CS 230: Introduction to Computers and Computer Systems

University of Waterloo

Andrew Wang

Contents

Number Represenation and Boolean Algebra

Radix Representation

A n-digit number in base r is written and has the value of

$$d_{n-1}\, d_{n-2} \cdots d_1\, d_0 \quad \rightarrow \quad \sum_{i=1}^{n-1} d_i r^i$$

- We use base-10 (decimal)
- Computers use base-2 (binary)
  - ethernet light/no light, cpu/keyboard high/low voltage, hdd magnet north/south
- Convert from larger to smaller base: Divide by target base until you reach 0 using the remainder at each step as the digit (the remainder of the first division is the *least significant*)
- Convert from smaller to larger base: Multiply each digit with the base to the power of its position

Decimal and Binary Conversion

Example: 19_{10} to binary (Decimal to Binary)

$$\begin{aligned} 19/2 = 8 \quad & R\ 1 \\ 9/2 = 4 \quad & R\ 1 \\ 4/2 = 2 \quad & R\ 0 \\ 2/2 = 1 \quad & R\ 0 \\ 1/2 = 0 \quad & R\ 1 \\ & = 10011_2 \end{aligned}$$

Example: 10011_2 to decimal (Binary to Decimal)

$$\begin{aligned} 1 \times 2^0 &= 1 \\ 1 \times 2^1 &= 2 \\ 0 \times 2^2 &= 0 \\ 0 \times 2^3 &= 0 \\ 1 \times 2^4 &= 16 \\ 1 + 2 + 16 &= 19_{10} \end{aligned}$$

Decimal and Hexadecimal Conversion

Dec	0-9	10	11	12	13	14	15
Hex	0-9	A	B	C	D	E	F

Example: 300_{10} to hexadecimal (Decimal to Hexadecimal)

$$\begin{aligned} 300/16 = 18 \quad & R\ 12 \\ 18/16 = 1 \quad & R\ 2 \\ 1/16 = 0 \quad & R\ 1 \\ & = 12\text{C}_{16} \text{ or } 0\text{x}12\text{C} \end{aligned}$$

Example: 1ED_{16} to decimal (Hexadecimal to Decimal)

$$\begin{aligned} \text{D} \times 16^0 = 13 \times 1 \quad &= 13 \\ \text{E} \times 16^1 = 14 \times 16 \quad &= 224 \\ 1 \times 16^2 = \quad 1 \times 256 &= 256 \\ 13 + 224 + 256 &= 493_{10} \end{aligned}$$

