CSCI 260 Notes

Ralph Vente Nick Szewczak Anton Goretsky

https://github.com/rvente/CSCI-260-Notes

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

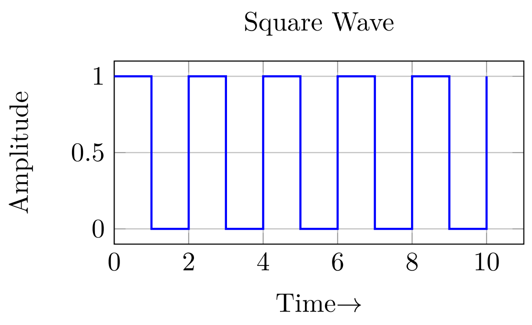
- latency is for a single instruction
- response time for the entire program

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 $\dfrac{\text{time}_B}{\text{time}_A} = n$ 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

(1) Example:

x = a + b - c;

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

First item is always destination in MIPS.

```
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.

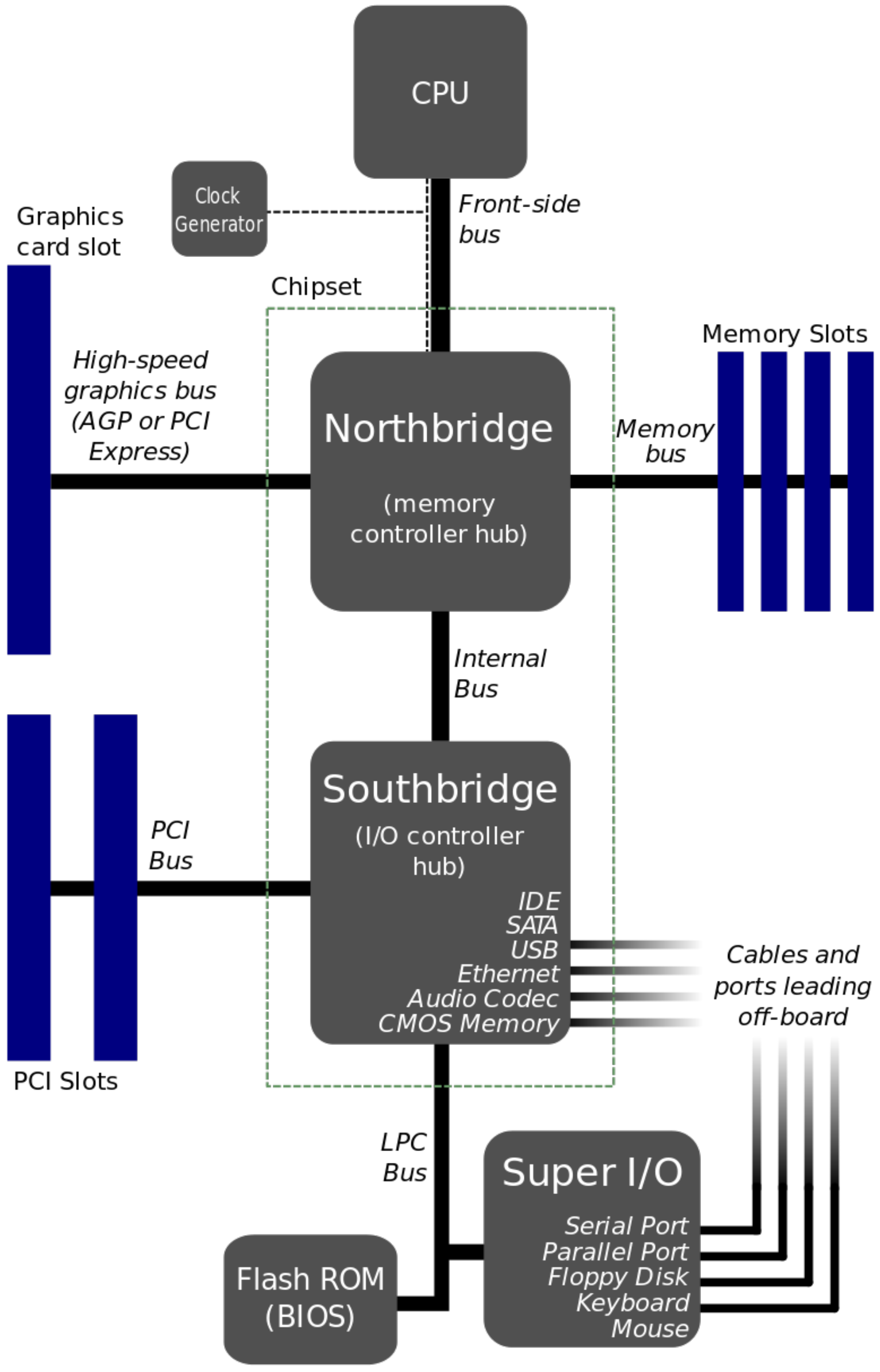


Figure 1: northbridge southbridge diagram

- 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.

Section 2.3

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'
(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

