CH.1

* Throughput Also called bandwidth- Another measure of performance, it is the number of tasks completed per unit time
* Assembler -A program that translates a symbolic version of instructions into the binary version.
* assembly language - A symbolic representation of machine instructions.
* machine language - A binary representation of machine instructions.
* Why we can't speed up anymore? we have run into the practical power limit for cooling commodity microprocessors.
* RISC - Reduced instruction set computing
* CISC - Complex instruction set computing
* MIPS (million instructions per second) - A measurement of program execution speed based on the number of millions of instructions. MIPS is computed as the instruction count divided by the product of the execution time and 106.
* CPI (clock cycles per instruction) -the average number of clock cycles each instruction takes to execute
* Performance = 1/execution time
* Execution time = CPU time =Instruction count\* CPI\* Clock cycle time=(Instruction count \*CPI)/clock rate
* CPU clock cycles =Instructions for a program \*Average clock cycles per instruction

CH.2

* Reason for 32-bit registers: may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther. Also, the number of bits it would take in the instruction format
* 32-Bits is a word
* 2's complement go to the first one and invert everything after it, ex 4 = 0100, -4=1100
* There are 2n-1-1 positive bits to 2n-1 negative bits, leading bit is signed bit i.e. 1 = 1 negative 0=0 positive
* Latency - the time it takes to complete an individual instruction
* RISC- Emphasis on software-Single-clock, reduced instruction only -Register to register: "LOAD" and "STORE" are independent instructions -Low cycles per second,large code sizes -Spends more transistors on memory registers - use less power have better perfromance
* CISC        -Emphasis on hardware - Includes multi-clock complex instructions - Memory-to-memory:"LOAD" and "STORE" incorporated in instructions - Small code sizes, high cycles per second - Transistors used for storing complex instructions
* RISC disadvantage takes more lines to do something CISC can do in one but can do in same amount of time

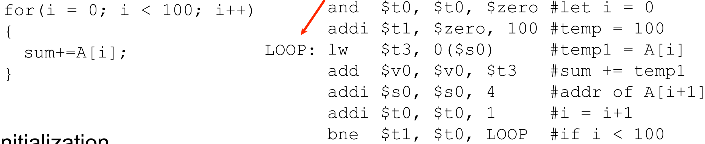
CH.3

* Overflow - means that the exponent is too large to be represented in the exponent field.
* Underflow - occurs when the negative exponent is too large to fit in the exponent field.
* Floating point single precision (-1)sign(1+Frac)\*2(exp-127)--> double 2(exp-1023)
* Ex: 63.25 into floating point 🡪 63 base 2 = 11 1111 🡪.25 base 2 = .01 🡪63.25 = 11 1111.01 = 1.1111101\*25 🡪positive sign = 0 frac = .111110100..0 exp = 5 🡪(-1)0(1.1111101)\*2(1028-1023), 1028 base 2 = 10000000100 floating number=1000000010011111010000000000000
* 25 base 8 to binary🡪2 =010, 5 = 101🡪010101 = 25 base 8, to go backwards group numbers in 3’s
* 25 base 16 to binary 🡪 2 = 0010, 5 =0101 🡪 00100110 base 2

C Conditional Operator MIPS Assembly Instruction

a == b beq $t0, $t1, then

a != b bne $t0, $t1, then

****a < b blt $t0, $t1, then

a > b bgt $t0, $t1, then

a <= b ble $t0, $t1, then

a >= b bge $t0, $t1, then

a == 0 beqz $t0, then

**for loop c code**

int i; for( i=0;i< 10 ) { loop body i++; }

**for loop mips**

li $t0, 10 # t0 is a constant 10

li $t1, 0 # t1 is our counter (i)

loop:

beq $t1, $t0, end # if t1 == 10 we are done

loop body

addi $t1, $t1, 1 # add 1 to t1

j loop # jump back to the top

end:

**c if-else:** if ( i == j ) i++ ; else j-- ; j += i ;

**mips if-else:** bne $s1, $s2, ELSE # branch if !( i == j )

addi $s1, $s1, 1 # i++

j NEXT # jump over else

ELSE: addi $s2, $s2, -1 # else j--

NEXT: add $s2, $s2, $s1 # j += i

**C switch**: switch( i ) { case 1: i++ ; case 2: i += 2 ; break; case 3: i += 3 ; }

**Mips switch:** addi $s4, $zero, 1 # case 1: set temp to 1

bne $s1, $s4, C2\_COND # false: branch to case 2 cond

j C1\_BODY # true: branch to case 1 body

C2\_COND: addi $s4, $zero, 2 # case 2: set temp to 2

bne $s1, $s4, C3\_COND # false: branch to case 3 cond

j C2\_BODY # true: branch to case 2 body

C3\_COND: addi $s4, $zero, 3 # case 3: set temp to 3

bne $s1, $s4, EXIT # false: branch to exit

j C3\_BODY # true: branch to case 3 body

C1\_BODY: addi $s1, $s1, 1 # case 1 body: i++

C2\_BODY: addi $s1, $s1, 2 # case 2 body: i += 2

j EXIT # break

C3\_BODY: addi $s1, $s1, 3 # case 3 body: i += 3

EXIT:

