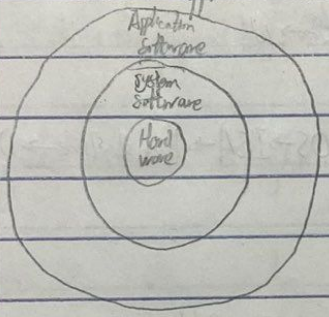
CS2340 DAF2021, Spring 2023

Five main components of a computer system:

CPU, Memory, Storage, Input, Output

Software: Application software and System software



Application software: written by HLL

System software: OS, compiler...

These are called program: Instruction & Data

Abstraction: help deals with complexity, and it hide low level details.

ISA: Instruction set Architecture: The hardware/software interface

Application binary interface: ISA plus system software interface

Implementation: details underlying the interface

Level of program code: HLL → Assembly → Machine code

Inside CPU: Data Path: performs operations on info; Control: control Data path; Cache memory; memory

Moore's law: The number of transistors on an Integrated chip doubles every two years.

All data and programs are represented as binary.

Volatile main memory: normal memory Non-volatile secondary memory: disk, Flash memory, CD/DVD

kilobyte: 2^{10}/1024 → round to 10^3 Megabyte: 2^{20}/1048576 → 10^6 Gigabytes: 2^{30}/1073,741,824 → 10^9

Terabyte: $2^{40} \rightarrow 10^{12}$ Petabyte: $2^{50} \rightarrow 10^{15}$ Exabyte: $2^{60} \rightarrow 10^{18}$

Abstract levels: Problem-orientation language → Assembly → OS → ISA → Microstructure → Digital logic

From High level language:

HLL → Compiler → Assembler → linker → loader → Executable

Linker: merge all the object files and program library together

Loader: Load the program and library to memory

Assembly:

two types: native and pseudo

native instruction: can be understand by machine directly: Ex: add, Rs, Rd, Rt

pseudo instruction: sugar-coated instructions for programers.

Assembler can convert pseudo to native

Assembler can understand instructions, but not cpu

- start with "."
- Executed by assembler at assembly time, not at run-time

Directives for allocating data items: .word, .half, .bytes, .asciiz

Directives for segment information: .data, .text

Symbol related directive: .globl

MIPs syntax: number are base 10, Hex start with "0x"

Lables are followed by ":"

Identifier begin with letter and may contains alphanumeric, underscore and dots.

Key words and instructions cannot be use as identifier.

Comments start with "#"

Assembly language statements cannot be split across multiple lines

[label:] optinal [operand1 [, operand2 [, operand3]]] [#comment]

Syscall: A special instruction request for a OS service.

put requested service at $v0

put input value at $a0 (or $f12 if it is float)

Get out put value from $v0 after called

Example:

```
li $v0, 4  # syscall 4: print string
la $a0, hello # load label hello's string to $a0
syscall    # call
```

Service	code	Arguments	Return
print int	1	$a0 = value	
print float	2	$f12 = value	
print double	3	$f12 = value	
print str	4	$a0 = address of str	
read int	5		$v0 = value read
read float	6		$f12 = value read
read double	7		$f12 = value read
read str	8	$a0 = address of str (label) $a1 = number of chars $a0 = bytes of storage desired	
memory allocation	9		$v0 = address of block
exit	10		

Ex:

```
.text
.global main
main:
la $a0, Hello  # load address -> load value in label Hello to $a0
li $v0, 4      # set request to 4
syscall  # request print string

li $v0, 5  # read a int
syscall  # execute
sw $v0, X  # save word, save $v0 value to label X
lw $t0, X  # load word, load X's value to $t0

sub $t2, $t0, $t1  # subtract, save value of t0-t1 to t2
```

.data

X: .space 4 # allocate 4 bytes of memory for label X

Hello: .asciiz "Hello world" # store string in memory

Performance of compiler program: speed and cost

Standard for performance: response time and throughput

response time: how long does it takes to do a task

how many second does it takes to compile a program

Throughput: total work done in a unit of time

Both are determined by execution time, time to execute a program.

$$\text{Performance} = \frac{1}{\text{execution time}}$$

How to calculate execution time:

elapsed time: total response time, including cpu, I/O, OS, all speeds

cpu time: time cost by cpu to execute a task

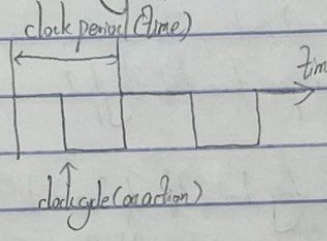
elapsed time determine system performance,

cpu time compromise user cpu time and system cpu time

cpu time is very important parameter of execution time

When we say X is n times higher than Y in performance: $\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{execution time}_Y}{\text{execution time}_X} = n$