```
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

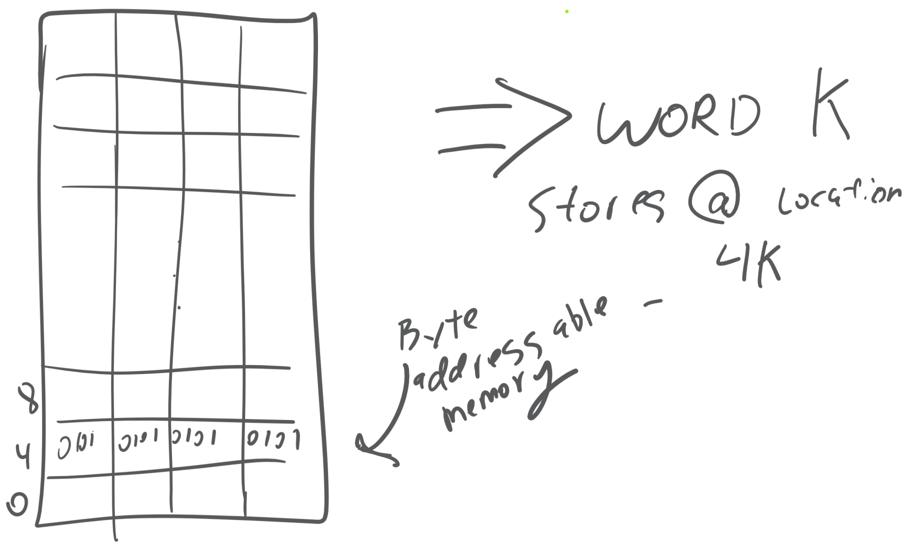


Figure 2: ram illustration

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;
```

```
#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] read data
add   $s2, $t0, $s1    # x  <- arr[i] +y
```

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

2019 02 06

Section 2.3 cont.

Example

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

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

http://people.cs.pitt.edu/~xujie/cs447/AccessingArray.htm

Section 2.6

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.

2019 02 11

Section 2.7

Branching Instructions

```
beq            $t0, $t1, target      # if $t0 == $t1 then target   // branch on equal
bne            $t0, $t1, target      # if $t0 != $t1 then target   // branch on not equal
j              target                # jump to target              // unconditional jump
slt <dest>, <operand1>, <Operand2>
  # if operand 1 < operand 2, dest <- '1'
  # else "                 ", dest <- '0'
```

Control flow Graph (CFG) useful for charting out conditions so that conversion into assembly is trivial

Note: Professor uses '->' for allocation, '<-' for assignment

Example 1

(2) Example: Implement the following `c` code.

