CSCI 260 Notes

Ralph Vente

Contents

2019 01 30

Every Instruction Set Architecture (ISA) has cross-compatability with processors which implement that ISA.

Measuring Performance

Performance Issues

Latency is the pause between instruction and execution

Throughput is the rate of instruction completion (work per unity time)

- measured in MIPS (**M**illions of **I**nstructions **P**er **S**econd)
- FLOPS (**F**loating **P**oint **O**perations **P**er **S**econd)

NOTE: Memorize all metric prefixes

Bandwidth like throughput, but measured in the context of networks

- bps - bits per second
- Bps - bytes per second (8 bits)
- word - 4 bytes

Response time like latency, but for larger amounts of work

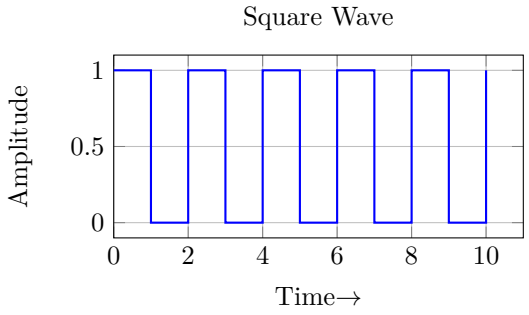
- latency is for a single instruction
- response time for the entire program

Bottleneck something is said to be the bottleneck when it is the limiting factor in execution

Performance Metrics

1. Clock Rate

In a modern computer, there is a clock, which is basically just a square wave. Processing only happens on the rising edge of the square wave.



Peak to peak (or trough to trough) is equivalent to a clock cycle.

- 1Hz = 1 cycle per second
- 2GHz = $2 \cdot 10^9$ cycles per second *or* $1/(2 \cdot 10^9)$ seconds per cycle which is equal to. .5 nanoseconds

This is a bad performance metric because there is a *variable* number of operations per clock across processors.

Complementary Metal Oxide Semiconductor (CMOS)

2. MIPS

Each ISA has different instructions, so MIPS can't be used to compare processors in different ISAs

3. Benchmarks

e.g. SPECMARKS

These benchmarks are the geometric mean of performance across geometric mean of "typical" programs

$$\text{geometric mean} = \left(\prod_{i=1}^{n} \frac{\text{time}_i}{(\text{reference time})_i} \right)^{1/n}$$

Comparing Performance

Unit % improvement

i.e. if $\frac{\text{time}_B}{\text{time}_A} = n$ then A is n times faster than B or $(n-1) \times 100$ percent faster.

Lessons in Evaluating Performance

- Additive v. multiplicative comparison
- Get the units right using dimensional analysis.
- Weighted Averages

instruction type	A	B	C
cycles per instruct.	2	3	4
percentage of time per instruct type	50	20	30

At 36 Hz, ...

For each average, multiply the CPI by the percentage and then sum these components to find the weighted average CPI. Then convert to MIPS using dimensional analysis

2019 02 04

Ex (not MIPS)

x = a + b - c;

```
"add" t, a, b   #t <- a+b
"sub" x, t, c   #t <- (a+b) - c
```