CPU clocking: Operation of digital hardware governed by a constant-rate clock



clock cycle: an up and down

clock period: the time required to finish a clock period

ex: 250ps = 0.25ns = 250×10^{-12} sec

clock frequency: the number of clock cycles it can do in 1 second

ex: 4GHz = 4000MHz = 4.0×10^9 Hz (4×10^9 clock cycles per S)

$\text{CPU Time} = \underbrace{\text{CPU clock cycles}}_{\text{(number of clock cycles)}} \times \underbrace{\text{Clock cycle time}}_{\text{(clock period)}} = \frac{\text{CPU clock cycles}}{\text{clock Rate}} = \frac{\text{number of clock cycles}}{\text{Clock frequency}}$

CPU performance can be improve by:

reducing number of clock period

increasing clock rate

Hardware design must trade off between clock rate against cycle count

Example: Computer A; 2GH clock, 10s cpu time

Design Computer B aim for 6s and can do faster clock, but result in 1.2× clock cycles

How fast must computer B clock be

$\text{Clock Rate}_B = \frac{\text{clock cycles of B}}{\text{CPU Time of B}} = \frac{1.2 \times \text{clock cycles of A}}{6s}$

$\text{clock cycles of A} = \text{CPU time}_A \times \text{Clock Rate A} = 10s \times 2GHz = 20 \times 10^9$

$\text{Clock Rate B} = \frac{1.2 \times 20 \times 10^9}{6s} = \frac{24 \times 10^9}{6s} = 4GHz$

clock cycles = Instruction Count × Cycles per instruction

CPI: cycle per instruction: clock cycles needed for each instruction

clock cycles = number of instructions × cycles needed per instruction

$$\text{CPU time} = \text{number of instruction} \times \text{CPI} \times \text{clock cycle time} = \frac{\text{number of instruction} \times \text{CPI}}{\text{clock frequency}}$$

Instruction count for a program: Determined by program, ISA, and compiler.

Average cycles per instruction: Determined by CPU hardware and different instructions have different CPI

CPI example

Computer A: Cycle time = 250 ps, CPI = 2

Computer B: Cycle time = 500 ps, CPI = 1.2

CPU time A = $1 \times 2 \times 250 = 500$

CPU time B = $1 \times 1.2 \times 500 = 600$

$$\frac{\text{CPU time B}}{\text{CPU time A}} = \frac{600}{500} = 1.2 \rightarrow \text{A is 1.2 times faster than B}$$

$$\text{CPI} = \frac{\text{total clock cycles}}{\text{number of instructions}}$$

$$\text{CPU Time} = \frac{\text{number of instructions}}{\text{program}} \times \frac{\text{total clock cycles}}{\text{instruction}} \times \frac{\text{Seconds}}{\text{clock cycle}}$$

Performance depends on:

Algorithm: affect instruction count, possibly average CPI

Programming language: affect instruction count, average CPI

Compiler: Same as programming language

Instruction set architecture: All above, and T_c

Power = Capacitive load × Voltage^2 × Frequency

Pitfall: MIPS as a performance Metric

MIPS: Millions of Instructions Per Second

- Don't account for
  - Different computers' difference in ISA
  - Different instructions' difference in complexity

$$MIPS = \frac{\text{Instruction Count}}{\text{execution time} \times 10^6} = \frac{IC}{\frac{IC \times CPI}{\text{Clock frequency}} \times 10^6} = \frac{\text{clock frequency}}{CPI \times 10^6}$$

Amdahl's law: $T_{improve} = \frac{T_{affected}}{\text{Improvement factor}} + T_{unaffected}$

Ex: 80s/100s, how much improvement in performance must we get 5x overall?

$20 = \frac{80}{n} + 20 \rightarrow$ can't be done

The power is always a limit on performance

Ex: hex to dec (If first number of hex is greater than 7, then it is negative)

a) 0x FADE — negative something

F: 15 → 1111
A: 10 → 1010
D: 13 → 1101
E: 14 → 1110

1111 1010 1101 1110

$2^{15}+2^{13}+2^{12}+2^{11}+2^{10}+2^{8}+2^{6}+2^{5}+2^{4}+2^{3}+2^{2}+2^{1} = 64222$