Binary and Hexadecimal Conversion

Bin	0000	0001	0010	0011	0100	0101	0110	0111
Hex	0	1	2	3	4	5	6	7

Bin	1000	1001	1010	1011	1100	1101	1110	1111
Hex	8	9	A	B	C	D	E	F

Example: Binary to Hexadecimal

$$101110110111010100010000_2$$

1011	1011	0111	0101	0001	0000
11	11	7	5	1	0

$$= 0\text{xBB7510}$$

Example: Hexadecimal to Binary

$$0\text{x}3\text{BDF71}$$

3	B	D	F	7	1
0011	1011	1101	1111	0111	0001

$$= 111011110111111011100001_2$$

Boolean Algebra

Boolean Gates

Boolean OR

X	Y	X ∨ Y
0	0	0
0	1	1
1	0	1
1	1	1

Boolean AND

X	Y	X ∧ Y
0	0	0
0	1	0
1	0	0
1	1	1

Boolean NOT

X	¬ X
0	1
1	0

Boolean NOR

$\neg (X \vee Y)$

X	Y	X ↓ Y
0	0	1
0	1	0
1	0	0
1	1	0

Boolean XOR

$(X \wedge \neg Y) \vee (\neg X \wedge Y)$

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

Boolean XNOR

$(X \wedge Y) \vee (\neg X \wedge \neg Y)$

X	Y	X ⊙ Y
0	0	1
0	1	0
1	0	0
1	1	1

Boolean NAND

$\neg (X \wedge Y)$

X	Y	X | Y
0	0	1
0	1	1
1	0	1
1	1	0

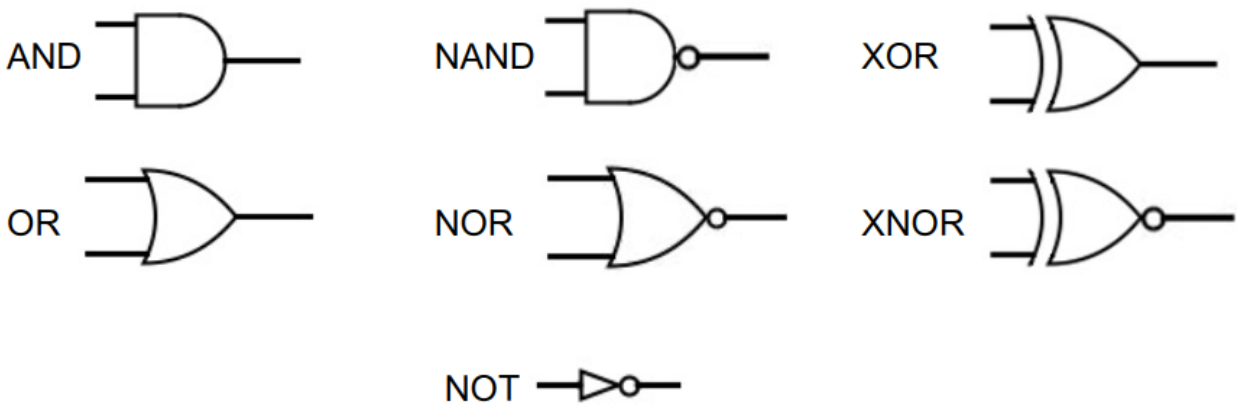
Precedence

Order from highest to lowest

$$\text{NOT} > \text{AND} = \text{NAND} > \text{XOR} = \text{XNOR} > \text{OR} = \text{NOR}$$

Equal precedence are evaluated left-to-right (parenthesis override precedence)

Drawing Boolean Gates



Boolean Algebra Rules

Identities
$A \vee 0 = A$
$A \wedge 1 = A$
$A \vee A = A$
$A \wedge A = A$

Annihilators
$A \vee 1 = 1$
$A \wedge 0 = 0$

Complements
$A \vee \neg A = 1$
$A \wedge \neg A = 0$

Involution
$\neg\neg A = A$

Commutative Law
$A \vee B = B \vee A$
$A \wedge B = B \wedge A$

Associative Law
$A \vee (B \vee C) = (A \vee B) \vee C$
$A \wedge (B \wedge C) = (A \wedge B) \wedge C$

Distributive Law
$A \wedge (B \vee C) = A \wedge B \vee A \wedge C$
$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$
(Recall precedence: $\wedge$ is before $\vee$)

De Morgan's Law
$\neg (A \vee B) = \neg A \wedge \neg B$
$\neg (A \wedge B) = \neg A \vee \neg B$

Binary Arithmetic and Two's Complement

Binary and decimal addition and multiplication both work the same way

Add 1011_2 and 11101_2

$$\begin{array}{cccccc} & & 1 & 0 & 1 & 1 \\ + & 1 & 1 & 0 & 1 & 1 \\ \hline 1 & 0 & 1 & 0 & 0 & 0 \end{array}$$

Multiply 1001_2 and 110_2

$$\begin{array}{cccccc} & & 1 & 0 & 0 & 1 \\ & \times & & 1 & 1 & 0 \\ \hline & & 0 & 0 & 0 & 0 \\ & 1 & 0 & 0 & 1 & \\ + & 1 & 0 & 0 & 1 & \\ \hline 1 & 1 & 0 & 1 & 1 & 0 \end{array}$$

Note that in both examples the size of the result is larger than any of the input

- However if used a fixed size representation then there could be *overflow*

  for a n-bit binary number: Min $= 0$ Max $= 2^n - 1$

- If we want to represent negative values we have a couple of options
  - Signed Magnitude (Bad)
    - sign MSB (most significant bit) with one if neg and zero if pos
    - issue: addition with pos and neg are not trivial and -0 is possible

- One's Complement (Bad)
  - Negative numbers are inverted positive numbers
  - issue: addition only sort of works and -0 is possible
- Two's Complement (Good)
  - For negative number invert and add 1
  - To convert back invert and add 1 (then turn to decimal and add negative sign)
  - range: -2^{n-1} to $2^{n-1}-1$
  - no -0 and both addition and multiplication work

-37_{10} to 8-bit two's complement

- find $+37_{10}$

$$00100101_2$$

- invert then add 1

$$\begin{array}{cccccccc} & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ + & & & & & & & & 1 \\ \hline & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{array}$$

10001111_2 from 8-bit two's complement to decimal

- if MSB is 1 then invert then add 1

$$\begin{array}{cccccccc} & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ + & & & & & & & & 1 \\ \hline & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}$$

