컴퓨터구조 HW 3

201932045 지동환

1

(a)

이 중 Load와 Store 만이 data memory를 사용한다. 그러므로 $20 + 15 = 35\%$

(b)

모든 instruction이 instruction memory를 사용한다. 100%

(c)

R-type instruction만 sign extender를 사용하지 않는다.
$23 + 20 + 15 + 11 + 4 = 73\%$

(d)

sign extend는 항상 계산을 한다. 만약 output이 사용되지 않으면 사용되지 않고 그냥 넘어간다.

2

(a)

추가적인 logic blocks이 필요 없다.

(b)

Control name	Value
Branch	0 (false)
MemRead	0 (false)
MemtoReg	0
ALUOp	0010 (add)
MemWrite	0 (false)
ALUSrc	1
RegWrite	1

Don't care인 control signal은 없다.

3

(a)

1000ps에 latency 250ps가 더해진다. 1250ps

(b)

원래 실행하는 instruction의 수 : n

원래 실행 시간 : 1000ps * n

새롭게 실행하는 instruction의 수 : 0.96 * n

새로운 실행 시간 : 1250ps * (0.96) * n

둘의 비율 = 새로울 실행 시간 / 원래 실행 시간 = $\frac{1250 \times 0.96 \times n}{1000 \times n} = 1.2$

즉, 새로운 실행 시간이 오히려 길어졌다. Performance의 비율은 $1/1.2 \simeq 0.83$

speed up은 0.83배 되었다.

(c)

Multiplier를 추가함으로서 instruction 수가 4% 감소한다.

새로울 실행 시간 / 원래 실행 시간 = $\frac{x \times 0.96 \times n}{1000 \times n} \leq 1$ 이 되어야 한다.

$x \leq 1000/0.96 = 1041.67$ 이 되고, 가능한 x 중 가장 큰 수는 1041이다.

즉, 새롭게 추가된 multiplier는 41ps 이하의 delay를 가져야 Performance를 향상시킬 수 있다.

4

```
lwi.d   rd, rs1,  rs2
```

은 `Reg[rd] = Mem[Reg[rs1]+Reg[rs2]]` 이다.

Control name	Value
Branch	0 (false)
MemRead	1 (true)
MemtoReg	1
ALUOp	0010 (add)
MemWrite	0 (false)
ALUSrc	0
RegWrite	1 (true)

로 Control signal이 설정된다.

5

```
logic          branch, branch_true;
logic          alu_zero;       // zero detection in ALU
result
logic [63:0]   alu_result;     // ALU result
```

```
assign branch = (opcode==7'b1100011) ? 1'b1: 1'b0;

always_comb begin
    case (funct3[2:0])
        3'b000: branch_true = alu_zero;
        3'b001: branch_true = ~alu_zero;
// FILL THIS
        3'b100: branch_true = (alu_result < 0);
// FILL THIS
        3'b101: branch_true = (alu_result > 0 ||
alu_zero);     // FILL THIS
        default: branch_true = 1'b0;
    endcase
end
assign pc_next = (branch & branch_true) ? pc_next_branch:
pc_next_plus4;
```

6

(a)

non-pipelined processor : 모든 과정이 한 1 cycle 안에 진행된다.

그러므로, $250 + 400 + 250 + 300 + 200 = 1400$ps

pipelined processor : 가장 실행 시간이 긴 stage (여기서는 ID)의 실행 시간에 따라 결정된다.

그러므로, 400ps

(b)

non-pipelined processor : instruction의 종류에 상관 없이 일정한 시간이 걸린다.

그러므로, 1400ps

pipelined processor : IF -> ID -> EX -> MEM -> WB의 과정을 모두 거쳐야 한다.

Clock cycle은 400ps. 그러므로, 2000ps가 걸린다.

(c)

가장 latency가 큰 stage가 Performance를 결정한다 => ID를 쪼개야 한다.

ID를 반으로 쪼개 각각 200ps의 latency를 가지게 했다.

새로운 clock cycle time은 새로운 datapath에서 가장 긴 latency를 가진 MEM에 의해 결정된다.

그러므로, 300ps

(d)

Load와 Store가 data memory를 사용한다.