So this is −64222

− 0000 0000 0000 0001

1111 1010 1101 1101 — flip sign

0000 0101 0010 0010

2^{10} 2^{8} 2^{5} 2^{1}

$1024+256+32+2 = 1314$

b) 0x 4BEF

4: 0100
B: 11 → 1011
E: 14 → 1110
F: 15 → 1111

2^{14} 2^{13} 2^{12} 2^{11} 2^{10} 2^{9} 2^{8}

0100 1011 1110 1111 = 19439

$2^{14}+2^{12}+2^{11}+2^{10}+2^{9}+2^{8}+2^{5}+2^{3}+2^{2}+2^{1}+2^{0}$

$16384+2048+1024+512+256+32+8+4+2+1$

15

Ex: Dec to hex (32 bits)

a) −24312

24312 ÷ 2 = 12156 … 0
12156 ÷ 2 = 6078 … 0
6078 ÷ 2 = 3039 … 0
3039 ÷ 2 = 1519 … 1
1519 ÷ 2 = 759 … 1
759 ÷ 2 = 379 … 1
379 ÷ 2 = 189 … 1
189 ÷ 2 = 94 … 1

94 ÷ 2 = 47 … 0
47 ÷ 2 = 23 … 1
23 ÷ 2 = 11 … 1
11 ÷ 2 = 5 … 1
5 ÷ 2 = 2 … 1
2 ÷ 2 = 1 … 0
1 ÷ 2 = 0 … 1
sign = 0

24312 = 0101 1110 1111 1000

A 1 0 8

−24312 = 1010 0001 0000 0111 + 0001 = 1010 0001 0000 1000

A108 → extend to 32 bits → FFFFA108 (Because it is negative, so pad F)

b) 32544

32544 ÷ 16 = 2034 ... 0 → 0

2034 ÷ 16 = 127 ... 2 → 2

127 ÷ 16 = 7 ... 15 → F

7 ÷ 16 = 0 ... 7 → 7

7F20 → Extend to 32 bits 00007F20 (positive, pad with 0)

Ex: Hex to IEEE binary

a) 0.375

move decimal of 0011 to the left by 1 to get a 1.1, then the right side 2^{-3} times 2 become 2^{-2}

$-0.375 = -\frac{3}{8} = -\frac{3}{2^3} = -1 \times 3 \times 2^{-3} = -1 \times 0011_2 \times 2^{-3} = -1 \times 1.1 \times 2^{-2}$

S: sign = 1 (negative), exponent: $-2 + 127 = 125 = 0111\ 1101$ (power at the end)

Fraction: 100000...00 (save the decimal part of 1.something)

(23 bits)

IEEE = 1 | 0111 1101 | 100000...

(1 | 8 | 23)

Ex: IEEE binary to Hex

a) BEE0 0000

B: 0011

E: 14 → 1110

0011 1110 | 110 0000 0000 0000 0000 0000

positive, S = 0

0111 1101 = 125 → 125 − 127 = −2, Exponent = −2

1100 0000... → 1 + 0.11 = 1.11 → Fraction = 1.11

$1 \times 1.11 \times 2^{-2} = 1 \times 0111 \times 2^{-4} = 1 \times 7 \times 2^{-4} = \frac{7}{16}$

(← Fraction, ← Exponent)

S is 0, so positive

move 2 decimal to right, and pad 0 at front

moved to 2 decimal right, so divide by $2^{-2} \div 2^2 = 2^{-4}$

0111 = 7

RISC vs CISC:

RISC: reduced instruction set computer
CISC: complex instruction set computer

RISC: single word instruction, fixed-field encoding, load/store architecture, simple operation
CISC: Variable length instruction, variable format, memory operands, complex operations.

Why to use RISC: save memory, register, to make chip design simple and regular (universal/common)

A MIPS instruction must be 32 bits long as a word

MIPS Instructions:

- op: operation code (opcode)
- rs: first source register number
- rt: second source register number
- rd: destination register number
- shamt: shift amount
- funct: function code

R-format

opcode (6 bits) + rs (5 bits) + rt (5 bits) + rd (5 bits) + shamt (5 bits) + Funcode (6 bits)

6 + 5 + 5 + 5 + 5 + 6 = 32 bits

I-format

opcode (6 bits) + rs (5 bits) + rt (5 bits) + Immediate (16 bits)

6 + 5 + 5 + 16 = 32 bits

Logical operation: add (add)
Data transportation: lw (load word)
chain control: j (jump)

J-format

opcode (6 bits) + address (26 bits)

6 + 26 = 32 bits

Ex: Sub $t1 $S1 $S2

opcode: 0 9 17 18

funcode: 0x22=34

R: opcode rs rt rd shamt funcode 先化 6

0 17 18 9 0 0x22 → 00|100010|

000000 10001 10010 01001 00000 100010

Arithmetic operations use register operands

I format: Immediate arithmetic and load/Store instructions:

addi $v0, $v1, 4 , add 4 and $v1 and store in v0

Immediate instructions are used to handle constant.

