20181614 김주연

Computer Architecture and Logic HW #2

Due:5/3(Sun) 23:59

Problem 1 [10points]

2.23 [5] <§2.7> Assume `$t0` holds the value `0x00101000`. What is the value of `$t2` after the following instructions?

```
        slt  $t2, $0,  $t0      # $0 < $t0, so $t2 = 1
        bne  $t2, $0,  ELSE     # $t2 != $0, so go to ELSE
        j    DONE
ELSE:   addi $t2, $t2, 2        # $t2 += 2, so $t2 = 1+2 = 3
DONE:
```

∴ $t2 = 3

2.24 [5] <§2.7> Suppose the program counter (PC) is set to `0x2000 0000`. Is it possible to use the jump (j) MIPS assembly instruction to set the PC to the address as `0x4000 0000`? Is it possible to use the branch-on-equal (beq) MIPS assembly instruction to set the PC to this same address?

0x2000 0000 = 0010 0000 0000 0000 0000 0000 0000 0000

0x4000 0000 = 0100 0000 0000 0000 0000 0000 0000 0000

∴ Jump instruction keeps the PC's leftmost 4 bits.
So it is not possible to use the jump(j) instruction.

∵ Branch instruction has 16 bits offset.
And 0x2000 0000 can't be expressed with 16 bits.
So it is not possible to use the branch instruction.

Problem 2 [15points]

2.39 [5] <§2.10> Write the MIPS assembly code that creates the 32-bit constant $0010\ 0000\ 0000\ 0001\ 0100\ 1001\ 0010\ 0100_{two}$ and stores that value to register `$t1`.

0010 0000 0000 0001 0100 1001 0010 0100 = 0x2001 4924

∴ lui $t1, 0x2001
ori $t1, $t1, 0X4924

2.41 [5] <§§2.6, 2.10> If the current value of the PC is `0x00000600`, can you use a single branch instruction to get to the PC address as shown in Exercise 2.39?

PC : 0x0000 0600

PC + 4 + 0x1FFFC = 0x0002 0600
PC + 4 – 0x20000 = 0xFFFE 0604
So, the branch address range is 0xFFFE 0604 ~ 0x0002 0600

∴ So we can't use a single branch instruction to get to 0x2001 4924.

2.42 [5] <§§2.6, 2.10> If the current value of the PC is `0x1FFFf000`, can you use a single branch instruction to get to the PC address as shown in Exercise 2.39?

PC : 0x1FFF F000

PC + 4 + 0x1FFFC = 0x2001 F000
PC + 4 – 0x20000 = 0x1FFD F004
So, the branch address range is 0x1FFD F004 ~ 0x2001 F000

∴ So we can use a single branch instruction to get to 0x2001 4924.

Problem 3 [10points]

2.46 Assume for a given processor the CPI of arithmetic instructions is 1, the CPI of load/store instructions is 10, and the CPI of branch instructions is 3. Assume a program has the following instruction breakdowns: 500 million arithmetic instructions, 300 million load/store instructions, 100 million branch instructions.

2.46.1 [5] <§2.19> Suppose that new, more powerful arithmetic instructions are added to the instruction set. On average, through the use of these more powerful arithmetic instructions, we can reduce the number of arithmetic instructions needed to execute a program by 25%, and the cost of increasing the clock cycle time by only 10%. Is this a good design choice? Why?

CPU time = Instruction Count * CPI * Clock Cycle Time

= (A*1 + L*10 + B*3) * Clock Cycle Time

new CPU time = (0.75*A*1 + L*10 + B*3) * 1.1*Clock Cycle Time

∴ No. New CPU time is not faster than old one because of the extra clock cycle time.

2.46.2 [5] <§2.19> Suppose that we find a way to double the performance of arithmetic instructions. What is the overall speedup of our machine? What if we find a way to improve the performance of arithmetic instructions by 10 times?

$$\frac{500M*1 + 300M*10 + 100M*3}{500M*1*0.5 + 300M*10 + 100M*3} \text{ X } 100 = 107.04\%$$

∴ 107.04%

$$\frac{500M*1 + 300M*10 + 100M*3}{500M*1*0.1 + 300M*10 + 100M*3} \text{ X } 100 = 113.43\%$$

∴ 113.43%

Problem 4 [30points]

3.6 [5] <§3.2> Assume 185 and 122 are unsigned 8-bit decimal integers. Calculate 185 – 122. Is there overflow, underflow, or neither?

```
  1011 1001 (185)
– 0111 1010 (122)
-------------------
  0011 1111 (63)
```

∴ neither(63)

3.7 [5] <§3.2> Assume 185 and 122 are signed 8-bit decimal integers stored in sign-magnitude format. Calculate 185 + 122. Is there overflow, underflow, or neither?

```
185 -> 1011 1001 (–57)
       0111 1010 (122)
     – 0011 1001 (+57)
-----------------------
       0100 0001 (65)
```

∴ neither(65)

3.18 [20] <§3.4> Using a table similar to that shown in Figure 3.10, calculate 74 divided by 21 using the hardware described in Figure 3.8. You should show the contents of each register on each step. Assume both inputs are unsigned ~~6-bit~~ 7-bit integers.