그러므로, $20 + 15 = 35\%$

(e)

ALU/logic과 Load가 Write-register port를 사용한다.

그러므로, $45 + 20 = 65\%$

7

표를 통해서 생각해 보자.

N instruction	K stage
1	k
2	k+1
3	k+2
...	...
n	k+n-1

재귀적으로 생각할 수 있다.

N개의 instruction 실행 시간을 알고 싶다. T_N

N-1개의 instruction 실행 시간을 T_{N-1} 이라고 하고, 이 값을 알고 있다고 생각하자.

이론상 최소한의 cycles == stall, hazards가 없는 상황. N-1번째 instruction이 두 번째 stage에 들어갔을 때, N번째 instruction이 첫번째 stage에 들어간다. 그러므로, 임의의 N에 대해서 $T_N - T_{N-1}$ 은 반드시 1.

$$T_N = T_{N-1} + 1$$

의 관계를 가진다.

$T_1 = k$ 임을 쉽게 알 수 있다. 등차 수열의 일반항을 구하면, $T_N = n + k - 1$ 이 된다.

8

```
ld    x4,   0(x5)
add   x11,  x4,   x6
add   x11,  x11,  x11
add   x12,  x11,  x4
```

(a)

```
ld    x4,   0(x5)
NOP
NOP
add   x11,  x4,   x6
NOP
NOP
add   x11,  x11,  x11
NOP
NOP
add   x12,  x11,  x4
```

(b)

```
ld    x4,   0(x5)
NOP
add   x11,  x4,   x6
NOP
NOP
add   x11,  x11,  x11
NOP
NOP
add   x12,  x11,  x4
```

(c)

```
ld    x4,   0(x5)
NOP
add   x11,  x4,   x6
add   x11,  x11,  x11
add   x12,  x11,  x4
```

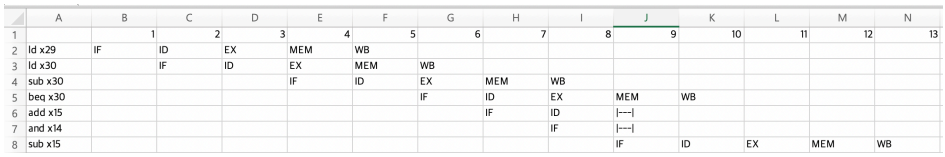
9

```
ld    x29,  12(x16)
ld    x30,  8(x16)
sub   x30,  x29,  x30
beq   x30,  x5, label
add   x15,  x11,  x14
and   x14,  x4,   x5
label:
sub x15,  x30,  x14
```

(a)

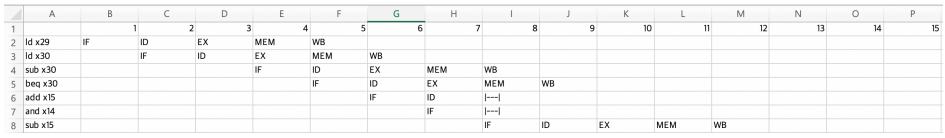
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	ld x29	IF	ID	EX	MEM	WB										
3	ld x30		IF	ID	EX	MEM	WB									
4	sub x30					IF	ID	EX	MEM	WB						
5	beq x30								IF	ID	EX	MEM	WB			
6	add x15									IF	ID	|---|				
7	and x14										IF	|---|				
8	sub x15											IF	ID	EX	MEM	WB

(b)



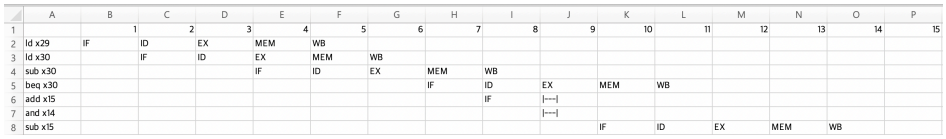
	1	2	3	4	5	6	7	8	9	10	11	12	13
ld x29	IF	ID	EX	MEM	WB								
ld x30		IF	ID	EX	MEM	WB							
sub x30				IF	ID	EX	MEM	WB					
beq x30						IF	ID	EX	MEM	WB			
add x15							IF	ID	|---|				
and x14								IF	|---|				
sub x15									IF	ID	EX	MEM	WB