```
if (x == y)
  z = x - y;
else
  z = x + y;
z = 2*z;
```

```
# allocate: x -> $s0; y ->$s1; z -> $s2
        beq $s0, $s1, LThen   # if x = y goto LThen
        add $s2, $s0, $s1     # z <- x + y
        j LendiF              # finish iF-then-else
LThen:  sub $s2, $s0, $s1     # z <- x - y
LEndiF: add $s2, $s2, $s2     # z <- 2z
```

`j LendiF` is needed because we must skip over the next line, LThen. If the jump wasn't present, it would (incorrectly) execute the next line of code.

After the LThen line is executed, control flow goes on to the next line, executing the common condition.

Example 2

(3) Execute the following `c` code.

```
//ex:
if x < y
  x = *p;
else
  x = y;
```

Implementation of program (3).

```
# allocate: x -> $s4, y -> $s3, y -> $s2
      slt $t6, $s4, $s3   # t6 <- 1 if x<y, else 0
      bne $t6, 0, Then    # goto Then if x<y
      addi $s4, $sr, 0    # x <-y
      j   End             # Goto end == done!
Then: lw $s4, 0($s2)      # x<-*p
End: ...
```

Loops

Example 3

(4) Execute the following `c` code.

```
for (i+0; i<n; i++) {
  x = 2*x;
}
y = x;
```

Implementation of program (4) in MIPS

```
    # allocate i -> $s0, x -> $s1, n -> $s2, y -> $s3
        addi   $s0, $zero, 0                     # $s0 = $zero + 0 //i <-0
        slt    $t3, $s2, $s0       # t3 <-1 if n < i else  t3 <-0
        bne    $t3, $zero, EndLP   # if $t0 != $zero then target
        add    $s1, $s1, $s1       # $s1 = 2x
        addi   $s0, $s0, 1         # $s0 = $t1 + 1 // i++
        j              LP          # jump to LP //itterate
endLP: add     $s3, $s1, $zero     # $s3 = $t1 + $t2
```

```
#alloc. i -> $s3, j -> $s4, k -> $s5, save -> $s6

Loop: add $t7, $s3, $s3   # t7 <- 2i
      add $t7, $t7, $t7   # t7 <- 4i
      add $t7, $t7, $s6   # t7 <- addr. of save[:]
      lw  $t7, 0(st7)     # t7 <- element value
      bne $t7, $s5, Exit  # Exit loop if save [i] != K
      add $s3, $s3, $s4   # i <- i + j
      j   Loop            # next iteration
Exit:
```

2019 02 13

This section provides diagrams of the binary codes implemented in MIPS. Each diagram is an illustration of what value is stored in memory and how the instructions are stored in memory and they physical layout of their arguments.

R-type instructions

```
op rd, rs, rt

----------------------------------------
|        | rs | rt | rd | shamt | code |
----------------------------------------
```

```
lw rt, offset(rs)
sw rt, offset(rs)
addi rt, rs, imm

----------------------------------------
|        | rs | rt |    offset         |
----------------------------------------
|  6     | 5  | 5  |      16           |
----------------------------------------
```

`beq`: Branch on equal

```
----------------------------------------
| 000000 | rs | rt |    offset         |
----------------------------------------
|  6     | 5  | 5  |      16           |
----------------------------------------
```

What is actually stored is the word offset to the next instruction.

j: Jump

```
----------------------------------------
| 000000 |       target              |
----------------------------------------
|  6     |         26                |
----------------------------------------
```

target = concat(PC[31:28],target * 4)

pg 851 appendix

Note: Don't forget to add 1 in twos complement notation.

2019 02 20

New pseudoinstructions

Problem: we would like a way to store 64 bit integers, but we only have 32 bits to work with.

