bits
// write: on rising edge; data width 32 bit; address width 5 bit
// read: on falling edge; data width 32 bit; address width 5 bit
// control: write on rising edge if (RegWrite == 1)
// async reset: set all register content to 0
module  Register_File  (read_addr_1,  read_addr_2,  write_addr,  read_data_1,
read_data_2, write_data, RegWrite, clk, reset);
      input [4:0] read_addr_1, read_addr_2, write_addr;
      input [31:0] write_data;
      input clk, reset, RegWrite;
      output [31:0] read_data_1, read_data_2;

      reg [31:0] Regfile [31:0];
      integer k;

      assign read_data_1 = Regfile[read_addr_1];
      assign read_data_2 = Regfile[read_addr_2];

      always @(posedge clk or posedge reset) // Ou combines the block of reset
into the block of posedge clk
      begin
            if (reset==1'b1)
            begin
                  for (k=0; k<32; k=k+1)
                  begin
                        Regfile[k] = 32'b0;
                  end
            end

            else if (RegWrite == 1'b1) Regfile[write_addr] = write_data;
      end



endmodule

// rising edge sync-write, async-read D-MEM
// height: 64, width: 32 bits (from document "Project Two Specification (V3)")
// address input: 6 bits (64 == 2^6)
// data input/output: 32 bits
// write: on rising edge, when (MemWrite == 1)
// read: asynchronous, when (MemRead == 1)
module Data_Memory (addr, write_data, read_data, clk, reset, MemRead, MemWrite);
      input [7:0] addr;
      input [31:0] write_data;
      output [31:0] read_data;
      input clk, reset, MemRead, MemWrite;
      reg [31:0] DMemory [63:0];
      integer k;
      wire [5:0] shifted_addr;
      assign shifted_addr=addr[7:2];
      assign read_data = (MemRead) ? DMemory[addr] : 32'bx;

      always @(posedge clk or posedge reset)// Ou modifies reset to posedge
      begin
            if (reset == 1'b1)
```

```
                begin
                      for (k=0; k<64; k=k+1) begin
                            DMemory[k] = 32'b0;
                      end
                end
          else
                if (MemWrite) DMemory[addr] = write_data;
      end
endmodule

// async control signal generation unit based on OpCode
// as specified in Fig 4.22
// attached in email as file "Fig 4_22 Single Cycle Control"
// input: 6 bits OpCode
// output: all 1 bit except ALUOp which is 2-bits wide
module Control (OpCode, RegDst, Jump, Branch, MemRead, MemtoReg, ALUOp,
MemWrite, ALUSrc, RegWrite);
      input [5:0] OpCode;
      output RegDst, Jump, Branch, MemRead, MemtoReg, MemWrite, ALUSrc,
RegWrite;
      output [1:0] ALUOp;

      assign
RegDst=(~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(~OpCode[1])&(~OpCode
[0]);//000000
      assign
Jump=(~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(OpCode[1])&(~OpCode[0]
);//000010
      assign
Branch=(~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(OpCode[2])&(~OpCode[1])&(~OpCode[
0]);//000100
      assign
MemRead=(OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(OpCode[1])&(OpCode[0
]);//100011
      assign
MemtoReg=(OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(OpCode[1])&(OpCode[
0]);//100011
      assign
MemWrite=(OpCode[5])&(~OpCode[4])&(OpCode[3])&(~OpCode[2])&(OpCode[1])&(OpCode[0
]);//101011
      assign
ALUSrc=((~OpCode[5])&(~OpCode[4])&(OpCode[3])&(~OpCode[2])&(~OpCode[1])&(~OpCode
[0]))                                                                    |
((~OpCode[5])&(~OpCode[4])&(OpCode[3])&(OpCode[2])&(~OpCode[1])&(~OpCode[0]))   |
((OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(OpCode[1])&(OpCode[0]))    |
(((OpCode[5])&(~OpCode[4])&(OpCode[3])&(~OpCode[2])&(OpCode[1])&(OpCode[0])));  /
/001000,001100,100011,101011
      assign
RegWrite=(~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(~OpCode[1])&(~OpCo
de[0])                                                                   |
((~OpCode[5])&(~OpCode[4])&(OpCode[3])&(~OpCode[2])&(~OpCode[1])&(~OpCode[0]))  |
((~OpCode[5])&(~OpCode[4])&(OpCode[3])&(OpCode[2])&(~OpCode[1])&(~OpCode[0]))   |
((OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(OpCode[1])&(OpCode[0]));//
000000,001000,001100,100011
      assign
ALUOp[1]=((~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(~OpCode[1])&(~OpC
ode[0]))|
((~OpCode[5])&(~OpCode[4])&(OpCode[3])&(OpCode[2])&(~OpCode[1])&(~OpCode[0]));//
000000, 001100(andi)
```

```
      assign                                                    ALUOp[0]=
((~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(OpCode[2])&(~OpCode[1])&(~OpCode[0]))|
((~OpCode[5])&(~OpCode[4])&(OpCode[3])&(OpCode[2])&(~OpCode[1])&(~OpCode[0]));//
000100,001100(andi)
endmodule

// async control to generate ALU input signal
// as specified in Fig 4.12
// attached in email as file "Fig 4_12 ALU Control Input"
// input: 2-bit ALUOp control signal and 6-bit funct field from instruction
// output: 4-bit ALU control input
module ALUControl (ALUOp, funct, out_to_ALU);
      input [1:0] ALUOp;
      input [5:0] funct;
      output [3:0] out_to_ALU;

      assign out_to_ALU[3]=0;
      assign               out_to_ALU[2]=((~ALUOp[1])&(ALUOp[0]))               |
((ALUOp[1])&(~ALUOp[0])&(~funct[3])&(~funct[2])&(funct[1])&(~funct[0]))         |
((ALUOp[1])&(~ALUOp[0])&(funct[3])&(~funct[2])&(funct[1])&(~funct[0]));
      assign  out_to_ALU[1]=((~ALUOp[1])&(~ALUOp[0]))|((~ALUOp[1])&(ALUOp[0]))  |
((ALUOp[1])&(~ALUOp[0])&(~funct[3])&(~funct[2])&(~funct[1])&(~funct[0]))         |
((ALUOp[1])&(~ALUOp[0])&(~funct[3])&(~funct[2])&(funct[1])&(~funct[0]))|
((ALUOp[1])&(~ALUOp[0])&(funct[3])&(~funct[2])&(funct[1])&(~funct[0]));
      assign
out_to_ALU[0]=((ALUOp[1])&(~ALUOp[0])&(~funct[3])&(funct[2])&(~funct[1])&(funct[
0]))|((ALUOp[1])&(~ALUOp[0])&(funct[3])&(~funct[2])&(funct[1])&(~funct[0]));
endmodule

// sign-extend the 16-bit input to the 32_bit output
module Sign_Extension (sign_in, sign_out);
      input [15:0] sign_in;
      output [31:0] sign_out;
      assign sign_out[15:0]=sign_in[15:0];
      assign sign_out[31:16]=sign_in[15]?16'b1111_1111_1111_1111:16'b0;
endmodule

// shift-left-2 for branch instruction
// input width: 32 bits
// output width: 32 bits
// fill the void with 0 after shifting
module Shift_Left_2_Branch (shift_in, shift_out);
      input [31:0] shift_in;
      output [31:0] shift_out;
      assign shift_out[31:0]={shift_in[29:0],2'b00};
endmodule

// shift-left-2 for jump instruction
// input width: 26 bits
// output width: 28 bits
// fill the void with 0 after shifting
// we don't need to shift in this case, becasue the address of the instructions
// are addressed by words
module Shift_Left_2_Jump (shift_in, shift_out);
      input [25:0] shift_in;
      output [27:0] shift_out;
      assign shift_out[27:0]={shift_in[25:0],2'b00};
endmodule
```

```
// N-bit 2-to-1 Mux
// input: 2 N-bit input
// output: 1 N-bit output
// control: 1 bit
// possible value of N in single cycle: 5, 6, 32
module Mux_N_bit (in0, in1, mux_out, control);
      parameter N = 32;
      input [N-1:0] in0, in1;
      output [N-1:0] mux_out;
      input control;
      assign mux_out=control?in1:in0;
endmodule

// 32-bit ALU
// data input width: 2 32-bit
// data output width: 1 32-bit and one "zero" output
// control: 4-bit
// zero: output 1 if all bits of data output is 0
// as specified in Fig 4.12
// attached in email as "Fig 4_12 ALU Control Input"
module ALU (inA, inB, alu_out, zero, control);
      input [31:0] inA, inB;
      output [31:0] alu_out;
      output zero;
      reg zero;
      reg [31:0] alu_out;
      input [3:0] control;
      always @ (control or inA or inB)
      begin
            case (control)
            4'b0000:begin zero<=0; alu_out<=inA&inB; end
            4'b0001:begin zero<=0; alu_out<=inA|inB; end
            4'b0010:begin zero<=0; alu_out<=inA+inB; end
            4'b0110:begin if(inA==inB) zero<=1; else zero<=0; alu_out<=inA-inB;
end
            4'b0111:begin  zero<=0;  if(inA-inB>=32'h8000_0000)  alu_out<=32'b1;
else alu_out<=32'b0; end// how to implement signed number
            default: begin zero<=0; alu_out<=inA; end
            endcase
      end
endmodule

