Project Report Outline

Tasks:

descriptions of all aspects of modeling and implementation:

---

1. overall hierarchical structure of modules:
   would be uploaded to WeChat Group
2. description of different modules and implementations

    1. for pc, instruction memory, mux, sign extention, adder, shifter, register file, control. alu, data_memory: a very brief description of its function (maybe attached with codes ?) and schematic diagram is okay

        - NOTE1: for data memory and instruction memory, it'd be better emphasize that the memory is actually designed to be word-addressable, namely the module is implemented by:

          ```
          reg     [7:0]   memory  [3:0][size-1:0];    // byte-addressable
          ```

          where the data memory size is 4 KB, and instruction memory size is 0.25 KB.

        - NOTE2: for register file, since we never need to extract only some bytes from it, it's designed to be only word-addressable, namely it's implemented by:

          ```
          reg     [31:0]  regMem  [31:0];           //  word-addressable
          ```

    2. for forwarding unit and data harzard detection unit, it's better to clarify their logic in report (?:

        - various cases for data hazard:

            1. ALU data hazard (handled by forwarding unit)
            2. load use data hazard (handled by hazard detection unit)
            3. branch data hazard (in total 4 cases to be considered, handled by hazard detection unit)

                - case 1

```
add $1, xx, xx
beq $1, xx, xx
```

how to solve: insert 1 stall, then refer to case 2.

- case 2

```
add $1, xx, xx
xxx xx, xx, xx  # any unrelated instruction
beq $1, xx, xx
```

how to solve: `ALUResult` forwarding from MEM stage to EX stage, through two muxes.

- case 3

```
lw  $1, xx (xx)
beq $1, xx, xx
```

how to solve: insert 1 stall to EX stage, then refer to case 4.

- case 4

```
lw  $1, xx (xx)
xx xx, xx, xx
beq $1, xx, xx
```

how to solve: insert 1 stall to EX stage.

- forwarding unit: pseudo-code

```
input:  EXMEMRegWrite, ExMemDst, IdExRs, IdExRt, MemWbDst, MemWbRegWrit
output: ForwardA, ForwardB

if      ( ExMemRegWrite && ExMemDst != 0 && ExMemDst == IdExRs )
// 1 & 2 rs hazard
    ForwardA = 2'b10;
else if ( MemWbRegWrite && MemWbDst != 0 && MemWbDst == IdExRs )
// 1 & 3 rs hazard
    ForwardA = 2'b01;
else
    ForwardA = 2'b00;

if      ( ExMemRegWrite && ExMemDst != 0 && ExMemDst == IdExRt )
// 1 & 2 rs hazard
    ForwardB = 2'b10;
else if ( MemWbRegWrite && MemWbDst != 0 && MemWbDst == IdExRt )
// 1 & 3 rt hazard
    ForwardB = 2'b01;
else
    ForwardB = 2'b00;
```

- data hazard unit:

```
input:  IFIDRt, IFIDRs, IDEXRt, IDEXMemRead, IDBranch, IDEXRegWrite,
        EXMEMRegWrite, EXMEMMemRead, EXDst, EXMEMDst, IDRt, IDRs;
output: IFIDWrite, IDEXFlush, forward1, forward2,
        PCWrite;

if (
    // load use data hazard
    ( IDEXMemRead && ( EXDst == IDRs || EXDst == IDRt )  ) ||
    // branch data hazard, case 1 and case 3
    ( IDEXRegWrite && IDBranch && ( EXDst == IDRs || EXDst == IDRt ) )
    // branch data hazard, case 4
    ( EXMEMMemRead && IDBranch && ( EXMEMDst == IDRs || EXMEMDst == IDR
)
        { PCWrite = 0; IFIDWrite = 0; IDEXFlusb = 1; }
else // no hazard
        { PCWrite = 1; IFIDWrite = 1; IDEXFlusb = 0; }

// branch data hazard, case 2
if ( EXMEMRegWrite && !EXMEMMemRead && IDBranch )
    if ( EXMEMDst == IDRs )     forward1 = 1;
    if ( EXMEMDst == IDRt )     forward2 = 1;
else
    { forward 1 = 0; forward2 = 0; }
```

3. how control hazard (beq, bne and j) are solved?