1st item is always destination

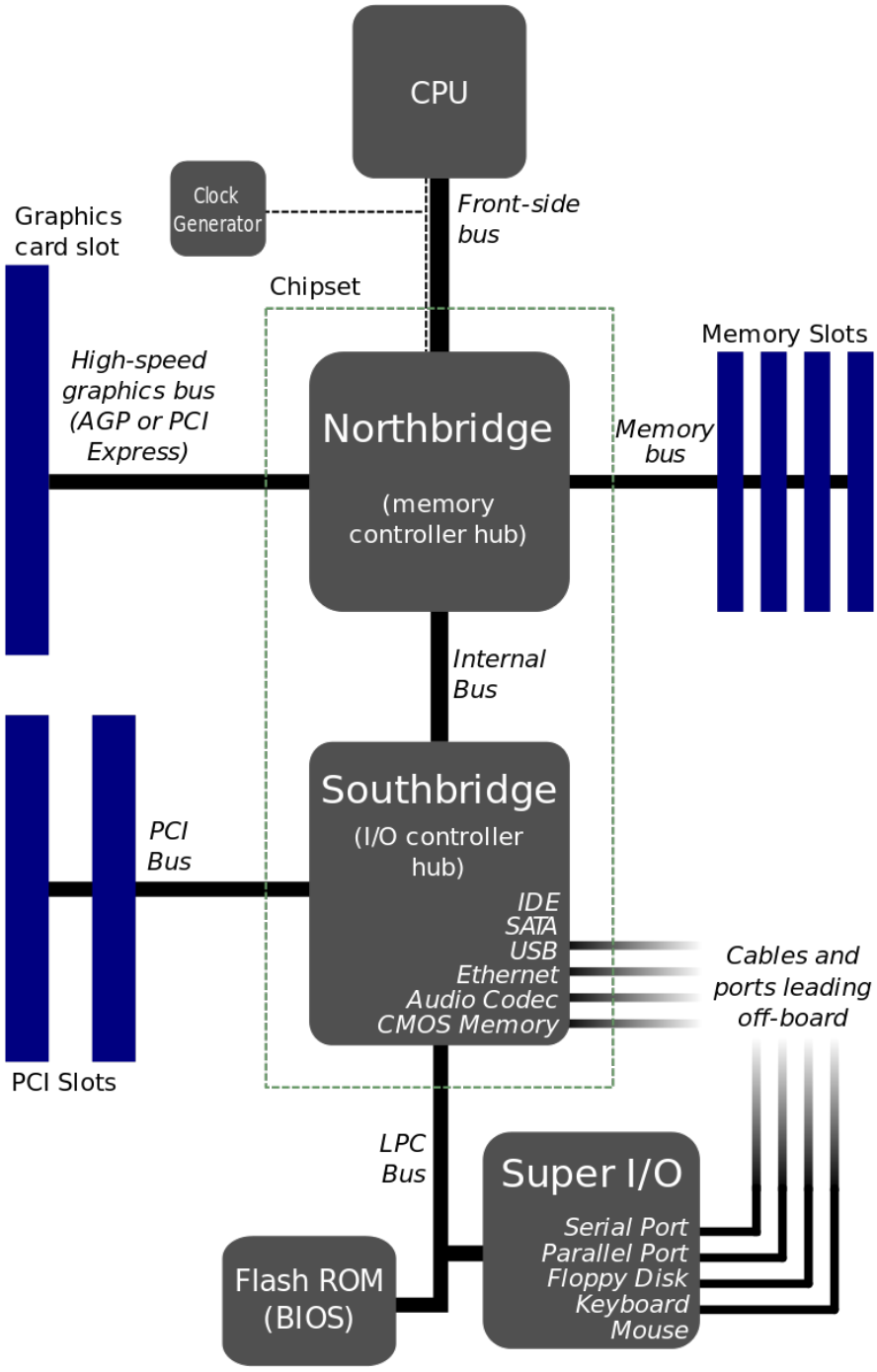


Figure 1: northbridge southbridge diagram

See figure 1 in slack

```
read MEM[a]              # 8 cycles
read MEM[b]              # 8 cycles
add above 2              # 1
store result at MEM[t]   # 8
read MEM[t]              # 8
read MEM[c]              # 8
sub ...                  # 1
store result at MEM[x]   # 8
//TOTAL of 50 Cycles
```

FSB Front Side Bus.

CPU operates at 3.2GHz, FSB operates at 400MHz, meaning one read of memory takes 8

Ways to improve this problem:

1. increase the speed of the FSB
   - the problem with this is that finite distance scale of the FSB limits speed improvements
2. Package things shorter
   - some advantages but FSB also has to attach to other items
3. create a cache of processor local RAM
   - not covered in this course.
   - programmer transparent, does not know whether something is in cache or not. CPU decides what is copied into the cache.
   - registers: these are a much smaller than the cache, for high priority variables. Programmer decided/controlled.