// 32-bit ALU for addition only
// data input width: 2 32-bit
// data output width: 1 32-bit, no "zero" output
// control: no control input, only addition operation implemeneted
// as specified in Fig 4.12
// attached in email as "Fig 4_12 ALU Control Input"
module ALU_add_only (inA, inB, add_out);
      input [31:0] inA, inB;
      output [31:0] add_out;
      assign add_out=inA+inB;
endmodule

module Dff_asy (q, d, clk, rst);
      input d, clk, rst;
      output reg q;

      always @ (posedge clk or posedge rst)
```

```
            if (rst == 1) q <= 0;
            else q <= d;
endmodule

// The following modules implement SSD display & clock slow-down
module divide_by_500 (clock, reset, clock_out); // modified to divide-by-4 for
simulation
      parameter N = 9;
      input       clock, reset;
      wire        load, asyclock_out;
      wire        [N-1:0] Dat;
      output      clock_out;
      reg         [N-1:0] Q;
      assign      Dat = 9'b000000000;
      assign      load  =  Q[1]  &  Q[0];  //  modified  to  load  =  3  (DEC)  for
simulation
      always @ (posedge reset or posedge clock)
      begin
            if (reset == 1'b1) Q <= 9'b000000000;
            else if (load == 1'b1) Q <= Dat;
            else Q <= Q + 1;
      end
      assign      asyclock_out = load;
      Dff_asy                                                          Unit_Dff
(.q(clock_out), .d(asyclock_out), .clk(clock), .rst(reset));
endmodule

module divide_by_100k (clock, reset, clock_out); // modified to divide-by-4 for
simulation
      parameter N = 17;
      input clock, reset;
      wire  load, asyclock_out;
      wire  [N-1:0] Dat;
      output      clock_out;
      reg   [N-1:0] Q;
      assign      Dat = 0;
      assign      load  =  Q[1]  &  Q[0];  //  modified  to  load  =  3  (DEC)  for
simulation
      always @ (posedge reset or posedge clock)
      begin
            if (reset == 1'b1) Q <= 0;
            else if (load == 1'b1) Q <= Dat;
            else Q <= Q + 1;
      end
      assign      asyclock_out = load;
      Dff_asy                                                          Unit_Dff
(.q(clock_out), .d(asyclock_out), .clk(clock), .rst(reset));
endmodule

module Ring_4_counter(clock, reset, Q);
      input             clock, reset;
      output reg  [3:0]Q;

      always @(posedge clock or posedge reset)
      begin
            if (reset == 1) Q <= 4'b1110;
            else
            begin
                  Q[3] <= Q[0];
```

```
                Q[2] <= Q[3];
                Q[1] <= Q[2];
                Q[0] <= Q[1];
            end
      end
endmodule

module ssd_driver (in_BCD, out_SSD);
      input [3:0] in_BCD; // input in Binary-Coded Decimal
      output [6:0] out_SSD; // output to Seven-Segment Display
      reg [6:0] out_SSD;
      always @(in_BCD) begin
            case (in_BCD)
            0:out_SSD=7'b0000001;
            1:out_SSD=7'b1001111;
            2:out_SSD=7'b0010010;
            3:out_SSD=7'b0000110;
            4:out_SSD=7'b1001100;
            5:out_SSD=7'b0100100;
            6:out_SSD=7'b0100000;
            7:out_SSD=7'b0001111;
            8:out_SSD=7'b0000000;
            9:out_SSD=7'b0000100;
            10:out_SSD=7'b0001000;
            11:out_SSD=7'b1100000;
            12:out_SSD=7'b0110001;
            13:out_SSD=7'b1000010;
            14:out_SSD=7'b0110000;
            15:out_SSD=7'b0111000;
                  default out_SSD = 7'b1111111; // no ssd
            endcase
      end
endmodule

module choose_chathode(tho, hun, ten, one, AN, CA);
      input [6:0]tho;
      input [6:0]hun;
      input [6:0]ten;
      input [6:0]one;
      input [3:0]AN;
      output      [6:0]CA;
      assign CA = (AN==4'b1110) ? one : 7'bzzzzzzz,
               CA =     (AN==4'b1101) ? ten : 7'bzzzzzzz,
               CA = (AN==4'b1011) ? hun : 7'bzzzzzzz,
               CA =     (AN==4'b0111) ? tho : 7'bzzzzzzz;
endmodule

`timescale 1ns / 1ps

module test_bench;

      reg clkFast;
      reg reset;
      reg [4:0] SwitchSelector;
      reg switchRun;
      reg clkread;

      integer count;
```

```
// Outputs
wire [31:0]reg_read_data_1;

// Instantiate the Unit Under Test (UUT)
single_cycle uut (
	.clkFast(clkFast),
	.reset(reset),
	.SwitchSelector(SwitchSelector),
	.switchRun(switchRun),
	.reg_read_data_1(reg_read_data_1));

initial begin
	// Initialize Inputs
	count = 0;
	clkFast = 0;
	reset = 1;
	switchRun = 0;
	SwitchSelector = 5'd0;
	clkread = 0;
	#4 reset=0;
	#3000;
	#50 $stop;
end

always begin #1 clkFast=~clkFast; end
always begin #200 clkread = ~clkread; end

always @(posedge clkread)
begin
	$display ("Time: %d", count);
		SwitchSelector = 5'd16;
	#10 $display ("[$s0] = %h", reg_read_data_1);
		SwitchSelector = 5'd17;
	#10 $display ("[$s1] = %h", reg_read_data_1);
		SwitchSelector = 5'd18;
	#10 $display ("[$s2] = %h", reg_read_data_1);
		SwitchSelector = 5'd19;
	#10 $display ("[$s3] = %h", reg_read_data_1);
		SwitchSelector = 5'd20;
	#10 $display ("[$s4] = %h", reg_read_data_1);
		SwitchSelector = 5'd21;
	#10 $display ("[$s5] = %h", reg_read_data_1);
		SwitchSelector = 5'd22;
	#10 $display ("[$s6] = %h", reg_read_data_1);
		SwitchSelector = 5'd23;
	#10 $display ("[$s7] = %h", reg_read_data_1);
		SwitchSelector = 5'd8;
	#10 $display ("[$t0] = %h", reg_read_data_1);
		SwitchSelector = 5'd9;
	#10 $display ("[$t1] = %h", reg_read_data_1);
		SwitchSelector = 5'd10;
	#10 $display ("[$t2] = %h", reg_read_data_1);
		SwitchSelector = 5'd11;
	#10 $display ("[$t3] = %h", reg_read_data_1);
		SwitchSelector = 5'd12;
	#10 $display ("[$t4] = %h", reg_read_data_1);
		SwitchSelector = 5'd13;
	#10 $display ("[$t5] = %h", reg_read_data_1);
```

```
                SwitchSelector = 5'd14;
            #10 $display ("[$t6] = %h", reg_read_data_1);
                SwitchSelector = 5'd15;
            #10 $display ("[$t7] = %h", reg_read_data_1);
                SwitchSelector = 5'd24;
            #10 $display ("[$t8] = %h", reg_read_data_1);
                SwitchSelector = 5'd25;
            #10 $display ("[$t9] = %h", reg_read_data_1);

            #2 switchRun = 1;
            #32 switchRun = 0;
            count = count + 1;

    end


endmodule
```

V.II Pipeline code (loaded with demo instructions; for simulation)

```
// p2_pipeline.v
// ver 8.0
// VE370 Project #2

// Due:
// Report & Source code due Nov 27 23:00 through Sakai
// Demonstration is due by 4:00pm, November 27, 2015

// Created by Hua Xing, Junqi Qian, Xinyue Ou on 11/12/15
// Copyright (c) 2015. All rights reserved.

// Version History
// ver 1.0: top level schematics ready

// ver 2.0: added clock-divider and SSD output. SSD outputs PC or reg file
content.

// ver 3.0: combine the v2 and the proof reading result.
// major changes:
// 1. change the bus width of some wires.
// 2. add a new wire ID_EX_funct, and hence change the I/O of the stage register
of ID_EX.
// 3. change the connection of the 3to1 mux to make the structure
// of the connection parallel (and hence easier to implement forwarding unit).
// 4. implement the modules other than the stage registers.

// ver 4.0. FPGA implementation complete.
// ver 5.0. Initial simulation code ready. But buggy.
// ver 6.0. Modified clock dividers for simulation. Simulation complete.
// ver 7.0: expanded PC width to 7 bits (5 bit + 2 bit offset). I-MEM now
supports 2^5 = 32 lines.
// ver 8.0: added TA demo code
// ver 10.1 update control to accomodate andi; a version for for simulation
module pipeline (clkFast, reset, SwitchSelector, switchRun, Cathode, AN,
LEDIndicator, reg_read_data_1);
      input clkFast, reset;          // clkFast (5m Hz) feeds clock divider
      input switchRun;               // decide run(1) or check reg file(0)
      input [4:0] SwitchSelector;    // select $0 to $31 for output
      output [6:0] Cathode;          // SSD
      output [3:0] AN;
      output LEDIndicator;           // output clkNormal for user reference
      output [31:0] reg_read_data_1;

      // wires in IF stage
      wire [31:0] PC_in_original;
      wire [6:0] PC_out_short;
      wire [31:0] PC_out_unsign_extended, PC_plus4;
      wire [31:0] instruction;
      wire [31:0] branch_jump_addr;
      // wires in ID stage
      wire [31:0] IF_ID_PC_plus4, IF_ID_instruction;
      wire [4:0] MEM_WB_RegisterRd;
      wire [31:0] reg_read_data_2;

      wire [31:0] immi_sign_extended;
            // jump within ID stage
      wire [31:0] jump_addr;
```