**C while:** while ( i < j ){ k++ ; i = i \* 2 ;}

**Mips while:** L1: bge $s1, $s2, DONE# branch if ! ( i < j )

addi $s3, $s3, 1 # k++

add $s1, $s1, $s1 # i = i \* 2

j L1 # jump back to top of loop

DONE:

**FETCH:** TheFETCH phase obtains the next instruction from memory and loads it into the instruction register (IR) of the control unit. 3 clock/machine cycles

**State 1)** PC 🡪MAR and then PC + 1 🡪PC. The MAR is loaded with the contents of the PC and the PC is incremented.

**State 2)** M [MAR] 🡪MDR. The MDR is loaded with the instruction which is read from memory

**State 3))** MDR 🡪IR. The data is transferred from the MDR to the instruction register (IR).

**DECODE:** The DECODE phase examines the instruction in order to figure out what the micro-architecture is being asked to do. 1 clock/machine cycle.

**State 4)** opcode (ADD, LDR, JMP). Using the external input (IR), and in particular the opcode bits of the instruction, the finite state machine can go to the appropriate next state for processing instructions.

**EVALUATE ADDRESS:** This phase computes the address of the memory location that is needed to process the instruction.

**FETCH OPERANDS:** This phase obtains the source operands needed to process the instruction.

**EXECUTE:** This phase carries out the execution of the instruction.

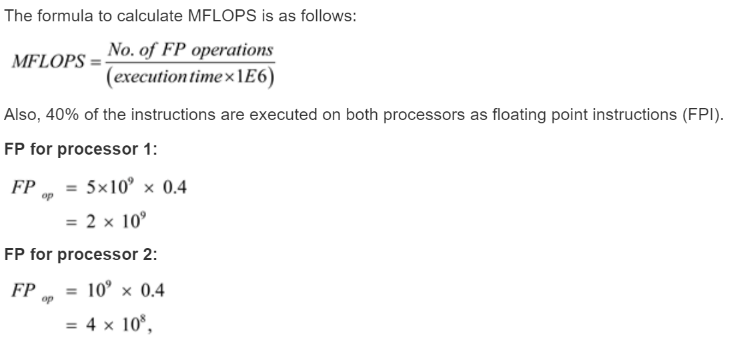
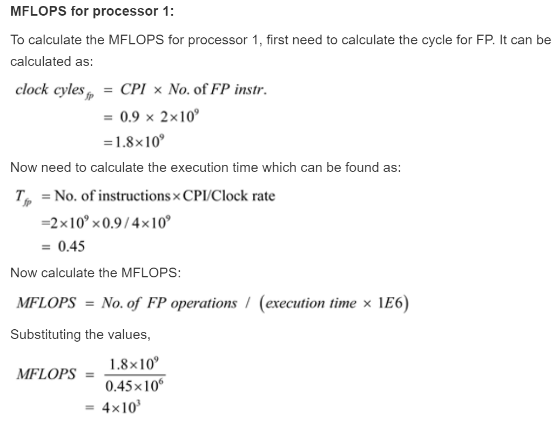
**STORE RESULT:** The final phase of an instruction’s execution. The result is written to its designated destination.

ISA (Instruction Set Architecture)

* Design issues in MIPS (or similar)

MIPS(Constraint is number of bits in architecture)

Risc and 32-bit machine (MIPS)

 Risc = one word instructions which is easier to pipeline because it can go in one cycle. Makes for better throughput

**Floating Point Addition (0.5 + (-0.4375))**

0.5 → 0.1 → 1.000 x 2-1

-0.4375 = -0.0111 → -1.110 x 2-2

1.) Shift the decimal of the one with the lesser exponent until it matches the one with the larger exponent: -1.110 x 2-2 → -0.111 x 2-1

2.) Now add the significands: 1.000 x 2-1 + (-0.111 x 2-1) = 0.001 x 2-1

3.) Normalize the sum checking for underflow or overflow: 0.001 x 2-1 = 0.010 x 2-2 = 0.100 x 2-3 = 1.000 x 2-4

Since 127 >= + 4 >= -126, there is no overflow or underflow. The biased exponent would be -4 + 127 = 123.

