컴퓨터구조 HW 2

201932045 지동환

1

```
slli  x30, x28, 2     ; x30 = i * 4
sub   x30, x30, x29   ; x30 = i * 4 - j
slli  x30, x30, 3     ; x30 = (i * 4 - j) * 8
add   x30, x10, x30   ; x30 = &A[0] + (i * 4 - j) * 8
ld    x30, 0(x30)     ; x30 = Memory[&A[0] + (i * 4 - j) * 8]
sd    x30, 32(x11)    ; Memory[&B[0] + 32] = Memory[&A[0] + (i * 4 - j) * 8]
```

2

```
slli  x30, x5, 3      ; x30 -> f * 8
add   x30, x10, x30   ; x30 -> x10 (&A[0]) + x30 (f * 8 ) = &A[f]
slli  x31, x6, 4      ; x31 -> g * 16
add   x31, x11, x31   ; x31 -> x11 (&B[0]) + x31 (g * 16) = &B[2 * g]
ld    x5, 0(x30)      ; x5 (f) -> *(&A[f]) = A[f]
addi  x12, x30, 8     ; x12 -> x30 (&A[f]) + 8 = &A[f+1]
ld    x30, 0(x12)     ; x30 -> *(&A[f+1]) = A[f+1]
sub   x30, x30, x5    ; x30 -> x30 (A[f+1]) - x5 (A[f])
sd    x30, 0(x31)     ; *(&B[2 * g]) -> x30 (A[f+1] - A[f])
```

In C code :

```
B[g * 2] = A[f + 1] - A[f]
```

3

(a)

```
for (i = 9; i >= 0; i--) {
  f = A[ i ];
  A[ i ] = B[ i ];
  B[ i ] = f;
}
```

x5 = f

x6 = i

x10 = &A[0]

x11 = &B[0]

x28,x39,x30,x31은 temp

```
  li    x6, 9           ; i = 9
loop1:
  slli  x28, x6, 3      ; x28 = i * 8
  add   x29, x10, x28   ; x29 = &A[0] + i * 8 (Address of A[ i ])
  add   x30, x11, x28   ; x30 = &B[0] + i * 8 (Address of B[ i ])
  ld    x5, 0(x29)      ; x5 (f) = Memory[ &A[0] + i * 8 ] = A[ i ]
  ld    x31, 0(x30)     ; x31 = Memory[ &B[0] + i * 8 (Address of B[ i ]) ] = B[ i ]
  sd    x31, 0(x29)     ; Memory[ &A[0] + i * 8 ] = A[ i ] = x31 = Memory[ &B[0] + i
* 8 (Address of B[ i ]) ] = B[ i ]
  sd    x5, 0(x30)      ; Memory[ &B[0] + i * 8 (Address of B[ i ]) ] = B[ i ] = x5
(f) = Memory[ &A[0] + i * 8 ] = A[ i ]
  addi  x6, x6, -1      ; x6 (i) = i - 1
  bge   x6, x0, loop1   ; if x6 (i) >= x0 (0) go to loop1
```

(b)

```
pB = &B[9];
for (pA = &A[9]; pA >= &A[0]; pA = pA - 1) {
  f = *pA
  *pA = *pB;
  *pB = f;
  pB = pB - 1;
}
```

x5 = f

x6 = i

x10 = &A[0]

x11 = &B[0]

x28,x39,x30,x31은 temp

```
  addi  x28, x11, 72      ; x28 (pB) = &B[ 0 ] + 8 * 9 = Address of B[9]
  addi  x29, x10, 72      ; x29 (pA) = &A[ 0 ] + 8 * 9 = Address of A[9]
loop2:
  ld    x5, 0(x29)        ; x5 (f) = Memory[&A[ 0 ] + 8 * 9 ] = A[9] (*pA)
  ld    x30, 0(x28)       ; x30 = Memory[&B[ 0 ] + 8 * 9 ] = B[9] (*pB)
  sd    x30, 0(x29)       ; Memory[&A[ 0 ] + 8 * 9 ] = A[9] (*pA) = B[9] (*pB)
  sd    x5, 0(x28)        ; Memory[&B[0] + 8 * 9] = B[9] (*pB) = x5 (f)
  addi  x28, x28, -8      ; x28 (pB) = x28 (pB) - 8
  addi  x29, x29, -8      ; x29 (pA) = x29 (pA) - 8
  bgeu  x29, x10, loop2   ; if pA >= &A[0] then loop2
```

(c)

loop1, loop2는 각각 10번 씩 돈다.