```
      wire [27:0] jump_base28;
            // control signal generation within ID stage
      wire RegDst, Jump, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite;
      wire [1:0] ALUOp;
      // wires in EX stage
      wire     ID_EX_RegDst,     ID_EX_Jump,     ID_EX_Branch,     ID_EX_MemRead,
ID_EX_MemtoReg, ID_EX_MemWrite, ID_EX_ALUSrc, ID_EX_RegWrite;
      wire [1:0] ID_EX_ALUOp;
      wire [31:0] ID_EX_jump_addr;
      wire [31:0] ID_EX_PC_plus4, ID_EX_reg_read_data_1, ID_EX_reg_read_data_2;
      wire [31:0] ID_EX_immi_sign_extended;
      wire  [4:0]  ID_EX_RegisterRs,  ID_EX_RegisterRt,  ID_EX_RegisterRd;//  Ou
modifies: [31:0]
      wire [4:0] EX_RegisterRd;
      wire [5:0] ID_EX_funct;
      wire [3:0] out_to_ALU;
      wire [31:0] muxA_out, muxB_out;
      wire [31:0] after_ALUSrc;
      wire [31:0] ALU_result;
      wire ALU_zero;
      wire [31:0] after_shift, branch_addr;
      // wires in MEM stage
      wire [4:0] EX_MEM_RegisterRd;
      wire  EX_MEM_RegWrite,  EX_MEM_MemtoReg,  EX_MEM_Branch,  EX_MEM_MemRead,
EX_MEM_MemWrite, EX_MEM_Jump;
      wire [31:0] EX_MEM_jump_addr, EX_MEM_branch_addr;
      wire EX_MEM_ALU_zero;
      wire [31:0] EX_MEM_ALU_result, EX_MEM_reg_read_data_2;
      wire [31:0] D_MEM_data;
      wire Branch_taken;
      // wires in WB stage
      wire [31:0] reg_write_data;
      wire MEM_WB_RegWrite, MEM_WB_MemtoReg;
      wire [31:0] MEM_WB_D_MEM_read_data, MEM_WB_D_MEM_read_addr;
      // wires for forwarding
      wire [1:0] ForwardA, ForwardB;
      // wires for lw hazard stall
      wire PCWrite;           // PC stops writing if PCWrite == 0
      wire IF_ID_Write;       // IF/ID reg stops writing if IF_ID_Write == 0
      wire ID_Flush_lwstall;
      // wires for jump/branch control hazard
      wire PCSrc;
      wire IF_Flush, ID_Flush_Branch, EX_Flush;

      // SSD display & clock slow-down
      wire clkSSD, clkNormal, clkRF, clk;
            // clkFast: 5m Hz
            // clkSSD: 500 Hz for ring counter
            // clkNormal: 1 Hz
    wire  [3:0] tho; // Binary-Coded-Decimal 0-15
      wire  [3:0] hun;
      wire  [3:0] ten;
      wire  [3:0] one;
    wire  [6:0] thossd;
      wire  [6:0] hunssd;
      wire  [6:0] tenssd;
      wire  [6:0] onessd;
      // multi-purpose I-MEM read_addr_1
      wire [4:0] multi_purpose_read_addr;
```

```
      wire multi_purpose_RegWrite;

      // reg to resolve always block technicalities
      reg clkRF_reg, clk_reg, multi_purpose_RegWrite_reg;
      reg [4:0] multi_purpose_read_addr_reg;
      reg [3:0] tho_reg, hun_reg, ten_reg, one_reg;

   //new forward, change id_ex stage register
      wire ForwardC,ForwardD;
      wire [31:0] muxC_out,muxD_out;
      Mux_N_bit                              #(32)                              Unit26
(.in0(reg_read_data_1),.in1(reg_write_data),.mux_out(muxC_out),.control(ForwardC
));
      Mux_N_bit                              #(32)                              Unit27
(.in0(reg_read_data_2),.in1(reg_write_data),.mux_out(muxD_out),.control(ForwardD
));


      // output processor clock (1 Hz or freeze) to a LED
      assign LEDIndicator = clk;
      // IF stage
      Program_Counter Unit0 (
            .clk(clk), .reset(reset),
            .PC_in(PC_in_original[6:0]),
            .PC_out(PC_out_short),
            .PCWrite(PCWrite));
      Instruction_Memory Unit1 (.read_addr(PC_out_short),
            .instruction(instruction), .reset(reset));
      assign   PC_out_unsign_extended   =   {26'b0000_0000_0000_0000_0000_0000_0,
PC_out_short}; // from 7 bits to 32 bits
      ALU_add_only Unit2 (.inA(PC_out_unsign_extended),
                          .inB(32'b0100), .add_out(PC_plus4)); // PC + 4
      Mux_N_bit #(32) Unit3 (.in0(PC_plus4), .in1(branch_jump_addr),
                          .mux_out(PC_in_original), .control(PCSrc));
      IF_ID_Stage_Reg Unit4 (.PC_plus4_in(PC_plus4),
                  .PC_plus4_out(IF_ID_PC_plus4),
                  .instruction_in(instruction),
                  .instruction_out(IF_ID_instruction),
                  .IF_ID_Write(IF_ID_Write), .IF_Flush(IF_Flush),
                  .clk(clk), .reset(reset));
      // ID stage
      Register_File Unit5 (.read_addr_1(multi_purpose_read_addr),
                          .read_addr_2(IF_ID_instruction[20:16]),
                          .write_addr(MEM_WB_RegisterRd),
                          .read_data_1(reg_read_data_1),
                          .read_data_2(reg_read_data_2),
                          .write_data(reg_write_data),
                          .RegWrite(multi_purpose_RegWrite), .clk(clkRF),
                          .reset(reset));
      Sign_Extension Unit6 (.sign_in(IF_ID_instruction[15:0]),
                          .sign_out(immi_sign_extended));
      // jump within ID stage
      Shift_Left_2_Jump Unit7 (.shift_in(IF_ID_instruction[25:0]),
                          .shift_out(jump_base28));
      assign jump_addr = {IF_ID_PC_plus4[31:28], jump_base28}; // jump_addr =
(PC+4)[31:28] joined with jump_base28[27:0]
      Control                                                                  Unit8
(.OpCode(IF_ID_instruction[31:26]), .RegDst(RegDst), .Jump(Jump), .Branch(Branch
), .MemRead(MemRead), .MemtoReg(MemtoReg), .ALUOp(ALUOp), .MemWrite(MemWrite), .
```

```
ALUSrc(ALUSrc), .RegWrite(RegWrite));
      ID_EX_Stage_Reg                                                    Unit9
(.ID_Flush_lwstall(ID_Flush_lwstall), .ID_Flush_Branch(ID_Flush_Branch), .RegWri
te_in(RegWrite), .MemtoReg_in(MemtoReg), .RegWrite_out(ID_EX_RegWrite), .MemtoRe
g_out(ID_EX_MemtoReg), .Branch_in(Branch), .MemRead_in(MemRead), .MemWrite_in(Me
mWrite), .Jump_in(Jump), .Branch_out(ID_EX_Branch), .MemRead_out(ID_EX_MemRead),
      .MemWrite_out(ID_EX_MemWrite), .Jump_out(ID_EX_Jump), .RegDst_in(RegDst),
.ALUSrc_in(ALUSrc), .RegDst_out(ID_EX_RegDst), .ALUSrc_out(ID_EX_ALUSrc), .ALUOp
_in(ALUOp), .ALUOp_out(ID_EX_ALUOp), .jump_addr_in(jump_addr), .PC_plus4_in(IF_I
D_PC_plus4), .jump_addr_out(ID_EX_jump_addr), .PC_plus4_out(ID_EX_PC_plus4),
      .reg_read_data_1_in(muxC_out), .reg_read_data_2_in(muxD_out), .immi_sign_e
xtended_in(immi_sign_extended), .reg_read_data_1_out(ID_EX_reg_read_data_1), .re
g_read_data_2_out(ID_EX_reg_read_data_2), .immi_sign_extended_out(ID_EX_immi_sig
n_extended), .IF_ID_RegisterRs_in(IF_ID_instruction[25:21]),
      .IF_ID_RegisterRt_in(IF_ID_instruction[20:16]), .IF_ID_RegisterRd_in(IF_ID
_instruction[15:11]), .IF_ID_RegisterRs_out(ID_EX_RegisterRs), .IF_ID_RegisterRt
_out(ID_EX_RegisterRt), .IF_ID_RegisterRd_out(ID_EX_RegisterRd), .IF_ID_funct_in
(IF_ID_instruction[5:0]),.IF_ID_funct_out(ID_EX_funct),.clk(clk), .reset(reset))
;//Ou adds  IF_ID_funct_in and out
      // EX stage
      Mux_N_bit                                #(5)                      Unit10
(.in0(ID_EX_RegisterRt), .in1(ID_EX_RegisterRd), .mux_out(EX_RegisterRd), .contr
ol(ID_EX_RegDst));
      ALUControl                                                         Unit11
(.ALUOp(ID_EX_ALUOp),    .funct(ID_EX_funct),    .out_to_ALU(out_to_ALU));//    Ou
modifies:  funct  should  not  be  IF_ID_instruction.  Rather,  it  should  be
ID_EX_funct(a new wire)
      Mux_32bit_3to1                                                 Unit12_muxA
(.in00(ID_EX_reg_read_data_1), .in01(reg_write_data), .in10(EX_MEM_ALU_result),
.mux_out(muxA_out), .control(ForwardA));
      Mux_32bit_3to1                                                 Unit13_muxB
(.in00(ID_EX_reg_read_data_2), .in01(reg_write_data), .in10(EX_MEM_ALU_result),
.mux_out(muxB_out),  .control(ForwardB));//Ou  modifies:  keep  the  structure
paralleled with muxA
      Mux_N_bit                                #(32)                     Unit14
(.in0(muxB_out), .in1(ID_EX_immi_sign_extended), .mux_out(after_ALUSrc), .contro
l(ID_EX_ALUSrc));
      ALU                                                                Unit15
(.inA(muxA_out), .inB(after_ALUSrc), .alu_out(ALU_result), .zero(ALU_zero), .con
trol(out_to_ALU));
      Shift_Left_2_Branch                                                Unit16
(.shift_in(ID_EX_immi_sign_extended), .shift_out(after_shift));
      ALU_add_only                                                       Unit17
(.inA(ID_EX_PC_plus4),  .inB(after_shift),  .add_out(branch_addr));  //  (PC+4)  +
branch_addition*4Z
      //  in  EX/MEM  stage  reg,  note  muxB_out  is  used  in  the  place  of
reg_read_data_2 as a result of forwarding;ygb
      EX_MEM_Stage_Reg                                                   Unit18
(.EX_Flush(EX_Flush), .RegWrite_in(ID_EX_RegWrite), .MemtoReg_in(ID_EX_MemtoReg)
, .RegWrite_out(EX_MEM_RegWrite), .MemtoReg_out(EX_MEM_MemtoReg),
      .Branch_in(ID_EX_Branch), .MemRead_in(ID_EX_MemRead), .MemWrite_in(ID_EX_M
emWrite),
      .Jump_in(ID_EX_Jump), .Branch_out(EX_MEM_Branch), .MemRead_out(EX_MEM_MemR
ead),
      .MemWrite_out(EX_MEM_MemWrite), .Jump_out(EX_MEM_Jump), .jump_addr_in(ID_E
X_jump_addr),
      .branch_addr_in(branch_addr), .jump_addr_out(EX_MEM_jump_addr),
      .branch_addr_out(EX_MEM_branch_addr), .ALU_zero_in(ALU_zero),
      .ALU_zero_out(EX_MEM_ALU_zero), .ALU_result_in(ALU_result), .reg_read_data
```