```
li reg, const # load 32 bit const into register
la reg, label # load 32bit address corresponding to the label into register
```

example of usage:

```
L1: li $t4, 0xABCD1234
    la $t6, L1
```

This allows us to store said integer, bypassing the afforementioned limitation.

2019 02 24

Note: class topics are important because the textbook has incomplete information on this subject.

Functions in MIPS

(5) Example: Implement the following c code in MIPS

```
main() {
  d = foo(2*a, b);
}

int foo(int x, int y) {
  return (x+y);
}
```

```
#allocate a -> $s0, b -> $s1, d -> $s2

      add  $a0, $s0, $s0    #a0 <- 2a (argument 0 register)
      add  $a1, $s1, $zero  #a1 <- b (" ")
      jal  foo              # call foo (see note)
      add  $s2, $v0, $zero  # d <- Return Value (2a,b)
      ...
foo:  add  $v0, $a0, $a1    # v0 <- x+y
      jr   $ra              # return


      # jal sticks result of next function into $ra
      # jr is transferring control to main
```

Note: we should not be writing foo at this point, this is a top down approach.

Issues

1. What if main and foo use same registers?

   Have a caller and &callee follow a convention to save registers somewhere special.

Table 5: MIPS Conventions

register	contents	Saved by
0	`$zero`	
1	`$at` assembler temp	
2, 3	return vale	caller
4,7	`$a0-3`, arg regs	caller
8..15	`$t0-8`, temp regs	caller
16.. 23	`$s0-7`, s-registers	callee
24, 25	`$t3`, `$t9` temp reg	caller
26, 27	`$k0`, `$k1` resvd os	
28	`$gp` global ptr	callee
29	`$sp` stack pointr	implicitly preserved
30	`$fp` frame pointr	callee
31	`$ra` return address	callee

2. What if foo calls bar (or is recursive)?

   Last in First out Stacked function calls

Use a stack to store different values of `$ra`.

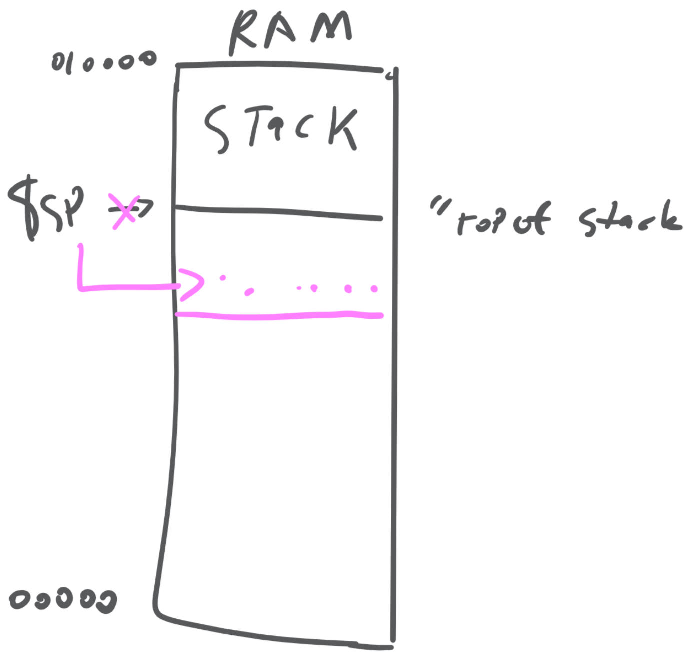


Figure 3: stack visualization

```
# push s1 onto stack:
   addi $sp, $sp, -4 # because stack grows downward
   sw $s1, 0($sp)
# pop s5 off the stack:
   lw $s5, 0($sp)
   add $sp, $sp, 4
```

3. What if > 4 arguments?

   same as 4

4. Where do functions store local vars that don't fit in registers?

   store extra args & local [non-static!!!] that don't fit in regs in an "activation record" (also called a "procedure frame"). place frame on the stack, with $fp pointing to it.

   access local vars as offsets to $fp

Frame

```
| caller -saved regs      |
| Extra args              |
| saved $ra               |   <--- $fp
| saved $fp               |
| other callee saved-regs |
| Local Vars              |
| ------------------      |  < --- $sp
|                         |
```

For more information, observe the procedure in `procedure.pdf` supplied (and copyrighted) by Shankar.