Computation Structures — Tutorial 12

December 4, 2018

1 A β Machine with a 2-Stage Pipeline

1. Give a plausible implementation of `NOP()`.

   ***** Solution *****

   A possible (safe) implementation of the `NOP()` instruction could be: `ADD(R31, R31, R31)`.

2. Give a *software* solution to the jump problems for each of the following programs:

(a)
```
1 Main: ADDC(R31,0,R1)
2       ADDC(R31,2,R2)
3 Incr: ADDC(R1,5,R1)
4       SUBC(R2,1,R2)
5       BT(R2,Incr)
6 Oper: ADD(R1,R3,R3)
```

(b)
```
1 Cas1: ADDC(R1,4,R1)
2       SUBC(R2,12,R2)
3       CMPLT(R1,R2,R0)
4       BNE(R0,Cas2)
5       MULC(R1,5,R1)
6       BR(rtn)
7 Cas2: MULC(R2,5,R2)
8       BR(rtn)
```

***** Solution *****

1. Problem: on a machine with a pipeline, the next instruction execution begins before the previous one completes. This can lead to problems for codes features branching and jumping instructions.
   In this case, `ADD` would be executed even if `R2` is true, which would corrupt the register `R3` and put the program in an undesired state.

	t	$t+1$	$t+2$	$t+3$	$t+4$	$t+5$	$t+6$
IF	ADDC	ADDC	ADDC	SUBC	BT	ADD	...
EXE		ADDC	ADDC	ADDC	SUBC	BT	ADD ??

   Possible solutions:

   - Using a `NOP()` instruction between `BT` (line 5) and `ADD` (line 6). *Drawback*: some **clock-cycles** (one in this case) are **used for doing nothing**.
   - Re-ordering the instructions: move `ADDC(R1, 5, R1)` after `BT`. *Drawback*: error-prone, the next instruction should be chosen cautiously. Indeed, it must be taken in the **same block** as the branching operation and the **registers values should remain consistent**. Moreover, we lose portability as the code **doesn't work on a pipeline-less machine** anymore.

2 A β Machine with a 4-Stage Pipeline

1. Give a software *and* a hardware solution to the data conflicts problems for the following program:

```
ADD(R1,R2,R3)
SUB(R3,R4,R5)
MULC(R2,5,R17)
ADD(R5,R1,R1)
SUB(R17,R1,R17)
```

***** Solution *****

Problem(s):

(1) `SUB` needs the result of the preceeding `ADD` in `R3` but `ADD` is only at the `ALU` phase so the hasn't been written back yet.

(2) The second `ADD` needs the result of the preceeding `SUB` in `R5` but `SUB` is only at the `WB` phase so the result will only be written back at the next clock cycle.

(3) The second `SUB` needs the results of both the second `ADD` and `MULC` which are respectively at phases `ALU` and `WB`. Therefore, they haven't written back yet.

	t	$t+1$	$t+2$	$t+3$	$t+4$	$t+5$	$t+6$	$t+7$	$t+8$
IF	ADD	SUB	MULC	ADD	SUB	...	...	...	...
RF		ADD	SUB$^{(1)}$	MULC	ADD$^{(2)}$	SUB$^{(3)}$	...	...	...
ALU			ADD$^{(1)}$	SUB	MULC	ADD$^{(3)}$	SUB	...	...
WB/MEM				ADD	SUB$^{(2)}$	MULC$^{(3)}$	ADD	SUB	...

- *Software solution*: in this case, one cannot re-order the instructions (too many conflicts and too few instructions). One could use `NOP()` operations at the cost of doubling the number of instructions:

```
ADD(R1,R2,R3)
NOP()
NOP()
SUB(R3,R4,R5)
NOP()
MULC(R2,5,R17)
ADD(R5,R1,R1)
NOP()
NOP()
SUB(R17,R1,R17)
```

- *Hardware solution*: one could do the `NOP` insertion at the hardware level (still we double the execution time). Second solution would consist **in using bypasses**. We need a `ALU-out` bypass for problems `(1)` and `(3)` and a `WB-out` bypass for problems `(2)` and `(3)`.

2. Give a *hardware* solution to the data conflicts problems for the following program:

```
LD(R1,0,R4)
ADD(R1,R4,R5)
XOR(R3,R4,R6)
```

***** Solution *****

(1) `ADD` needs the value loaded in `R4` from memory by `LD`, but `R4` is only saved at step $t+4$ as the memory is in the phase `WB/MEM`.

(2) `XOR` needs the value loaded in `R4` from memory by `LD` (same reason as above).

	t	$t+1$	$t+2$	$t+3$	$t+4$	$t+5$
IF	LD	ADD	XOR	...	...	...
RF		LD	ADD$^{(1)}$	XOR$^{(2)}$	...	...
ALU			LD$^{(1)}$	ADD	XOR	...
WB/MEM				LD$^{(2)}$	ADD	XOR

- **Problem (2)** can be handled by using a `WB-out` bypass.
- **Problem (1)** cannot be handled by using a bypass as the value from the memory is nowhere on the path (because the memory is only queried in the last phase `WB/MEM`). The only (simple) solution is to introduce a `NOP` instruction and then use a `WB-out` bypass:

```
LD(R1,0,R4)
NOP()
ADD(R1,R4,R5)
XOR(R3,R4,R6)
```

3. If the β Machine features 2 *bypasses*, what will be the result stored at `0x1000` after the execution of the following program? Why?

```
ADDC(R31,3,R0)
SUBC(R0,1,R1)
MUL(R0,R1,R2)
XOR(R0,R2,R3)
ST(R3,0x1000,R31)
```

***** Solution *****

Thanks to the bypasses, there is no conflicts and the result is that same as if this code ran on a pipeline-less machine. The operation performed is $3 \oplus (3 \times 2)$:

$$\begin{array}{cccccl} & 0 & 1 & 1 & \\ \oplus & 1 & 1 & 0 & \\ \hline & 1 & 0 & 1 & = 5 \end{array}$$

	t	$t+1$	$t+2$	$t+3$	$t+4$	$t+5$	$t+6$	$t+7$	$t+8$
IF	ADDC	SUBC	MUL	XOR	ST	...	...	...	...
RF		ADDC	SUBC$^{(1)}$	MUL$^{(2)}$	XOR$^{(3)}$	ST$^{(4)}$	...	...	...
ALU			ADDC$^{(1)}$	SUBC$^{(2)}$	MUL$^{(3)}$	XOR$^{(4)}$	ST	...	...
WB/MEM				SUBC$^{(2)}$	SUBC	MUL	XOR	ST	...