```
_2_in(muxB_out),
      .ALU_result_out(EX_MEM_ALU_result), .reg_read_data_2_out(EX_MEM_reg_read_d
ata_2),
      .ID_EX_RegisterRd_in(EX_RegisterRd), .EX_MEM_RegisterRd_out(EX_MEM_Registe
rRd), .clk(clk), .reset(reset));
      // MEM stage
      Data_Memory                                                    Unit19
(.addr(EX_MEM_ALU_result[7:0]), .write_data(EX_MEM_reg_read_data_2), .read_data(
D_MEM_data), .clk(clk), .reset(reset), .MemRead(EX_MEM_MemRead), .MemWrite(EX_ME
M_MemWrite));
      and (Branch_taken, EX_MEM_Branch, EX_MEM_ALU_zero);
      jump_OR_branch                                                 Unit20
(.Jump(EX_MEM_Jump), .Branch_taken(Branch_taken), .branch_addr(EX_MEM_branch_add
r), .jump_addr(EX_MEM_jump_addr), .PCSrc(PCSrc), .addr_out(branch_jump_addr));
      //WB stage
      MEM_WB_Stage_Reg                                               Unit21
(.RegWrite_in(EX_MEM_RegWrite), .MemtoReg_in(EX_MEM_MemtoReg), .RegWrite_out(MEM
_WB_RegWrite), .MemtoReg_out(MEM_WB_MemtoReg), .D_MEM_read_data_in(D_MEM_data),
.D_MEM_read_addr_in(EX_MEM_ALU_result), .D_MEM_read_data_out(MEM_WB_D_MEM_read_d
ata),
      .D_MEM_read_addr_out(MEM_WB_D_MEM_read_addr), .EX_MEM_RegisterRd_in(EX_MEM
_RegisterRd), .MEM_WB_RegisterRd_out(MEM_WB_RegisterRd), .clk(clk), .reset(reset
));
      Mux_N_bit                          #(32)                       Unit22
(.in0(MEM_WB_D_MEM_read_addr), .in1(MEM_WB_D_MEM_read_data), .mux_out(reg_write_
data), .control(MEM_WB_MemtoReg));
      Forwarding_Control                                             Utni23
(.EX_MEM_RegisterRd(EX_MEM_RegisterRd), .MEM_WB_RegisterRd(MEM_WB_RegisterRd), .
ID_EX_RegisterRs(ID_EX_RegisterRs), .ID_EX_RegisterRt(ID_EX_RegisterRt), .EX_MEM
_RegWrite(EX_MEM_RegWrite), .MEM_WB_RegWrite(MEM_WB_RegWrite), .IF_ID_RegisterRs
(IF_ID_instruction[25:21]),.IF_ID_RegisterRt(IF_ID_instruction[20:16]),.ForwardA
(ForwardA), .ForwardB(ForwardB),.ForwardC(ForwardC), .ForwardD(ForwardD));
      stall_for_lw_Control                                           Unit24
(.ID_EX_RegisterRt(ID_EX_RegisterRt), .IF_ID_RegisterRs(IF_ID_instruction[25:21]
), .IF_ID_RegisterRt(IF_ID_instruction[20:16]), .ID_EX_MemRead(ID_EX_MemRead), .
PCWrite(PCWrite), .IF_ID_Write(IF_ID_Write), .ID_Flush_lwstall(ID_Flush_lwstall)
);
      branch_and_jump_hazard_control                                 Unit25
(.MEM_PCSrc(PCSrc), .IF_Flush(IF_Flush), .ID_Flush_Branch(ID_Flush_Branch), .EX_
Flush(EX_Flush));
      // SSD Display
      divide_by_100k                                       Unit_Clock500HZ
(.clock(clkFast), .reset(reset), .clock_out(clkSSD));
      divide_by_500                                          Unit_Clock1HZ
(.clock(clkSSD), .reset(reset), .clock_out(clkNormal));
      Ring_4_counter Unit_Ring_Counter (.clock(clkSSD), .reset(reset), .Q(AN));
      ssd_driver  Unit_SSDTHO (.in_BCD(tho), .out_SSD(thossd));
      ssd_driver  Unit_SSDHUN (.in_BCD(hun), .out_SSD(hunssd));
      ssd_driver  Unit_SSDTEN (.in_BCD(ten), .out_SSD(tenssd));
      ssd_driver  Unit_SSDONE (.in_BCD(one), .out_SSD(onessd));
      choose_chathode                                          Unit_CHOOSE
(.tho(thossd), .hun(hunssd), .ten(tenssd), .one(onessd), .AN(AN), .CA(Cathode));

      assign clkRF = clkRF_reg;
      assign clk = clk_reg;
      assign multi_purpose_read_addr = multi_purpose_read_addr_reg;
      assign multi_purpose_RegWrite = multi_purpose_RegWrite_reg;
      assign tho = tho_reg;
      assign hun = hun_reg;
```

```
	assign ten = ten_reg;
	assign one = one_reg;

	always @(switchRun or clkSSD) begin
		if (switchRun) begin
			// sys status 1: run pipeline processor
			clkRF_reg <= clkNormal;		// 1 Hz
			clk_reg <= clkNormal;		// 1 Hz
			multi_purpose_read_addr_reg  <=  IF_ID_instruction[25:21];  //
reg-file-port1 reads from instruction
			// reg-file protection measure; explained in "else"
			multi_purpose_RegWrite_reg <= MEM_WB_RegWrite;
			// output PC to SSD, but since PC only has 6 bits
			tho_reg <= PC_out_unsign_extended[15:12]; // always 0
			hun_reg <= PC_out_unsign_extended[11:8];  // always 0
			ten_reg <= PC_out_unsign_extended[7:4];
			one_reg <= PC_out_unsign_extended[3:0];
		end
		else begin
			// sys status 2: pause processor; inspect Reg File content
			clkRF_reg <= clkSSD;	// 500 Hz
			clk_reg <= 1'b0;		// freeze at 0
			multi_purpose_read_addr_reg  <=  SwitchSelector;  //  reg-file-
port1 reads from SwitchSelector
			// Reg-file is not freezed in time, this protects against RF-
data-overwrite
			multi_purpose_RegWrite_reg <= 1'b0;
			// output reg file content to SSD, but only the lower 16 bits
(we only have 4 SSD)
			tho_reg <= reg_read_data_1[15:12];
			hun_reg <= reg_read_data_1[11:8];
			ten_reg <= reg_read_data_1[7:4];
			one_reg <= reg_read_data_1[3:0];
		end
	end

endmodule

// IF/ID stage register
// update content & output updated content at rising edge
module	IF_ID_Stage_Reg	(PC_plus4_in,	PC_plus4_out,	instruction_in,
instruction_out, IF_ID_Write, IF_Flush, clk, reset);
	// 1. data content
	input [31:0] PC_plus4_in, instruction_in;
	output [31:0] PC_plus4_out, instruction_out;
	// 2. hazard control
	// IF_ID_Write: sync; if (IF_ID_Write==1'b0), do not update content at
this rising edge
	// IF_Flush: sync; if (IF_Flush==1), clear ALL content, NOT ONLY control
signals
	input IF_ID_Write, IF_Flush;
	// 3. general contorl
	// reset: async; set all register content to 0
	input clk, reset;

	reg [31:0] PC_plus4_out, instruction_out;

	always @(posedge clk or posedge reset)
	begin
```

```
            if (reset==1'b1)
            begin
                  PC_plus4_out <= 32'b0;
                  instruction_out <= 32'b0;
            end
            else if (IF_Flush==1'b1)
            begin
                  PC_plus4_out <= 32'b0;
                  instruction_out <= 32'b0;
            end
            else if (IF_ID_Write==1'b1)
            begin
                  PC_plus4_out <= PC_plus4_in;
                  instruction_out <= instruction_in;
            end

      end

endmodule


// ID/EX stage register
// update content & output updated content at rising edge
module   ID_EX_Stage_Reg   (ID_Flush_lwstall,   ID_Flush_Branch,   RegWrite_in,
MemtoReg_in,  RegWrite_out,  MemtoReg_out,  Branch_in,  MemRead_in,  MemWrite_in,
Jump_in, Branch_out, MemRead_out, MemWrite_out, Jump_out, RegDst_in, ALUSrc_in,
RegDst_out,   ALUSrc_out,   ALUOp_in,   ALUOp_out,   jump_addr_in,   PC_plus4_in,
jump_addr_out,     PC_plus4_out,     reg_read_data_1_in,     reg_read_data_2_in,
immi_sign_extended_in,         reg_read_data_1_out,         reg_read_data_2_out,
immi_sign_extended_out,         IF_ID_RegisterRs_in,         IF_ID_RegisterRt_in,
IF_ID_RegisterRd_in,          IF_ID_RegisterRs_out,          IF_ID_RegisterRt_out,
IF_ID_RegisterRd_out,IF_ID_funct_in, IF_ID_funct_out,clk, reset);
      // 1. hazard control signal (sync rising edge)
      // if either ID_Flush_lwstall or ID_Flush_Branch equals 1,
      // then clear all WB, MEM and EX control signal to 0 on rising edge
      // do not need to clear addr, data or reg content
      input ID_Flush_lwstall, ID_Flush_Branch;
      // 2. WB control signal
      input RegWrite_in, MemtoReg_in;
      output RegWrite_out, MemtoReg_out;
      // 3. MEM control signal
      input Branch_in, MemRead_in, MemWrite_in, Jump_in;
      output Branch_out, MemRead_out, MemWrite_out, Jump_out;
      // 4. EX control signal
      input RegDst_in, ALUSrc_in;
      input [1:0] ALUOp_in;
      output RegDst_out, ALUSrc_out;
      output [1:0] ALUOp_out;
      // 5. addr content
      input [31:0] jump_addr_in, PC_plus4_in;
      output [31:0] jump_addr_out, PC_plus4_out;
      // 6. data content
      input        [31:0]        reg_read_data_1_in,        reg_read_data_2_in,
immi_sign_extended_in;
      output       [31:0]       reg_read_data_1_out,       reg_read_data_2_out,
immi_sign_extended_out;
      // 7. reg content
      input [4:0] IF_ID_RegisterRs_in, IF_ID_RegisterRt_in, IF_ID_RegisterRd_in;
      output       [4:0]       IF_ID_RegisterRs_out,       IF_ID_RegisterRt_out,
```