4.) Round the sum: 1.000 x 2-4 = 0.0001 = 1 / 24 =  0.0625 → 0.5 – 0.4375 = 0.625

**Floating Point Multiplication**(0.5 x -0.4375)

0.5 → 0.1 → 1.000 x 2-1

-0.4375 = -0.0111 → -1.110 x 2-2 → Keep multiplying only the remaining decimal by 2 and pull the 0 or 1 from the whole

1.) Add the exponents: -1 + (-2) = -3 → biased representation -3 + 127 = 124

2.) Multiply the significands:         1.000

                                                     x   1.110

                                                           0000

                                                        10000

                                                      100000

                                              +    1000000

                                                    1110000  → 1.110000 x 2-3 but we need to keep the 4 bits so its 1.110 x 2-3

3.) Now we check the product to make sure its normalized, and then check the exponent for overflow and underflow. 127 > -3 > -126, there is no over/underflow.

4.) Round the product appropriately. In this case 1.110 x 2-3 is appropriate

5.) Since the signs of the original operands had one negative its -1.110 x 2-3.

6.) Convert to decimal -1.110 x 2-3 = -0.001110 = -0.00111 = -7/25 = -7/32 = -0.21875 → 0.5 x -0.4375 = -0.21875

**Possible type of ISA question: You are working with a 23-bit machine, how many registers should it have?** How many registers you have is determined by how many bits you set aside to determine the register. For example, if you have 3 bits set aside for a register this would allow for 8 registers (23). This is usually the case for 16-bit ISAs. Generally an instruction will be one word minimum (4 bits). The amount of instructions an ISA has also affects the amount of bits it’s going to use. If you have 32 different instructions in an ISA this would need a minimum of 5 bits set aside.

|  |  |  |  |
| --- | --- | --- | --- |
| **Processor** | **Clock Rate** | **Average CPI** | **Instructions** |
| P1 | 4 GHz | 0.9 | 5.0 x 109 |
| P2 | 3 GHz | 0.75 | 1.0 x 109 |

**CPU time(P1) =** (Instruction count)(CPI)Clock Rate **=** (5.0 x 109)(0.9)4 x 109 **= 1.125 seconds**

**CPU time(P2) =** (Instruction count)(CPI)Clock Rate **=** (1.0 x 109)(0.75)3 x 109 **= 0.25 seconds**

**Processor P2 performs faster than processor P1. This shows that a larger clock rate does not mean a larger performance.**

Execution time(P1) = (Instructions)(CPI)Clock Rate= (1.0 x 109)(0.9)4 x 109 = 0.225 seconds