---

https://en.wikipedia.org/wiki/Northbridge_(computing)

FSB bottleneck is obvious in diagram. ~Note the screen resolution bottleneck example is strait from the textbook~

Behold the MIPS

Conventions: Registers on MIPS (32 registers of 32 bits each)

- general purpose registers
- there are conventions for what these 'general' registers

Register name	value stored in register
`$s0, $s1, ..., $s7`	HLL variables
`$t0, $t1, ..., $t9`	Temp. variables
`$zero`	Always stores 'zero'

Register name	value stored in register
(thirteen more...)	discussed later

Naming Conventions	Description
`rd, rs, tt`	registers (any of 32 gen purpose)
`imm`	immediate, ie, 2's complement constants (16-bit)

Instructions (MIPS)		Description
`add`	`rd, rs, rt`	`#rd <- rs + rt`
`sub`	`rd, rs, tt`	`#rd <- rs - rt`
`addi`	`rt, rs, imm`	`#rt <- rs _ imm`

Instructions (MIPS)

```
add   rd, rs, rt      #rd <- rs + rt
sub   rd, rs, tt      #rd <- rs - rt
addi  rt, rs, imm     #rt <- rs _ imm
```

Suppose you are going to write a MIPS program to execute the following C code

```
x=a + b - c + 5;
```

```
#allocate: a-> $30  b ->$s1 c->$s2  x->$s3
add   $t0, $s0, $s1   #$t0 <- a+b
sub   $st1, $t0, $s2  #$t1 <- a+b - c
addi  $s3, $t1, 5     #x <- a+b - c + 5 NOTE: USING 5 literally
```

RAM: Random Access Memory

See figure 2 slack

Memory instructions:

```
lw <destination>,<source>   #load word
sw <source>,<destination>   #store word
```

destination is a register (always) source (imm (register) i.e the address you are storing is a immutable)

Example implement the following C code in MIPS

```
x = arr[i] + y;
```

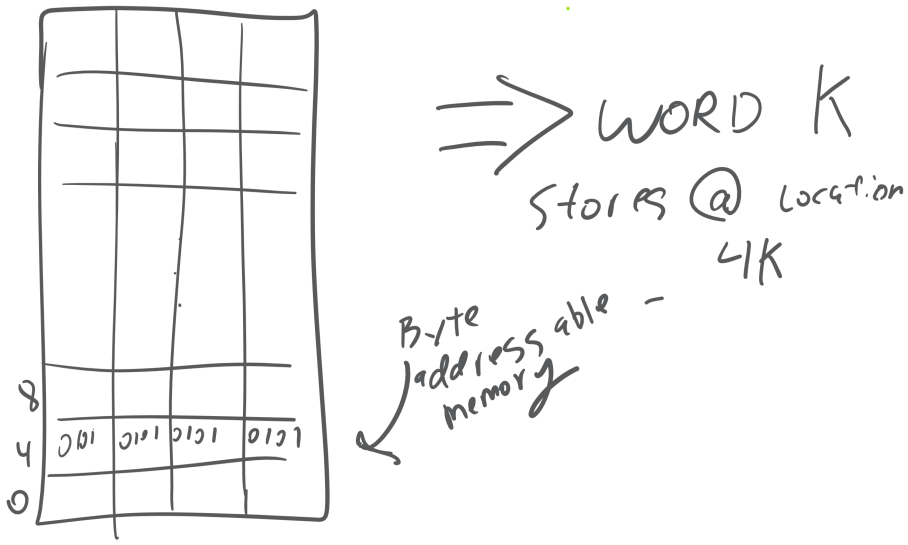


Figure 2: ram illustration

```
#Alloc: i -> $s0, y -> $s1 x -> $s2
#Alloc: arr -> $s3 # ibase address of arr`
add   $t1, $s0, $s0   # t1 <-2i
add   $t1, $t0, $t0   # t1 <-4i
add   $t1, $t1, $s3   # t1 <- addr. of arr[i]
lw    $t0, 0($t1)     # t0 <- arr[i]
add   $s2, $t0, $s1   # x  <- arr[i] +y
```

```
x = arr[i+5] + y;
```