```
IF_ID_RegisterRd_out;
      input [5:0] IF_ID_funct_in;
      output [5:0] IF_ID_funct_out;
      // general signal
      // reset: async; set all register content to 0
      input clk, reset;

      reg RegWrite_out, MemtoReg_out;
      reg Branch_out, MemRead_out, MemWrite_out, Jump_out;
      reg RegDst_out, ALUSrc_out;
      reg [1:0] ALUOp_out;
      reg [31:0] jump_addr_out, PC_plus4_out;
      reg        [31:0]        reg_read_data_1_out,        reg_read_data_2_out,
immi_sign_extended_out;
      reg        [4:0]        IF_ID_RegisterRs_out,        IF_ID_RegisterRt_out,
IF_ID_RegisterRd_out;
      reg [5:0] IF_ID_funct_out;

      always @(posedge clk or posedge reset)
      begin
            if (reset == 1'b1)
            begin
                  RegWrite_out = 1'b0;
                  MemtoReg_out = 1'b0;
                  Branch_out = 1'b0;
                  MemRead_out = 1'b0;
                  MemWrite_out = 1'b0;
                  Jump_out = 1'b0;
                  RegDst_out = 1'b0;
                  ALUSrc_out = 1'b0;
                  ALUOp_out = 2'b0;
                  jump_addr_out = 32'b0;
                  PC_plus4_out = 32'b0;
                  reg_read_data_1_out = 32'b0;
                  reg_read_data_2_out = 32'b0;
                  immi_sign_extended_out = 32'b0;
                  IF_ID_RegisterRs_out = 5'b0;
                  IF_ID_RegisterRt_out = 5'b0;
                  IF_ID_RegisterRd_out = 5'b0;
                  IF_ID_funct_out = 6'b0;
            end
            else if (ID_Flush_lwstall == 1'b1)
            begin
                  RegWrite_out = 1'b0;
                  MemtoReg_out = 1'b0;
                  Branch_out = 1'b0;
                  MemRead_out = 1'b0;
                  MemWrite_out = 1'b0;
                  Jump_out = 1'b0;
                  RegDst_out = 1'b0;
                  ALUSrc_out = 1'b0;
                  ALUOp_out = 2'b0;
            end
            else if (ID_Flush_Branch == 1'b1)
            begin
                  RegWrite_out = 1'b0;
                  MemtoReg_out = 1'b0;
                  Branch_out = 1'b0;
                  MemRead_out = 1'b0;
```

```
                MemWrite_out = 1'b0;
                Jump_out = 1'b0;
                RegDst_out = 1'b0;
                ALUSrc_out = 1'b0;
                ALUOp_out = 2'b0;
            end
            else begin
                RegWrite_out = RegWrite_in;
                MemtoReg_out = MemtoReg_in;
                Branch_out = Branch_in;
                MemRead_out = MemRead_in;
                MemWrite_out = MemWrite_in;
                Jump_out = Jump_in;
                RegDst_out = RegDst_in;
                ALUSrc_out = ALUSrc_in;
                ALUOp_out = ALUOp_in;
                jump_addr_out = jump_addr_in;
                PC_plus4_out = PC_plus4_in;
                reg_read_data_1_out = reg_read_data_1_in;
                reg_read_data_2_out = reg_read_data_2_in;
                immi_sign_extended_out = immi_sign_extended_in;
                IF_ID_RegisterRs_out = IF_ID_RegisterRs_in;
                IF_ID_RegisterRt_out = IF_ID_RegisterRt_in;
                IF_ID_RegisterRd_out = IF_ID_RegisterRd_in;
                IF_ID_funct_out = IF_ID_funct_in;
            end

        end

endmodule

// EX/MEM stage register
// update content & output updated content at rising edge
module EX_MEM_Stage_Reg (EX_Flush, RegWrite_in, MemtoReg_in, RegWrite_out,
MemtoReg_out, Branch_in, MemRead_in, MemWrite_in, Jump_in, Branch_out,
MemRead_out, MemWrite_out, Jump_out, jump_addr_in, branch_addr_in,
jump_addr_out, branch_addr_out, ALU_zero_in, ALU_zero_out, ALU_result_in,
reg_read_data_2_in, ALU_result_out, reg_read_data_2_out, ID_EX_RegisterRd_in,
EX_MEM_RegisterRd_out, clk, reset);
        // 1. hazard control signal (sync rising edge)
        // if EX_Flush equals 1,
        // then clear all WB, MEM control signal to 0 on rising edge
        // do not need to clear addr or data content
        input EX_Flush;
        // 2. WB control signal
        input RegWrite_in, MemtoReg_in;
        output RegWrite_out, MemtoReg_out;
        // 3. MEM control signal
        input Branch_in, MemRead_in, MemWrite_in, Jump_in;
        output Branch_out, MemRead_out, MemWrite_out, Jump_out;
        // 4. addr content
        input [31:0] jump_addr_in, branch_addr_in;
        output [31:0] jump_addr_out, branch_addr_out;
        // 5. data content
        input ALU_zero_in;
        output ALU_zero_out;
        input [31:0] ALU_result_in, reg_read_data_2_in;
        output [31:0] ALU_result_out, reg_read_data_2_out;
        input [4:0] ID_EX_RegisterRd_in;
```

```
      output [4:0] EX_MEM_RegisterRd_out;
      // general signal
      // reset: async; set all register content to 0
      input clk, reset;

      reg RegWrite_out, MemtoReg_out;
      reg Branch_out, MemRead_out, MemWrite_out, Jump_out;
      reg [31:0] jump_addr_out, branch_addr_out;
      reg ALU_zero_out;
      reg [31:0] ALU_result_out, reg_read_data_2_out;
      reg [4:0] EX_MEM_RegisterRd_out;

      always @(posedge clk or posedge reset)
      begin
            if (reset == 1'b1)
            begin
              RegWrite_out <= 1'b0;
              MemtoReg_out <= 1'b0;
              Branch_out <= 1'b0;
              MemRead_out <= 1'b0;
              MemWrite_out <= 1'b0;
              Jump_out <= 1'b0;
              jump_addr_out <= 32'b0;
              branch_addr_out <= 32'b0;
              ALU_zero_out <= 1'b0;
              ALU_result_out <= 32'b0;
              reg_read_data_2_out <= 32'b0;
              EX_MEM_RegisterRd_out <= 5'b0;
            end
            else if (EX_Flush == 1'b1)
          begin
              RegWrite_out <= 1'b0;
              MemtoReg_out <= 1'b0;
              Branch_out <= 1'b0;
              MemRead_out <= 1'b0;
              MemWrite_out <= 1'b0;
              Jump_out <= 1'b0;
            end
            else begin
              RegWrite_out <= RegWrite_in;
              MemtoReg_out <= MemtoReg_in;
              Branch_out <= Branch_in;
              MemRead_out <= MemRead_in;
              MemWrite_out <= MemWrite_in;
              Jump_out <= Jump_in;
              jump_addr_out <= jump_addr_in;
              branch_addr_out <= branch_addr_in;
              ALU_zero_out <= ALU_zero_in;
              ALU_result_out <= ALU_result_in;
              reg_read_data_2_out <= reg_read_data_2_in;
              EX_MEM_RegisterRd_out <= ID_EX_RegisterRd_in;
            end

      end

endmodule

// MEM/WB stage register
// update content & output updated content at rising edge
```

```
module MEM_WB_Stage_Reg (RegWrite_in, MemtoReg_in, RegWrite_out, MemtoReg_out,
D_MEM_read_data_in, D_MEM_read_addr_in, D_MEM_read_data_out,
D_MEM_read_addr_out, EX_MEM_RegisterRd_in, MEM_WB_RegisterRd_out, clk, reset);
	// 1. WB control signal
	input RegWrite_in, MemtoReg_in;
	output RegWrite_out, MemtoReg_out;
	// 2. data content
	input [31:0] D_MEM_read_data_in, D_MEM_read_addr_in;
	output [31:0] D_MEM_read_data_out, D_MEM_read_addr_out;
	input [4:0] EX_MEM_RegisterRd_in;
	output [4:0] MEM_WB_RegisterRd_out;
	// general signal
	// reset: async; set all register content to 0
	input clk, reset;

	reg RegWrite_out, MemtoReg_out;
	reg [31:0] D_MEM_read_data_out, D_MEM_read_addr_out;
	reg [4:0] MEM_WB_RegisterRd_out;

	always @(posedge clk or posedge reset)
	begin
		if (reset == 1'b1)
		begin
			RegWrite_out <= 1'b0;
			MemtoReg_out <= 1'b0;
			D_MEM_read_data_out <= 32'b0;
			D_MEM_read_addr_out <= 32'b0;
			MEM_WB_RegisterRd_out <= 5'b0;
		end
		else begin
			RegWrite_out <= RegWrite_in;
			MemtoReg_out <= MemtoReg_in;
			D_MEM_read_data_out <= D_MEM_read_data_in;
			D_MEM_read_addr_out <= D_MEM_read_addr_in;
			MEM_WB_RegisterRd_out <= EX_MEM_RegisterRd_in;
		end

	end

endmodule

// forwarding unit
// all connecions are aynchronous; no clock signal is provided
// implement forwarding from EX stage as shown in "EX forward" attached in email
// implement forwarding from MEM stage as shown in "MEM forward" attached in
email
//		note the MEM forward is modified according to Zheng Gang's lecture
// module I/O was implemented according to Fig 4.56, attached in email as "4.56
forwarding diagram"
module	Forwarding_Control	(EX_MEM_RegisterRd,	MEM_WB_RegisterRd,
ID_EX_RegisterRs,	ID_EX_RegisterRt,	EX_MEM_RegWrite,
MEM_WB_RegWrite,IF_ID_RegisterRs,IF_ID_RegisterRt, ForwardA, ForwardB, ForwardC,
ForwardD);
	input	[4:0]	EX_MEM_RegisterRd,	MEM_WB_RegisterRd,	ID_EX_RegisterRs,
ID_EX_RegisterRt,IF_ID_RegisterRs,IF_ID_RegisterRt;
	input EX_MEM_RegWrite, MEM_WB_RegWrite;
	output ForwardC,ForwardD;
	output [1:0] ForwardA, ForwardB;
	reg [1:0] ForwardA, ForwardB;
```