∴ 74/21 = 3 remainder 11

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000000	0010101 0000000	0000000 1001010
1	1: Rem = Rem – Div	0000000	0010101 0000000	1101011 1001010
	2b: Rem <0, –> +Div, sll Q, Q0 =0	0000000	0010101 0000000	0000000 1001010
	3: Shift Div right	0000000	0001010 1000000	0000000 1001010
2	1: Rem = Rem – Div	0000000	0001010 1000000	1110110 0001010
	2b: Rem <0, –> +Div, sll Q, Q0 =0	0000000	0001010 1000000	0000000 1001010
	3: Shift Div right	0000000	0000101 0100000	0000000 1001010
3	1: Rem = Rem – Div	0000000	0000101 0100000	1111011 0101010
	2b: Rem <0, –> +Div, sll Q, Q0 =0	0000000	0000101 0100000	0000000 1001010
	3: Shift Div right	0000000	0000010 1010000	0000000 1001010
4	1: Rem = Rem – Div	0000000	0000010 1010000	1111110 1111010
	2b: Rem <0, –> +Div, sll Q, Q0 =0	0000000	0000010 1010000	0000000 1001010
	3: Shift Div right	0000000	0000001 0101000	0000000 1001010
5	1: Rem = Rem – Div	0000000	0000001 0101000	1111111 0100010
	2b: Rem <0, –> +Div, sll Q, Q0 =0	0000000	0000001 0101000	0000000 1001010
	3: Shift Div right	0000000	0000000 1010100	0000000 1001010
6	1: Rem = Rem – Div	0000000	0000000 1010100	0000000 1110110
	2b: Rem <0, –> +Div, sll Q, Q0 =0	0000000	0000000 1010100	0000000 1001010
	3: Shift Div right	0000000	0000000 0101010	0000000 1001010
7	1: Rem = Rem – Div	0000000	0000000 0101010	0000000 0100000
	2b: Rem <0, –> +Div, sll Q, Q0 =0	0000001	0000000 0101010	0000000 0100000
	3: Shift Div right	0000001	0000000 0010101	0000000 0100000
8	1: Rem = Rem – Div	0000001	0000000 0010101	0000000 0100000
	2b: Rem <0, –> +Div, sll Q, Q0 =0	0000011	0000000 0010101	0000000 0001011
	3: Shift Div right	0000011	0000000 0001010	0000000 0001011

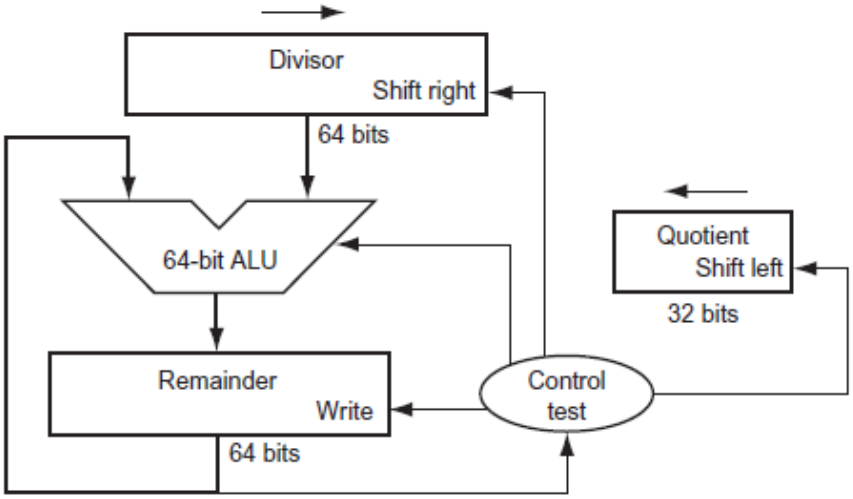

FIGURE 3.8 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	(1)110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	(1)111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	(1)111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	(0)000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	(0)000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

FIGURE 3.10 Division example using the algorithm in Figure 3.9. The bit examined to determine the next step is circled in color.

Problem 5 [20points]

3.22 [10] <§3.5> What decimal number does the bit pattern 0×0C000000 represent if it is a floating point number? Use the IEEE 754 standard.

0x0C00 0000 = 0000 1100 0000 0000 0000 0000 0000 0000

sign : positive
exponent = 24−127 = −103
mantissa = 0

∴ 1.0X2^−103

3.24 [10] <§3.5> Write down the binary representation of the decimal number 63.25 assuming the IEEE 754 double precision format.

63.25 = 111111.01
= 1.1111101 X 2^5

sign : positive
exponent = 5 + 1023 = 1028

∴ 0 100 0000 0100 1111 1010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000