Execution time(P2) = (Instructions)(CPI)Clock Rate= (I)(0.75)4 x 109

0.225 seconds = (I)(0.75)3 x 109

I= (3 x 109)(0.225)0.75=9 x 108 instructions

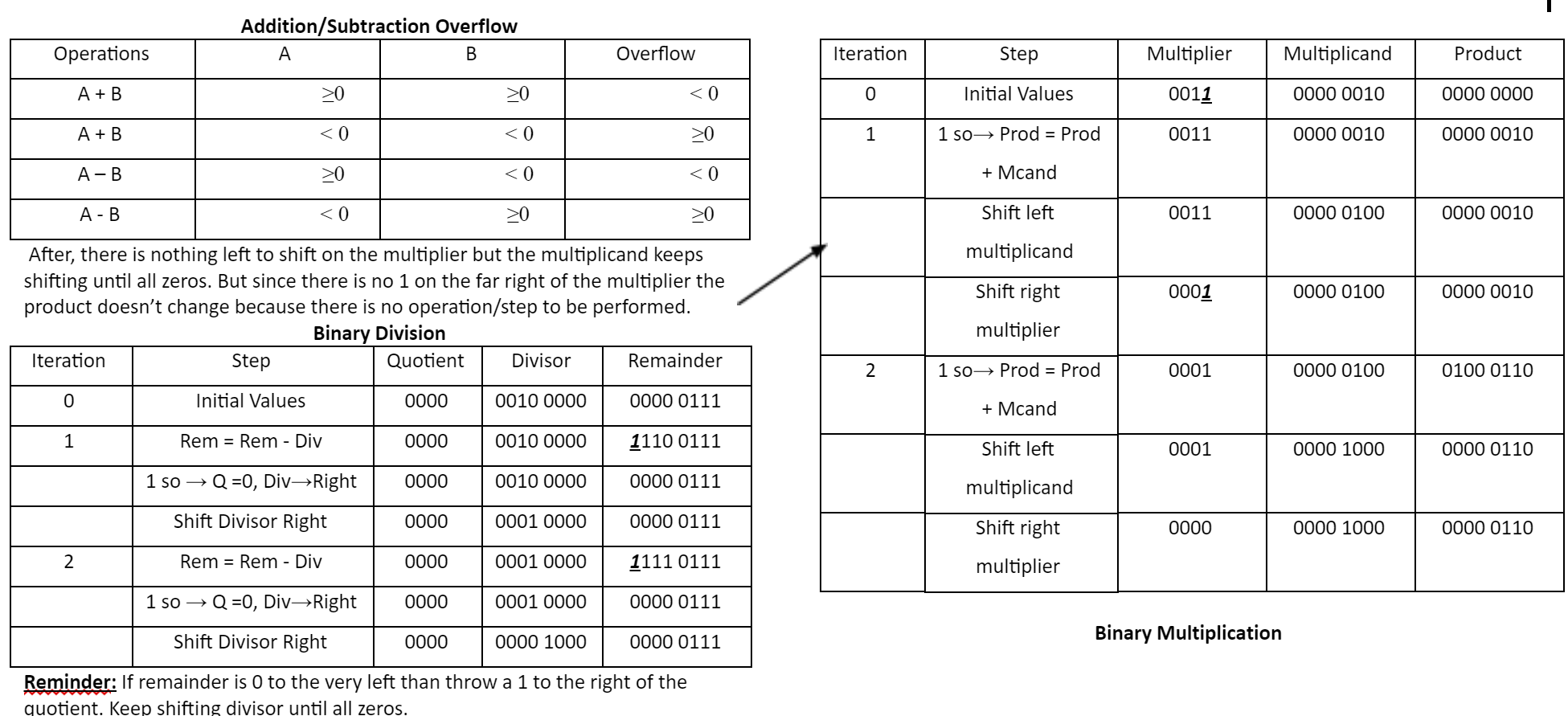
**Execution time(P1) =** (Instructions)(CPI)Clock Rate

Instructions= 4 x 1090.9=4.44 x 109 **instructions**

**Execution time(P2) =** (Instructions)(CPI)Clock Rate

Instructions= 3 x 1090.75= 4.00 x 109 **instructions**

**As noted earlier, processor P2 performs faster than processer P1. This just shows that even if a processor performs faster, it doesn’t necessarily mean it will perform more instructions.**



**Multiplication Visualized (22 x 7)**

22 = 0001 0110 and 7 = 0000 0111

              10110

       x     00111

             10110

           101100

         1011000

       00000000

 + 000000000

  0010011010

**Division Visualized**

110101101=1 →    110 - 101 = 001 → 0 – 1 cannot work so you borrow 10 from the next row which gives 1. The next row is now 0-0 = 0. The next row is 1-1 = 0. Now drop the next 1.

0011 < 101 = 0 → drop the next zero.

00110 > 101 = 1 → 110 101 = 001

001 < 101 = 0 → drop the final 1.

0011 < 101 = 0 → Final answer is 1010 with a remainder of 0011

**Single Precision Floating Point ((-1)sign x fraction x 2exponent)** MIPS can range from 2.0 x 10-38 to 2.0 x 1038 for floating point → (exponent - bias) where bias= 127

|  |  |  |
| --- | --- | --- |
| s | exponent | fraction |
| 31 | 30-23 (8-bits) | 22-0 (23-bits) |

Underflow occurs when the negative exponent is too large to fit in the exponent field.

Overflow occurs when the positive exponent is too large to be represented in the exponent field.

To reduce underflow or overflow we offer double precision floating points (double in C arithmetic), Single precision is the above table.

**Double Precision Floating Point((-1)sign x (1 + Fraction) x 2exponent)** MIPS can range from 2.0 x 10-308 to 2.0x 10308→ (exponent - bias) where bias = 1023

|  |  |  |
| --- | --- | --- |
| s | exponent | fraction |
| 31 | 30-19 (11-bits) | 18-0 (20-bits) |

**Binary to Decimal Floating Point**

|  |  |  |
| --- | --- | --- |
| s | exponent | fraction |
| 1 | 1000 0001 | 010 0000 0000 0000 0000 0000 |

The sign bit is 1, the exponent field is 129, and the fraction field is 1 x 2-2 = ¼, or 0.25.

(-1)sign x 1 + fraction x 2exponent – bias = (-1)1 x (1 + 0.25) x 2129 – 127 = -1 x 1.25 x 22 = -1.25 x 4 = -5.0