(c)



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ld x29	IF	ID	EX	MEM	WB										
ld x30		IF	ID	EX	MEM	WB									
sub x30				IF	ID	EX	MEM	WB							
beq x30					IF	ID	EX	MEM	WB						
add x15						IF	ID	|---|							
and x14							IF	|---|							
sub x15								IF	ID	EX	MEM	WB			

(d)



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ld x29	IF	ID	EX	MEM	WB										
ld x30		IF	ID	EX	MEM	WB									
sub x30				IF	ID	EX	MEM	WB							
beq x30							IF	ID	EX	MEM	WB				
add x15								IF	|---|						
and x14									|---|						
sub x15										IF	ID	EX	MEM	WB	

10

(a)

Clock cycle time이 감소하기 위해서는, 가장 긴 stage의 latency가 감소해야 한다.

두 stage를 합하면서, 가장 긴 stage의 latency의 감소는 있을 수 없다. 그러므로, Clock cycle time의 감소는 없다.

(b)

5단계였던 때에는 load가 MEM 이후에 값을 알 수 있어서 Clock cycle을 1개 낭비할 수 밖에 없었는데,

EX + MEM으로 실행된다면 이 낭비를 없앨 수 있다.

(c)

`ld x2, 16(x5)` 같이 offset에 더해서 load하는 명령어들이 addi를 한 후, ld 하는 두 가지 Instruction으로 쪼개져야 한다. 그래서 실행해야 하는 전체 instruction 수가 늘어날 수도 있다.

11

(a)

mispredicted 한다면, branch instruction이 EX 단계에 들어 갔을 때, IF, ID에 존재하던 instruction을 Flush 해야해서, 한번 예측이 틀릴 때마다 실행해야 하는 instruction의 수가 2 늘어난다. -> ID 단계에서 branch outcome이 나오고, EX에서 적용되기 때문.

Always-taken으로 증가하는 CPI는,

$1 + (1 - 0.40) \times (0.20) \times 2 = 1.24$ 이다.

(b)

같은 방식으로 계산할 수 있다.

Always-Not-Take로 증가하는 CPI는,

$1 + (1 - 0.60) \times (0.20) \times 2 = 1.16$ 이다.

(c)

같은 방식으로 계산할 수 있다.

2-bit predictor로 증가하는 CPI는,

$1 + (1 - 0.90) \times (0.20) \times 2 = 1.04$ 이다.

12

(a)

For the first loop :

Iteration	**1**	**2**	**3**	**...**	**100**
Prediction	0	1	1		1
Real	1	1	1		0

Hit ratio = $(100-2)/100 = 98\%$

For the second loop :

Iteration	**1**	**2**	**3**	**...**	**100**
Prediction	0	1	1		1
Real	1	1	1		0

Iteration	**101**	**102**	**103**	**...**	**200**
Prediction	0	1	1		1
Real	1	1	1		0

.
.
.

Iteration	**901**	**902**	**903**	**...**	**1000**
Prediction	0	1	1		1
Real	1	1	1		0

Hit ration = $(10000-200)/10000 = (100-2)/100 = 98\%$

(b)

For the first loop :

Iteration	**1**	**2**	**3**	**...**	**100**
Prediction	00 -> 0	01 -> 0	11 -> 1		11 -> 1
Real	1	1	1		0

Hit ratio $= (100 - 3)/100 = 97\%$

For the second loop :

Iteration	**1**	**2**	**3**	**...**	**100**
Prediction	00 -> 0	01 -> 0	11 -> 1		11 -> 1
Real	1	1	1		0

Iteration	**101**	**102**	**103**	**...**	**200**
Prediction	10 -> 1	11 -> 1	11 -> 1		11 -> 1
Real	1	1	1		0

.
.
.

Iteration	**1**	**2**	**3**	**...**	**100**
Prediction	10 -> 1	11 -> 1	11 -> 1		11 -> 1
Real	1	1	1		0

Hit ratio $= (10000 - 102)/10000 = 98.98\%$