    - beq & bne:

        - detected and decided in ID stage (therefore may need to flush IF)
        - control generates signals: `Branch_Beq` and `Branch_Bne` according to instruction. if the instruction is `beq`, `Branch_Beq` is set to 1 while `Branch_Bne` is 0; if `bne`, `Branch_Bne` is set to 1 and `Branch_Beq` be 0. Otherwise, they are both 0
        - a comparator in ID stage gives signal `Equal`, which is taken by control unit
        - these signals together determines whether to branch or not
        - if branch, update next pc, flush IFID pipeline register and insert a bubble to ID stage.
        - pseudo-code:

```
if (
    ( Branch_Beq && Equal ) ||
    ( Branch_Bne && ~Equal )
)
        { IFID.Flush = 1; BranchTarget  = PC+4+4*Relative_Address; }
else
        { IFID.Flush = 0; BranchTarget  = PC+4; }
```

    - jump:

        - detected in ID stage (therefore may need to flush IF)
        - control generates signal `Jump`, which is set to 1 if it's a jump instruction
        - if jump, update next pc, flush IFID pipeline register and insert a bubble to ID stage.
        - pseudo-code:

```
if ( Jump )
    { IFID.Flush = 1; NextPC = JumpTarget; }
else
    { IFID.Flush = 0; NextPC = BranchTarget; }
    // BranchTarget is the signal determined in beq & bne section
```

4. one thing that worth attention:
   In both register file and data memory, data is only written at negative edge of clock, because that's safer (write data that arrives earrlier than control signal would not be accidentally written into the memory)

---

simulation results:

- simulation result txt file has been uploaded to WeChat Group
- emphasize that we correctly handle all the data hazards and control hazards
- the theoretically expected results are:

project2-expected results

#cycle	fetched operation in IF stage	IF	ID	EX	MEM	WB	changes in register file
1	1. addi $t0, $zero, 0x20	1	x	x	x	x	nothing
2	2. addi $t1, $zero, 0x37	2	1	x	x	x	nothing
3	3. and $s0, $t0, $t1	3	2	1	x	x	nothing
4	4. or $s0, $t0, $t1	4	3	2	1	x	nothing
5	5. sw $s0, 4($zero)	5	4	3	2	1	t0 = 0x20
6	6. sw $t0, 8($zero)	6	5	4	3	2	t1 = 0x37
7	7. add $s1, $t0, $t1	7	6	5	4	3	s0 = 0x20
8	8. sub $s2, $t0, $t1	8	7	6	5	4	s0 = 0x37, DataMemory[4] = 0x37
9	9. beq $s1, $s2, error0 (does not branch)	9	8	7	6	5	DataMemory[8] = 0x20

10	10. lw $s1, 4($zero)	10	9	8	7	6	nothing
11		10	9	nop	8	7	s1 = 0x57
12	11. andi $s2, $s1, 0x48	11	10	9	nop	8	s2 = 0xfffffe9
13	12. beq $s1, $s2, error1	12	11	10	9	nop	nothing
14		12	11	nop	10	9	nothing
15	13. lw $s3, 8($zero)	13	12	11	nop	10	s1 = 0x37
16		13	12	nop	11	nop	nothing
17	14. beq $s0, $s3, error2 (does not branch)	14	13	12	nop	11	s2 = 0x0
18	15. slt $s4, $s2, $s1 (Last)	15	14	13	12	nop	nothing
19		15	14	nop	13	12	nothing
20		15	14	nop	nop	13	s3 = 0x20
21	16. beq $s4, $0, EXIT (does not branch)	16	15	14	nop	nop	nothing
22	17. add $s2, $s1, $0	17	16	15	14	nop	nothing
23		17	16	nop	15	14	nothing
24	18. j Last	18	17	16	nop	15	s4 = 0x1

25	19. addi $t0, $0, 0(error0)	19	18	17	16	nop	nothing
26	15. slt $s4, $s2, $s1 (Last)	15	flushed (nop)	18	17	16	nothing
27	16. beq $s4, $0, EXIT (branch)	16	15	flushed (nop)	18	17	s2 = 0x37
28	17. add $s2, $s1, $0	17	16	15	flushed (nop)	18	nothing
29		17	16	nop	15	flushed (nop)	nothing
30	EXIT (nop)	EXIT (nop)	flushed (nop)	16	nop	15	s4 = 0x0
31	EXIT (nop)	EXIT (nop)	EXIT (nop)	flushed (nop)	16	nop	nothing
32	EXIT (nop)	EXIT (nop)	EXIT (nop)	EXIT (nop)	flushed (nop)	16	nothing
33	EXIT (nop)	EXIT (nop)	EXIT (nop)	EXIT (nop)	EXIT (nop)	flushed (nop)	nothing