```
	reg ForwardC,ForwardD;
	wire equal_EXMEM_rs,equal_EXMEM_rt,equal_MEMWB_rs,equal_MEMWB_rt;
	wire nonzero_EXMEM_rd,nonzero_MEMWB_rd;
	assign nonzero_EXMEM_rd=(EX_MEM_RegisterRd==0)?0:1;
	assign nonzero_MEMWB_rd=(MEM_WB_RegisterRd==0)?0:1;
	assign equal_EXMEM_rs=(EX_MEM_RegisterRd==ID_EX_RegisterRs)?1:0;
	assign equal_EXMEM_rt=(EX_MEM_RegisterRd==ID_EX_RegisterRt)?1:0;
	assign equal_MEMWB_rs=(MEM_WB_RegisterRd==ID_EX_RegisterRs)?1:0;
	assign equal_MEMWB_rt=(MEM_WB_RegisterRd==ID_EX_RegisterRt)?1:0;
	assign equal_WB_ID_rs=(MEM_WB_RegisterRd==IF_ID_RegisterRs)?1:0;
	assign equal_WB_ID_rt=(MEM_WB_RegisterRd==IF_ID_RegisterRt)?1:0;
	always@ (EX_MEM_RegWrite or MEM_WB_RegWrite or nonzero_EXMEM_rd or
nonzero_MEMWB_rd or equal_EXMEM_rs
	or equal_EXMEM_rt or equal_MEMWB_rs or equal_MEMWB_rt or equal_WB_ID_rs or
equal_WB_ID_rt)
	begin
		if(EX_MEM_RegWrite & nonzero_EXMEM_rd & equal_EXMEM_rs)
			ForwardA<=2'b10;
		else if (MEM_WB_RegWrite & nonzero_MEMWB_rd & equal_MEMWB_rs)
			ForwardA<=2'b01;
		else
			ForwardA<=2'b00;

		if(EX_MEM_RegWrite & nonzero_EXMEM_rd & equal_EXMEM_rt)
			ForwardB<=2'b10;
		else if (MEM_WB_RegWrite & nonzero_MEMWB_rd & equal_MEMWB_rt)
			ForwardB<=2'b01;
		else
			ForwardB<=2'b00;

		if(MEM_WB_RegWrite & nonzero_MEMWB_rd & equal_WB_ID_rs)
			ForwardC<=1;
		else
			ForwardC<=0;

	if(MEM_WB_RegWrite & nonzero_MEMWB_rd & equal_WB_ID_rt)
			ForwardD<=1;
		else
			ForwardD<=0;

	end
endmodule

// load word causes pipeline with EX/MEM -> EX forwarding scheme to stall for
ONE cycle
// all connections acynchronous; no clock signal is provided.
// In this module we are concerned with outputting 3 stall signals (don't care
about the ONE cycle).
// at next rising edge, ID_EX_MemRead will be flushed to 0, this module will
automatically stop stalling.
//
// stalling condition could be found in "stall_for_lw_Control code" attached in
email
// how to stall the pipeline:
//		1. set PCWrite and IF_ID_Write to 0
//		2. set ID_Flush_lwstall to 1 (modified from textbook MUX solution;
now we feed this signal to ID/EX stage register)
// IMPORTANT: if stalling condition is not met, set all 3 signals to the
opposite value!
```

```
module      stall_for_lw_Control      (ID_EX_RegisterRt,      IF_ID_RegisterRs,
IF_ID_RegisterRt, ID_EX_MemRead, PCWrite, IF_ID_Write, ID_Flush_lwstall);
      input [4:0] ID_EX_RegisterRt, IF_ID_RegisterRs, IF_ID_RegisterRt;
      input ID_EX_MemRead;
      output PCWrite, IF_ID_Write, ID_Flush_lwstall;
      wire equal_IDEXrt_IFIDrs,equal_IDEXrt_IFIDrt;
      assign equal_IDEXrt_IFIDrs=(ID_EX_RegisterRt==IF_ID_RegisterRs)?1:0;
      assign equal_IDEXrt_IFIDrt=(ID_EX_RegisterRt==IF_ID_RegisterRt)?1:0;
      reg PCWrite, IF_ID_Write, ID_Flush_lwstall;

      always@(ID_EX_MemRead or equal_IDEXrt_IFIDrs or equal_IDEXrt_IFIDrt)
      begin
            if(ID_EX_MemRead & (equal_IDEXrt_IFIDrs|equal_IDEXrt_IFIDrt))
            begin PCWrite<=0;IF_ID_Write<=0;ID_Flush_lwstall<=1; end
            else
            begin PCWrite<=1;IF_ID_Write<=1;ID_Flush_lwstall<=0; end
      end
endmodule


// this module flushes IF/ID, ID/EX and EX/MEM if branch OR jump is determined
viable at MEM stage
//          we previously assumed branch NOT-taken, so 3 next instructions need
to be flushed
//          we are pushig jump to MEM stage because there might be a jump
instruction right below our branch_not_taken assumption
//          so we need to wait for branch result to come out before executing
jump
//          and of course becuase of the wait, all jump need to flush the next 3
instrucitons
// all connecions are aynchronous; no clock signal is provided
module  branch_and_jump_hazard_control  (MEM_PCSrc,  IF_Flush,  ID_Flush_Branch,
EX_Flush);
      input MEM_PCSrc; // the PCSrc generated in MEM stage will be 1 if branch
is taken or a jump instruction is detected at MEM stage
      output IF_Flush, ID_Flush_Branch, EX_Flush;
      reg IF_Flush, ID_Flush_Branch, EX_Flush;
      always @(MEM_PCSrc)
      begin
            if(MEM_PCSrc)
            begin IF_Flush<=1; ID_Flush_Branch<=1; EX_Flush<=1; end
            else
            begin IF_Flush<=0; ID_Flush_Branch<=0; EX_Flush<=0; end
      end
endmodule

// this module is not shown in textbook
// all connections acynchronous; no clock signal is provided.
// this module is designed to merge branch(if taken) and jump instruction
// getting an output of PCSrc and a destintion PC address
module  jump_OR_branch  (Jump,  Branch_taken,  branch_addr,  jump_addr,  PCSrc,
addr_out);
      input Jump, Branch_taken;
      input [31:0] branch_addr, jump_addr;
      output PCSrc;
      output [31:0] addr_out;
      reg [31:0] addr_out;
      reg PCSrc;
      // only one of Jump or Branch_taken can be true in MEM in one cycle
```

```
	// so if Jump is true, assign jump_addr to addr_out, and set PCSrc to 1
	// and if Branch is true, assign branch_addr to addr_out, and set PCSrc to
1
	// if none of the two are true, set PCSrc to 0. addr_out could be
whatever.
	always @(Jump or Branch_taken or branch_addr or jump_addr)
	begin
		if(Branch_taken)
		begin addr_out<=branch_addr;PCSrc<=1; end
		else if (Jump)
		begin addr_out<=jump_addr;PCSrc<=1;end
		else
		begin PCSrc<=0; addr_out<=32'b0; end
	end

endmodule

// 32-bit 3-to-1 MUX for forwarding
// data input width: 3 32-bit
// data output width: 1 32-bit
// control: 2-bit
module Mux_32bit_3to1 (in00, in01, in10, mux_out, control);
	input [31:0] in00, in01, in10;
	output [31:0] mux_out;
	input [1:0] control;
	reg [31:0] mux_out;
	always @(in00 or in01 or in10 or control)
	begin
		case(control)
		2'b00:mux_out<=in00;
		2'b01:mux_out<=in01;
		2'b10:mux_out<=in10;
		default: mux_out<=in00;
		endcase
	end
endmodule

////////////////////////////////////////////////////////////////////////
// modules that require modification from single-cycle implementation ////
////////////////////////////////////////////////////////////////////////
// this one added a PCWrite signal.
// Stop writing on this rising edge if PCWrite equals 0
//
// original module description:
// rising-edge synchronous program counter
// output range: decimal 0 to 32 (== I-MEM height)
// data I/O width: 64 = 2^6
// async reset: set program counter to 0 asynchronously
module Program_Counter (clk, reset, PC_in, PC_out, PCWrite);
	input PCWrite; // new input for pipeline
	input clk, reset;
	input [6:0] PC_in;//at most 32 instructions
	output [6:0] PC_out;
	reg [6:0] PC_out;
	always @ (posedge clk or posedge reset)
	begin
		if(reset==1'b1)
			PC_out<=0;
		else if(PCWrite)
```

```
                PC_out<=PC_in;
      end
endmodule


////////////////////////////////////////////////////////////////////////////////
// modules that are direct copies from single cycle implementation    ////
////////////////////////////////////////////////////////////////////////////////
// async read I-MEM
// height: 64, width: 32 bits (as required by TA)
// PC input width: 64 = 2^6
// instruction output width: 32 bits (== I-MEM width)
// async reset: as specified in document "Project Two Specification (V3)",
//                      first reset all to 0, then hard-code instructions
module Instruction_Memory (read_addr, instruction, reset);
      input reset;
      input [6:0] read_addr;
      output [31:0] instruction;
      reg [31:0] Imemory [63:0];
      integer k;
      // I-MEM in this case is addressed by word, not by byte
      wire [4:0] shifted_read_addr;
      assign shifted_read_addr=read_addr[6:2];
      assign instruction = Imemory[shifted_read_addr];

      always @(posedge reset)
      begin

            for (k=16; k<64; k=k+1) begin// here Ou changes k=0 to k=16
                  Imemory[k] = 32'b0;
            end