Array operation:

g=h+A[8], g in $S1, h in $S2, base address of A is in $S3

offset, 4X8=32 as 4 bytes per data

lw $t0, 32($S3) ← base address

add $S1, $S2, $t0

Register is faster than memory, try use more registers

Access A[i], Base of A is in S4, b in S1, c in S2, i in S3

add $t1, $t1, $t1 # t1 X2 → i→2i

add $t1, $t1, $t1 # t1 X4 → i→4i

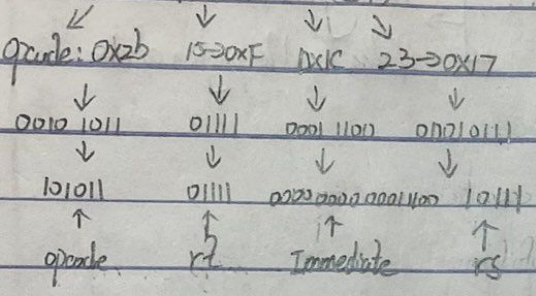
add $t1, $t1, $t4 # add 4i to base address, you get value's address

lw $t0, 0($t1) # load value in address stored in t1 to $t0 with offset 0

I format Example:

Data transportation:

SW $t7, 28($s7)



I	opcode	rs	rt	immediate
	101011	10111	01111	0000 0000 0001 1100

Control flow: Von Neumann computer architure:

Control flow is determined by sequence of executed instructions.

Typically, instructions are executed in sequentially order.

executed in sequence of address: 4byte

Jump and Branch instruction:

The jump and branch instructions change default flow of chain

if (i==j) f=g+h → bne $s3, $s4, Else # if $s3 not equal to $s4, go to Else label

else f=g-h

add $s0, $s1, $s2 # default, when $s3=$s4

j Exit # jump to exit label

Else: sub $s0, $s1 $s2 # Else case

Exit: ...

bne: branch not equal

beq: branch equal

Could use a register and add to it the 16 bits offset

16 bits → extend to 18 bits by ×4] → branching destination address
program counter → program counter + 4] → if true, update program counter

Example:

```
bne $S0, $S1, Lab1
add $S3, $S0, $S1
Lab1: ...
```

Lab1 is one instruction away from it

op: 0x5 rs: 0x10 rt: 0x11 offset: 0x1

000101 10000 10001 0000 0000 0000 0001

Jump: j label

j opCode address
6 bits 26 bits

jump get address: 26 bits address → extend to 28 bits by ×4,
4 bits from program counter + [2+ × address (26 bits)] (28 bits)

While loop: i in $S0, j in $S1, k in $S2

```
while (i!=k)
    i=i+j
```

```
Loop: beq $S0 $S2 Exit
      add $S0 $S0 $S1
      j loop
Exit: ...
```

jal: Jump and link, jump to address and store where are you from in $ra
jr: Jump to address store in register

slt: $t0 $s0 $s1 # if $s0 < $s1, set $t0 = 1 else $t0 = 0
slti: $t0 $s0 constant # if $s0 < constant, set $t0 = 1, else $t0 = 0

slt/slti: for signed, sltu/sltiu: for unsigned

Branch far away:
If target is further than 16 bits, the assembler will re-write the code

```
beq $s0 $s1 L1 #old

bne $s0 $s1 L2 #new
j L1 #j can jump much farther
L2: ...
```

shift left by i bits: multiply by 2^i shift right by i bits: divide by 2^i

There is no not as: A NOR 0 = NOT A Nor: not or, reverse of or → return 1 with two 0

Bytes/half word operation:
lb/lbu lh/lhu load byte, load byte unsigned, load half word, load half word unsigned

How to load 32 bits constant to register

lui rt, constant

ori rt, constant

1010 1010 0101 0101

lui $t0 10101010 # load upper half to register

ori $t0 0101 0101 # load lower part to register

Memory Organization:

main memory can be view as a large one-dimension array of bytes.

a memory address is an index into the memory array

Byte is the smallest address unit but most data use larger space: word

A word in MIPS architecture is a 32 bits or 4 bytes

Endianness: how are data stored in the memory

Memory

little endian			big endian		
	a	0D		a	0A
→	a+1	0C		a+1	0B
	a+2	0B		a+2	0C
	a+3	0A		a+3	0D

storing 0A0B0C0D

Ex: saving Woosh! in little end → sOOW !ooh

Bit: 0 or 1, Nibble: 4 bits, bytes: 8 bits, Word: may be 4, 8, 16, 32, 64 bits, depends on size of register

HLL: char 8 bits, char: 16 bits, int: 8, 16, 32, 64 bits, long: 32 bits.

Hex	bin				
A	1010	C	1100	E	1110
B	1011	D	1101	F	1111