(a)의 코드에서 실행되는 instruction을 계산하면,

1 + 10 * 9 = 91

(b)의 코드에서 실행되는 instruction을 계산하면,

2 + 10 * 6 = 62

모든 instruction의 CPI가 같다. 같은 Clock rate = k로 돌아간다면,

(b)의 실행 시간 = 62C

(a)의 실행 시간 = 91C가 되어서,

(b)가 (a)보다 performance $\frac{91}{62} \simeq 1.47$배가 된다. 47% 성능이 향상된다.

4

x5 = f
x6 = g
x7 = h
x28 = i
x29 = j

&A[0] = x10
&B[0] = x11

Code in assembly

```
addi  x30, x10, 16  ; x30 = x10 + 16 = &A[0] + 16 = &A[2]
addi  x31, x10, 8   ; x31 = x10 + 8 = &A[0] + 8 = &A[1]
ld    x5, 0(x30)    ; x5 = f ->  Memory[x30] = *(&A[2]) = A[2]
ld    x30, 8(x31)   ; x30 -> Memory[x31 + 8] = Memory[&A[1] + 8] = Memory[&A[2]] =
A[2]
add   x30, x30, x5  ; x30 -> x30 (A[2]) + x5 (A[2])
subi  x5, x30, 16   ; x5 = f -> x30 - 16 = A[2] + A[2] - 16
```

In C code

```
f = A[1 + 1] + A[2] - 16;
```

5

x5 = f
x6 = g
x7 = h
x28 = i
x29 = j

x30 = temporary value

(a)

```
sub   x6, x0, x5  ; x6 = g -> 0 - x5 = -f
```

(b)

```
slli  x30, x5, 3  ; x30 -> x5 (f) * 8
sub   x6, x30, x5 ; x6 = g -> x30 (f * 8) - x5 (f) = 7 * f
```

(c)

```
srli  x30, x5, 4  ; x30 -> x5 (f) / 16
slli  x30, x30, 4 ; x30 -> f / 16 * 16
sub   x6, x5, x30 ; x6 = g -> x5 (f) - f / 16 * 16 = f % 16
```

6

x5 = 0x000;0 0000 0000 0000

x6 = 0x0000 0000 0013 FF0F

(a)

```
LOOP:

beq   x6, x0, DONE  ; x6과 0이 다르다. DONE 실행 안 됨
srli  x7, x6, 1     ; x7에 x6 / 2를 넣는다. x7 = 0x0000 0000 0009 FF87
and   x6, x6, x7    ; x6 -> x6 & x7 = 0x0000 0000 0001 FF07
addi  x5, x5, 1     ; x5 = 0x0000 0000 0000 0001
jal   x0, LOOP      ; LOOP로 돌아가기. x5 = 0x1, x6 = 0x0001 FF07, x7 = 0x0009 FF97
```

```
LOOP:

beq   x6, x0, DONE  ; x6이 0이 아님.
srli  x7, x6, 1     ; x7 = 0xFF83
and   x6, x6, x7    ; x6 -> x6 & x7 = 0xFF03
addi  x5, x5, 1     ; x5 = 0x02
jal   x0, LOOP      ; LOOP 복귀. x5 = 0x2, x6 = 0xFF03

LOOP:

beq   x6, x0, DONE  ; x6이 0이 아님.
srli  x7, x6, 1     ; x7 = 0x7F81
and   x6, x6, x7    ; x6 -> x6 & x7 = 0x7F01
addi  x5, x5, 1     ; x5 -> 0x03
jal   x0, LOOP      ; LOOP. x5 = 0x03, x6 = 0x7F01

LOOP:

beq   x6, x0, DONE  ; x6 = 0x7F01, x5 = 0x03
srli  x7, x6, 1     ; x7 = 0x3F80
and   x6, x6, x7    ; x6 = 0x7F01 & 0x3F80 = 0x3F00
addi  x5, x5, 1     ; x5 = 0x04
jal   x0, LOOP

LOOP:

beq   x6, x0, DONE  ; x6 = 0x3F00, x5 = 0x04
srli  x7, x6, 1     ; x7 = 0x1F80
and   x6, x6, x7    ; x6 = 0x1F00
addi  x5, x5, 1     ; x5 = 0x05
jal   x0, LOOP

LOOP:

beq   x6, x0, DONE  ; x6 = 0x1F00, x5 = 0x05
srli  x7, x6, 1     ; x7 = 0x0F80
and   x6, x6, x7    ; x6 = 0x0F00
addi  x5, x5, 1     ; x5 = 0x06
jal   x0, LOOP

LOOP:

beq   x6, x0, DONE  ; x6 = 0x0F00, x5 = 0x06
srli  x7, x6, 1     ; x7 = 0x0780
and   x6, x6, x7    ; x6 = 0x0700
addi  x5, x5, 1     ; x5 = 0x07
jal   x0, LOOP

LOOP:

beq   x6, x0, DONE  ; x6 = 0x0700, x5 = 0x07
srli  x7, x6, 1     ; x7 = 0x0380
```

```
and   x6, x6, x7    ; x6 = 0x0300
addi  x5, x5, 1     ; x5 = 0x08
jal   x0, LOOP

LOOP:

beq   x6, x0, DONE  ; x6 = 0x0300, x5 = 0x08
srli  x7, x6, 1     ; x7 = 0x0180
and   x6, x6, x7    ; x6 = 0x0100
addi  x5, x5, 1     ; x5 = 0x09
jal   x0, LOOP

LOOP:

beq   x6, x0, DONE  ; x6 = 0x0100, x5 = 0x09
srli  x7, x6, 1     ; x7 = 0x80
and   x6, x6, x7    ; x6 = 0x0
addi  x5, x5, 1     ; x5 = 0x0A
jal   x0, LOOP

LOOP:

beq   x6, x0, DONE   ; x5 = 0x0A, x6 = 0x0, x7 = 0x80. DONE으로 이동.
DONE
```