            Imemory[0]   =   32'b00100000000010000000000000100000;   //addi   $t0,
$zero, 32
            Imemory[1]   =   32'b00100000000010010000000000110111;   //addi   $t1,
$zero, 55
            Imemory[2] = 32'b00000001000010011000000000100100; //and $s0, $t0,
$t1
            Imemory[3] = 32'b00000001000010011000000000100101; //or $s0, $t0,
$t1
            Imemory[4]   =   32'b10101100000100000000000000000100;   //sw   $s0,
4($zero)
            Imemory[5]   =   32'b10101100000100000000000000001000;   //sw   $t0,
8($zero)
            Imemory[6] = 32'b00000001000010011000100000100000; //add $s1, $t0,
$t1
            Imemory[7] = 32'b00000001000010011001000000100010; //sub $s2, $t0,
$t1
            Imemory[8] = 32'b00010010001100100000000000001001; //beq $s1, $s2,
error0
            Imemory[9]   =   32'b10001100000100010000000000000100;   //lw   $s1,
4($zero)
            Imemory[10]= 32'b00110010001100100000000001001000; //andi $s2, $s1,
48
            Imemory[11] =32'b00010010001100100000000000001001; //beq $s1, $s2,
error1
            Imemory[12]    =32'b10001100000100110000000000001000;    //lw    $s3,
8($zero)
            Imemory[13] =32'b00010010000100110000000000001010; //beq $s0, $s3,
```

```
error2
            Imemory[14]  =32'b00000010010100011010000000101010;  //slt  $s4,  $s2,
$s1 (Last)
            Imemory[15]  =32'b00010010100000000000000000001111;  //beq  $s4,  $0,
EXIT
            Imemory[16]  =32'b00000010001000001001000000100000;  //add  $s2,  $s1,
$0
            Imemory[17] =32'b00001000000000000000000000001110; //j Last
            Imemory[18]  =32'b00100000000010000000000000000000;  //addi  $t0,  $0,
0(error0)
            Imemory[19] =32'b00100000000010010000000000000000; //addi $t1, $0, 0
            Imemory[20] =32'b00001000000000000000000000011111; //j EXIT
            Imemory[21]  =32'b00100000000010000000000000000001;  //addi  $t0,  $0,
1(error1)
            Imemory[22] =32'b00100000000010010000000000000001; //addi $t1, $0, 1
            Imemory[23] =32'b00001000000000000000000000011111; //j EXIT
            Imemory[24]  =32'b00100000000010000000000000000010;  //addi  $t0,  $0,
2(error2)
            Imemory[25] =32'b00100000000010010000000000000010; //addi $t1, $0, 2
            Imemory[26] =32'b00001000000000000000000000011111; //j EXIT
            Imemory[27]  =32'b00100000000010000000000000000011;  //addi  $t0,  $0,
3(error3)
            Imemory[28] =32'b00100000000010010000000000000011; //addi $t1, $0, 3
            Imemory[29] =32'b00001000000000000000000000011111; //j EXIT

      end
endmodule

// 32-bit ALU for addition only
// data input width: 2 32-bit
// data output width: 1 32-bit, no "zero" output
// control: no control input, only addition operation implemeneted
// as specified in Fig 4.12
// attached in email as "Fig 4_12 ALU Control Input"
module ALU_add_only (inA, inB, add_out);
      input [31:0] inA, inB;
      output [31:0] add_out;
      assign add_out=inA+inB;
endmodule

// N-bit 2-to-1 Mux
// input: 2 N-bit input
// output: 1 N-bit output
// control: 1 bit
// possible value of N in single cycle: 5, 6, 32
module Mux_N_bit (in0, in1, mux_out, control);
      parameter N = 32;
      input [N-1:0] in0, in1;
      output [N-1:0] mux_out;
      input control;
      assign mux_out=control?in1:in0;
endmodule

// sync register file (write/read occupy half cycle each)
// height: 32 (from $0 to $ra), width: 32 bits
// write: on rising edge; data width 32 bit; address width 5 bit
// read: on falling edge; data width 32 bit; address width 5 bit
// control: write on rising edge if (RegWrite == 1)
// async reset: set all register content to 0
```

```
module Register_File (read_addr_1, read_addr_2, write_addr, read_data_1,
read_data_2, write_data, RegWrite, clk, reset);
      input [4:0] read_addr_1, read_addr_2, write_addr;
      input [31:0] write_data;
      input clk, reset, RegWrite;
      output [31:0] read_data_1, read_data_2;

      reg [31:0] Regfile [31:0];
      integer k;

      assign read_data_1 = Regfile[read_addr_1];
      assign read_data_2 = Regfile[read_addr_2];

      always @(posedge clk or posedge reset) // Ou combines the block of reset
into the block of posedge clk
      begin
            if (reset==1'b1)
            begin
                  for (k=0; k<32; k=k+1)
                  begin
                        Regfile[k] = 32'b0;
                  end
            end

            else if (RegWrite == 1'b1) Regfile[write_addr] = write_data;
      end

endmodule

// sign-extend the 16-bit input to the 32_bit output
module Sign_Extension (sign_in, sign_out);
      input [15:0] sign_in;
      output [31:0] sign_out;
      assign sign_out[15:0]=sign_in[15:0];
      assign sign_out[31:16]=sign_in[15]?16'b1111_1111_1111_1111:16'b0;
endmodule

// shift-left-2 for jump instruction
// input width: 26 bits
// output width: 28 bits
// fill the void with 0 after shifting
// we don't need to shift in this case, becasue the address of the instructions
// are addressed by words
module Shift_Left_2_Jump (shift_in, shift_out);
      input [25:0] shift_in;
      output [27:0] shift_out;
      assign shift_out[27:0]={shift_in[25:0],2'b00};
endmodule

// async control signal generation unit based on OpCode
// as specified in Fig 4.22
// attached in email as file "Fig 4_22 Single Cycle Control"
// input: 6 bits OpCode
// output: all 1 bit except ALUOp which is 2-bits wide
module Control (OpCode, RegDst, Jump, Branch, MemRead, MemtoReg, ALUOp,
MemWrite, ALUSrc, RegWrite);
      input [5:0] OpCode;
      output RegDst, Jump, Branch, MemRead, MemtoReg, MemWrite, ALUSrc,
RegWrite;
```

```
      output [1:0] ALUOp;

      assign
RegDst=(~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(~OpCode[1])&(~OpCode
[0]);//000000
      assign
Jump=(~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(OpCode[1])&(~OpCode[0]
);//000010
      assign
Branch=(~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(OpCode[2])&(~OpCode[1])&(~OpCode[
0]);//000100
      assign
MemRead=(OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(OpCode[1])&(OpCode[0
]);//100011
      assign
MemtoReg=(OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(OpCode[1])&(OpCode[
0]);//100011
      assign
MemWrite=(OpCode[5])&(~OpCode[4])&(OpCode[3])&(~OpCode[2])&(OpCode[1])&(OpCode[0
]);//101011
      assign
ALUSrc=((~OpCode[5])&(~OpCode[4])&(OpCode[3])&(~OpCode[2])&(~OpCode[1])&(~OpCode
[0]))                                                                          |
((~OpCode[5])&(~OpCode[4])&(OpCode[3])&(OpCode[2])&(~OpCode[1])&(~OpCode[0]))  |
((OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(OpCode[1])&(OpCode[0]))   |
(((OpCode[5])&(~OpCode[4])&(OpCode[3])&(~OpCode[2])&(OpCode[1])&(OpCode[0])));  /
/001000,001100,100011,101011
      assign
RegWrite=(~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(~OpCode[1])&(~OpCo
de[0])                                                                         |
((~OpCode[5])&(~OpCode[4])&(OpCode[3])&(~OpCode[2])&(~OpCode[1])&(~OpCode[0]))  |
((~OpCode[5])&(~OpCode[4])&(OpCode[3])&(OpCode[2])&(~OpCode[1])&(~OpCode[0]))   |
((OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(OpCode[1])&(OpCode[0]));//
000000,001000,001100,100011
      assign
ALUOp[1]=((~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(~OpCode[2])&(~OpCode[1])&(~OpC
ode[0]))|
((~OpCode[5])&(~OpCode[4])&(OpCode[3])&(OpCode[2])&(~OpCode[1])&(~OpCode[0]));//
000000, 001100(andi)
      assign                                                          ALUOp[0]=
((~OpCode[5])&(~OpCode[4])&(~OpCode[3])&(OpCode[2])&(~OpCode[1])&(~OpCode[0]))|
((~OpCode[5])&(~OpCode[4])&(OpCode[3])&(OpCode[2])&(~OpCode[1])&(~OpCode[0]));//
000100,001100(andi)
endmodule

// async control to generate ALU input signal
// as specified in Fig 4.12
// attached in email as file "Fig 4_12 ALU Control Input"
// input: 2-bit ALUOp control signal and 6-bit funct field from instruction
// output: 4-bit ALU control input
module ALUControl (ALUOp, funct, out_to_ALU);
      input [1:0] ALUOp;
      input [5:0] funct;
      output [3:0] out_to_ALU;

      assign out_to_ALU[3]=0;
      assign               out_to_ALU[2]=((~ALUOp[1])&(ALUOp[0]))             |
((ALUOp[1])&(~ALUOp[0])&(~funct[3])&(~funct[2])&(funct[1])&(~funct[0]))       |
((ALUOp[1])&(~ALUOp[0])&(funct[3])&(~funct[2])&(funct[1])&(~funct[0]));
```