2019 02 06

Example

```
*p = arr[i] + y; // assume p is initialized properly
```

```
# i -> $s0, y -> $s1, p -> $s2, arr -> $s3
add $t3,    # t3 <- 2i
add $t3, $t3, $t3  # t3 <- 4i
add $t3, $t3, $t3  # t3 <- address of arr[i]
lw $t3, 0($t3)     # t3 <- arr[i]
add $t3, $t3, $s1  # t3 <- arr[i] + y
sw $t3, 0($s2)     # *p <- arr[i] + y
```

Logical Instructions

rd, rt, shamt

```
sll # shift left Logical
srl # shift right logical
```

rd, rs, rt

```
# rd <- rs op rt
and
or
nor
xor
```

rt, rs, imm

```
andi
ori
xori
```

`imm` needs to be converted from 16 bits to 32 bits. It uses 0-extended to do this.

Pseudo instructions

```
not rt, rs  # rt <- (rs)'
```

Pseudo instructions are kind of like a "mini-compiler" Don't use yet, may cause problems in the hardware side of things down the line.

Computers have 3 types of shifts: logical, arithmetic, and rotate.

Logical Shift L left <- o's, shift directly left

Arithmetic Shift L. same as logical shift, but does a special case for the sign bit

rotate shift left, moves all bits left moving the leftmost to the right (for the counter-clockwise leftward rotation)

Example: split `$s0` into 4 bytes assume `$s0` is 16 bits `$s1` is Most Significant Byte. `$s4` is Least Significant Byte

```
# bytes stored is $s1, $s2, $s3, $s4


add $t0, $s0, $zero  # t0 <- s0
andi $s4, $t0, 255   # s4 <- LSB of s0, 255 = 0x00FF (line 2)

srl $t0, $t0, 8     # t0 <- s0 >> 8
andi $s3, $t0, 255  # s3 <- 2nd LSB of s0
```

```
srl $t0, $t0, 8      # t0 <- s0 >> 16
andi $s2, $t0, 255   # s2 <- 2nd MSB of s0
srl $t0, $t0, 8      # t0 <- s0 >> 24
andi $s1, $t0, 255   # s1 <- MSB of s0
```

Line 2 explanation: Zero out all but the last 8 bits by `and`ing

```
  1100 1011 0010 1010 0011 0111 1110 1010 = $s0
& 0000 0000 0000 0000 0000 0000 1111 1111 = 0x00FF
-----------------------------------------
  0000 0000 0000 0000 0000 0000 1110 1010 -> $s4
```

0x00FF is operating as a bit mask. It *zero*s out all but the last 8 bits. What if we want to

Relevant identities for bit manipulation

Assume x is a bit.

```
x & 0 = 0
x & 1 = x
x | 1 = 1
x | 0 = x
x xor 0 = x
x xor 1 = ~x
```

$$x \wedge 0 = 0$$
$$x \wedge 1 = x$$
$$x \vee 1 = 1$$
$$x \oplus 0 = x$$
$$x \oplus 1 = \bar{x}$$

Ex: We want to do the following some bit manipulation. We can do this using the logical operations `ori`, `andi`, and `xori`. In this case, the `i` at the end of the command is short for `integer`. We're using the binary representations of the numbers as the underlying mechanism for the result.

1. set bits 3 & 8 of $s5 (correspond to `0...0001 0000 1000`)
2. clear bits 2 & 11 of $s5
3. flip bits 0, 1, 7 of $s5
4. don't care for bits 31..16

```
ori $s5, $s5, 0x0108  # set bits 3, 8 of s5
andi $s5, s5, 0xF7FB  # clear bits 2, 4 of s5
xori $s5, $s5, 0x0083 # flip bits 0, 1, 7 of $s5
```

Use `xor` to flip bits. Or with 1 to set bit to 1.

Install the following packages under atom for convenient editing

- https://atom.io/packages/markdown-preview-plus
- https://atom.io/packages/teletype
- https://atom.io/packages/markdown-writer