x5 = 0x0000 0000 0000 000A

x6 = 0x0000 0000 0000 0000

x7 = 0x0000 0000 0000 0080

(b)

11번 실행되었다. LOOP가 10번 돌았고, 마지막에 한 번 더 beq한 후 DONE으로 이동.

7

PC = 0x3000 0000

(a)

jal은 UJ-format을 사용한다. 들어갈 수 있는 immediate 값의 길이가 20 bits이다.

주소는 signed bit를 사용하고, 왼쪽으로 shift를 한 번 한 후 사용되기 때문에 immediate 값으로 가능한 숫자 범위는

1000 0000 0000 0001 ($= -2^{19}$) ~ 0111 1111 1111 1110 ($= 2^{19} - 1$) 이 된다. (주소는 반드시 짝수 값)이고,

주소값의 범위는 $[-2^{20}, 2^{20} - 1]$ 이다.

0x3000 0000 - 0x000F FFFE = 0x2FF0 0000

0x3000 0000 + 0x000F FFFE = 0x300F FFFE

[0x2FF00000, 0x300FFFFE]이다.

(b)

beq 같은 branch instruction은 SB-format을 사용한다. 들어갈 수 있는 immediate value가 12 bits이다.

표현될 수 있는 값은 $-2^{12}, 2^{12} - 1$

즉, [-4096. 4094]만큼 주소가 변할 수 있다.

[0x2FFFF000, 0x30000FFE]

8

```
      addi  x5, x0, 0
      addi  x6, x0, 100
LOOP: blt   x6, x0, DONE
      addi  x6, x6, -1
      addi  x5, x5, 4
      jal   x0, LOOP
DONE:
```

(a)

x5 -> x0 + 0 = 0

x6 -> x0 + 100 = 100

LOOP: / / x5 = 0, x6 = 100

if x6 < x0 then DONE

x6 -> x6 - 1 = 99

x5 -> x5 + 4 = 4

jal x0, LOOP

LOOP: / / x5 = 4, x6 = 99

if x6 < x0 then DONE

x6 -> x6 - 1 = 98

x5 -> x5 + 4 = 8

jal x0, LOOP

.

.

.

LOOP: / / x5 = 396, x6 =1

if x6 < 0 then DONE

x6 -> x6 - 1 = 0

x5 -> x5 + 4 = 400

jal x0, LOOP

LOOP: / / x5 = 400, x6 = 0

if x6 < 0 then DONE

x6 -> x6 - 1 = -1

x5 -> x5 + 4 = 404

LOOP: / / x5 = 404, x6 = -1

if x6 < 0 then DONE

DONE:

최종적인 값은 404= 0x194,

(b)

```
i = 0;
for(j = 100; j >= 0;--j)
{
  i += 4;
}
```

9

```
for (i=0; i < a; i++)
  for (j=0; j < b; j+=2)
    D[2*j] = i + j;
```

x5 = a, x6 = b, x7 = i, x29 = j, x10 = &D[0], x30 / x31 temp

```
       addi   x7, x0, 0       ; x7 (i) = 0 + 0 = 0
LOOPI: bge    x7, x5, ENDI    ; if x7 (i) >= x5 (a) then go to ENDI
       addi   x31, x10, 0     ; x31 = x10 (&D[0]) + 0 = &D[0]
       addi   x29, x0, 0      ; x29 (j) = 0 + 0
LOOPJ: bge    x29, x6, ENDJ   ; if x29 (j) >= x6 (b) then go to ENDJ
       add    x30, x7, x29    ; x30 = x7 (i) + x29 (j)
       sd     x30, 0(x31)     ; Memory[x31] (D[2*j]) = x30 (i + j)
       addi   x31, x31, 16    ; x31 = x31 (&D[2*j]) + 16 = &D[2*(j+1)]
       addi   x29, 2          ; x29 (j) = j + 2
       jal    x0,  LOOPJ
ENDJ:  addi   x7,  1          ; x7 (x) = i + 1
       jal    x0,  LOOPI
ENDI:
```

10

0x1122334455667788 = 0001 0001 0010 0010 0011 0011 0100 0100 0101 0101 0110 0110 0111 0111 1000 1000_2

```
lui   x10,  0x11223       ; upper 20-bit constant
addi  x10,  x10,  0x344   ; upper 12-bit constant
slli  x10,  x10,  32      ; shift 32 bit
lui   x5,   0x55667       ; lower 20-bit
addi  x5,   x5, 0x788     ; lower 12-bit
add   x10,  x10,  x5      ; add upper and lower
```

11

arithmetic instruction = CPI 1, 800 million

load/store instruction = CPI 10, 300 million

branch instruction = CPI 3, 200 million

(a)

현재 필요한 Clock cycle

(800m * 1) + (300m * 10) + (200m * 3) = 4400million cycles

업그레이드 된 Clock cycle

(0.75 * 800m * 1) + (300m * 10) + (200m * 3) = 4200million cycles

업그레이드 된 arithmetic instruction에서 더 적은 cycles이 실행된다.

그러나, clock cycle time을 10% 증가시킨다.

그렇기 때문에 업그레이드 된 instruction set에서 4200 million cycles은

원래 컴퓨터에서 4200 * 1.1 = 4620 million cycles의 실행 시간과 같다. 즉, 실행 시간이 더 길어진다. 좋지 않은 선택이다.

(b)

현재 필요한 Clock cycle

(800m * 1) + (300m * 10) + (200m * 3) = 4400million cycles

arithmetic의 performance가 두배가 되었다면, 이의 실행 시간이 절반이 된 것

= arithmetic instruction의 CPI가 0.5이다.

(800m * 0.5) + (300m * 10) + (200m * 3) = 4000 million cycles

성능이 4400/4000 = 1.1배가 되었다. 10% 증가.

arithmetic의 performance가 10배가 되었다면, 이의 실행 시간이 1/10이 된 것

= arithmetic instruction의 CPI가 0.1이다.

(800m * 0.1) + (300m * 10) + (200m * 3) = 3680 million cycles

성능이 4400/3680 = 1.20배가 되었다. 20% 증가

12

x5 = A, x6 = B, x7 = i, x28 = res, x29/x30 = temporary

```
int i;
int res = 0;
for (i = 0; i < 32; i++) {
  if (B & 0x01)
    res += (A << i);
  B = B >> 1;
}
```

이를 어셈블리 코드로 바꾸면,

```
      addi  x7, x0, 0     ; x7 (i) = 0 + 0 = 0
      addi  x28, x0, 0    ; x28 (res) = 0 + 0 = 0
      addi  x29, x0, 32   ; x29 = 0 + 32 = 32
LOOP: bge   x7, x29, DONE ; if x7 (i) >= x29 (32) then go to DONE
      andi  x30, x6, 1    ; x30 = x6 (B) & 1
      beq   x30, x0, ELSE ; if x30 = 0 (B & 1 = false) then go to ELSE
      sll   x30, x5, x7   ; x30 = x5 (A) << i
      add   x28, x28, x30 ; x28 (res) = x28 (res) + x30 (A << i)
ELSE: srli  x6, x6, 1     ; x6 (B) = x6 (B) >> 1
      addi  x7, x7, 1     ; x7 (i) = x7 (i) + 1
      jal   x0, LOOP
DONE:
```