```
      assign  out_to_ALU[1]=((~ALUOp[1])&(~ALUOp[0]))|((~ALUOp[1])&(ALUOp[0]))  |
((ALUOp[1])&(~ALUOp[0])&(~funct[3])&(~funct[2])&(~funct[1])&(~funct[0]))       |
((ALUOp[1])&(~ALUOp[0])&(~funct[3])&(~funct[2])&(funct[1])&(~funct[0]))|
((ALUOp[1])&(~ALUOp[0])&(funct[3])&(~funct[2])&(funct[1])&(~funct[0]));
      assign
out_to_ALU[0]=((ALUOp[1])&(~ALUOp[0])&(~funct[3])&(funct[2])&(~funct[1])&(funct[
0]))|((ALUOp[1])&(~ALUOp[0])&(funct[3])&(~funct[2])&(funct[1])&(~funct[0]));
endmodule

// 32-bit ALU
// data input width: 2 32-bit
// data output width: 1 32-bit and one "zero" output
// control: 4-bit
// zero: output 1 if all bits of data output is 0
// as specified in Fig 4.12
// attached in email as "Fig 4_12 ALU Control Input"
module ALU (inA, inB, alu_out, zero, control);
      input [31:0] inA, inB;
      output [31:0] alu_out;
      output zero;
      reg zero;
      reg [31:0] alu_out;
      input [3:0] control;
      always @ (control or inA or inB)
      begin
            case (control)
            4'b0000:begin zero<=0; alu_out<=inA&inB; end
            4'b0001:begin zero<=0; alu_out<=inA|inB; end
            4'b0010:begin zero<=0; alu_out<=inA+inB; end
            4'b0110:begin if(inA==inB) zero<=1; else zero<=0; alu_out<=inA-inB;
end
            4'b0111:begin  zero<=0;  if(inA-inB>=32'h8000_0000)  alu_out<=32'b1;
else alu_out<=32'b0; end// how to implement signed number
            default: begin zero<=0; alu_out<=inA; end
            endcase
      end
endmodule

// shift-left-2 for branch instruction
// input width: 32 bits
// output width: 32 bits
// fill the void with 0 after shifting
module Shift_Left_2_Branch (shift_in, shift_out);
      input [31:0] shift_in;
      output [31:0] shift_out;
      assign shift_out[31:0]={shift_in[29:0],2'b00};
endmodule

// rising edge sync-write, async-read D-MEM
// height: 64, width: 32 bits (from document "Project Two Specification (V3)")
// address input: 6 bits (64 == 2^6)
// data input/output: 32 bits
// write: on rising edge, when (MemWrite == 1)
// read: asynchronous, when (MemRead == 1)
module Data_Memory (addr, write_data, read_data, clk, reset, MemRead, MemWrite);
      input [7:0] addr;
      input [31:0] write_data;
      output [31:0] read_data;
      input clk, reset, MemRead, MemWrite;
```

```
	reg [31:0] DMemory [63:0];
	integer k;
	wire [5:0] shifted_addr;
	assign shifted_addr=addr[7:2];
	assign read_data = (MemRead) ? DMemory[shifted_addr] : 32'bx;

	always @(posedge clk or posedge reset)// Ou modifies reset to posedge
	begin
		if (reset == 1'b1)
			begin
				for (k=0; k<64; k=k+1) begin
					DMemory[k] = 32'b0;
				end

			end
		else
			if (MemWrite) DMemory[shifted_addr] = write_data;
	end
endmodule

////////////////////////////////////////////////////////////////////////////
// modules for slowing the clock and displaying on SSD                 ////
////////////////////////////////////////////////////////////////////////////
module Dff_asy (q, d, clk, rst);
	input d, clk, rst;
	output reg q;

	always @ (posedge clk or posedge rst)
		if (rst == 1) q <= 0;
		else q <= d;
endmodule

// The following modules implement SSD display & clock slow-down
module divide_by_500 (clock, reset, clock_out); // modified to divide-by-4 for
simulation
	parameter N = 9;
	input		clock, reset;
	wire		load, asyclock_out;
	wire		[N-1:0] Dat;
	output		clock_out;
	reg		[N-1:0] Q;
	assign		Dat = 9'b000000000;
	assign		load = Q[1] & Q[0]; // modified to load = 3 (DEC) for
simulation
	always @ (posedge reset or posedge clock)
	begin
		if (reset == 1'b1) Q <= 9'b000000000;
		else if (load == 1'b1) Q <= Dat;
		else Q <= Q + 1;
	end
	assign		asyclock_out = load;
	Dff_asy                                                          Unit_Dff
(.q(clock_out), .d(asyclock_out), .clk(clock), .rst(reset));
endmodule

module divide_by_100k (clock, reset, clock_out); // modified to divide-by-4 for
simulation
	parameter N = 17;
	input clock, reset;
```

```
	wire  load, asyclock_out;
	wire  [N-1:0] Dat;
	output	clock_out;
	reg   [N-1:0] Q;
	assign	Dat = 0;
	assign	load = Q[1] & Q[0]; // modified to load = 3 (DEC) for
simulation
	always @ (posedge reset or posedge clock)
	begin
		if (reset == 1'b1) Q <= 0;
		else if (load == 1'b1) Q <= Dat;
		else Q <= Q + 1;
	end
	assign	asyclock_out = load;
	Dff_asy                                                    Unit_Dff
(.q(clock_out), .d(asyclock_out), .clk(clock), .rst(reset));
endmodule

module Ring_4_counter(clock, reset, Q);
	input		clock, reset;
	output reg  [3:0]Q;

	always @(posedge clock or posedge reset)
	begin
		if (reset == 1) Q <= 4'b1110;
		else
		begin
			Q[3] <= Q[0];
			Q[2] <= Q[3];
			Q[1] <= Q[2];
			Q[0] <= Q[1];
		end
	end
endmodule

module ssd_driver (in_BCD, out_SSD);
	input [3:0] in_BCD; // input in Binary-Coded Decimal
	output [6:0] out_SSD; // output to Seven-Segment Display
	reg [6:0] out_SSD;
	always @(in_BCD) begin
		case (in_BCD)
		0:out_SSD=7'b0000001;
		1:out_SSD=7'b1001111;
		2:out_SSD=7'b0010010;
		3:out_SSD=7'b0000110;
		4:out_SSD=7'b1001100;
		5:out_SSD=7'b0100100;
		6:out_SSD=7'b0100000;
		7:out_SSD=7'b0001111;
		8:out_SSD=7'b0000000;
		9:out_SSD=7'b0000100;
		10:out_SSD=7'b0001000;
		11:out_SSD=7'b1100000;
		12:out_SSD=7'b0110001;
		13:out_SSD=7'b1000010;
		14:out_SSD=7'b0110000;
		15:out_SSD=7'b0111000;
			default out_SSD = 7'b1111111; // no ssd
		endcase
```

```
	end
endmodule

module choose_chathode(tho, hun, ten, one, AN, CA);
	input [6:0]tho;
	input [6:0]hun;
	input [6:0]ten;
	input [6:0]one;
	input [3:0]AN;
	output	[6:0]CA;
	assign CA = (AN==4'b1110) ? one : 7'bzzzzzzz,
		CA =	(AN==4'b1101) ? ten : 7'bzzzzzzz,
		CA = (AN==4'b1011) ? hun : 7'bzzzzzzz,
		CA =	(AN==4'b0111) ? tho : 7'bzzzzzzz;
endmodule

`timescale 1ns / 1ps

module test_bench;

	reg clkFast;
	reg reset;
	reg [4:0] SwitchSelector;
	reg switchRun;
	reg clkread;

	integer count;


	// Outputs
	wire [31:0]reg_read_data_1;

	// Instantiate the Unit Under Test (UUT)
	pipeline uut (
		.clkFast(clkFast),
		.reset(reset),
		.SwitchSelector(SwitchSelector),
		.switchRun(switchRun),
		.reg_read_data_1(reg_read_data_1));

	initial begin
		// Initialize Inputs
		count = 0;
		clkFast = 0;
		reset = 1;
		switchRun = 0;
		SwitchSelector = 5'd0;
		clkread = 0;
		#4 reset=0;
		#10000;
		#50 $stop;
	end

	always begin #1 clkFast=~clkFast; end
	always begin #107 clkread = ~clkread; end

	always @(posedge clkread)
	begin
		$display ("Time: %d", count);
```

```
            SwitchSelector = 5'd16;
        #10 $display ("[$s0] = %h", reg_read_data_1);
            SwitchSelector = 5'd17;
        #10 $display ("[$s1] = %h", reg_read_data_1);
            SwitchSelector = 5'd18;
        #10 $display ("[$s2] = %h", reg_read_data_1);
            SwitchSelector = 5'd19;
        #10 $display ("[$s3] = %h", reg_read_data_1);
            SwitchSelector = 5'd20;
        #10 $display ("[$s4] = %h", reg_read_data_1);
            SwitchSelector = 5'd21;
        #10 $display ("[$s5] = %h", reg_read_data_1);
            SwitchSelector = 5'd22;
        #10 $display ("[$s6] = %h", reg_read_data_1);
            SwitchSelector = 5'd23;
        #10 $display ("[$s7] = %h", reg_read_data_1);
            SwitchSelector = 5'd8;
        #10 $display ("[$t0] = %h", reg_read_data_1);
            SwitchSelector = 5'd9;
        #10 $display ("[$t1] = %h", reg_read_data_1);
            SwitchSelector = 5'd10;
        #10 $display ("[$t2] = %h", reg_read_data_1);
            SwitchSelector = 5'd11;
        #10 $display ("[$t3] = %h", reg_read_data_1);
            SwitchSelector = 5'd12;
        #10 $display ("[$t4] = %h", reg_read_data_1);
            SwitchSelector = 5'd13;
        #10 $display ("[$t5] = %h", reg_read_data_1);
            SwitchSelector = 5'd14;
        #10 $display ("[$t6] = %h", reg_read_data_1);
            SwitchSelector = 5'd15;
        #10 $display ("[$t7] = %h", reg_read_data_1);
            SwitchSelector = 5'd24;
        #10 $display ("[$t8] = %h", reg_read_data_1);
            SwitchSelector = 5'd25;
        #10 $display ("[$t9] = %h", reg_read_data_1);

        #2 switchRun = 1;
        #32 switchRun = 0;
        count = count + 1;
    end
endmodule
```