- since MSD was 1 it was negative so

$$-113_{10}$$

Floating Point

Binary Fractions

3.625_{10} to unsigned binary

$$\begin{aligned} 3/2 &= 1 \quad R\ 1 \\ 1/2 &= 0 \quad R\ 1 \end{aligned}$$

$$3_{10} = 11_2$$

$$\begin{aligned} 0.625 \times 2 &= 1 + 0.25 \\ 0.25 \times 2 &= 0 + 0.5 \\ 0.5 \times 2 &= 1 + 0 \end{aligned}$$

$$0.625_{10} = 0.101_2$$

$$3.625_{10} = 11.101_2$$

101.0011_2 to decimal

$$1 \times 2^{-4} = 0.0625$$
$$1 \times 2^{-3} = 0.125$$
$$0 \times 2^{-2} = 0$$
$$0 \times 2^{-1} = 0$$
$$1 \times 2^{0} = 1$$
$$0 \times 2^{1} = 0$$
$$1 \times 2^{2} = 4$$

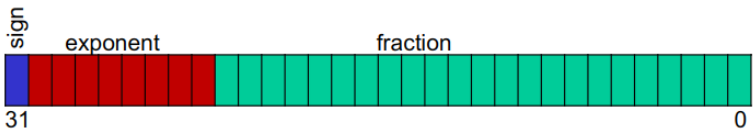
$$101.0011_2 = 0.0625 + 0.125 + 1 + 5 = 5.1865_{10}$$

Floating Point Representation

$$(-1)^{S} \times 1.F \times 2^{E\,-\,B}$$

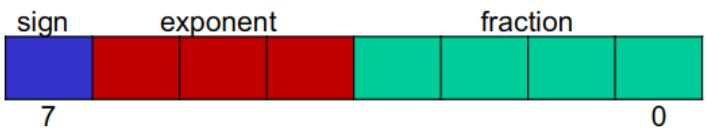
- S - sign bit
- F - binary fraction bits
- E - exponent bits
- B - bias

IEEE-754 Single Precision (32-Bit)



- 31 - sign (1 bit)
- 23-30 - exponent (8 bits)
- 0-22 - fraction (23 bits)
- Bias of 127

Simplified 8-Bit Model



- 7 - sign (1 bit)
- 4-6 - exponent (3 bit)
- 0-3 - fraction (4 bits)
- Bias of 3

Subnormal Case

When our number is really small we switch from 1.F to 0.F

$$\text{Normal } (-1)^{\mathrm{S}} \times 1.\mathrm{F} \times 2^{\mathrm{E}-\mathrm{B}} \qquad \text{Subnormal } (-1)^{\mathrm{S}} \times 0.\mathrm{F} \times 2^{1-\mathrm{B}}$$

To indicate the number is subnormal we use $\mathrm{E} = 0$

Note: When number is subnormal we use the exponent $1-\mathrm{B}$ to not leave a gap between subnormal numbers and numbers that use $\mathrm{E} = 1$

IEEE-754 Special Cases

Exponent	Fraction	Result
$000\cdots$	$000\cdots$	± 0
$000\cdots$	non-zero	subnormal
$111\cdots$	$000\cdots$	$\pm\infty$
$111\cdots$	non-zero	NaN
anything else	anything	normal

Floating Point Arithmetic

- Addition: align radix points and use normal addition
- Multiplication: add exponents and multiply significands ($1.F$ if normal, $0.F$ if subnormal)

Example: Add 00110101 and 01010011 using 8-bit model

$$S = 0 \qquad E = 011 \qquad F = 0101 \quad \rightarrow \quad (-1)^0 \times 1.0101 \times 2^{3-3}$$

$$S = 0 \qquad E = 101 \qquad F = 0011 \quad \rightarrow \quad (-1)^0 \times 1.0011 \times 2^{5-3}$$

$$\begin{array}{rcccccccc} & & & 1 & . & 0 & 1 & 0 & 1 & \times 2^0 \\ + & 1 & 0 & 0 & . & 1 & 1 & & & \times 2^0 \\ \hline & 1 & 1 & 0 & . & 0 & 0 & 0 & 1 & \times 2^0 \end{array}$$

$$1.100001 \times 2^2 \quad \rightarrow \quad (-1)^0 \times 1.100001 \times 2^{5-3}$$

After truncating extra fraction bits

$$S = 0 \quad E = 101 \quad F = 1000 \quad \rightarrow \quad 01011000$$

Example: Multiply 00111010 and 01101100 using 8-bit model

$$(-1)^0 \times 1.1010 \times 2^{3-3} \qquad (-1)^0 \times 1.1100 \times 2^{6-3}$$

$$\begin{array}{rcccccccc} & & 1. & 1 & 0 & 1 & 0 & & \times 2^0 \\ & & 1. & 1 & 1 & 0 & 0 & & \times 2^3 \\ \hline & & 1 & 1 & 0 & 1 & 0 & & \\ & & & 1 & 1 & 0 & 1 & 0 & \\ + & & & & 1 & 1 & 0 & 1 & 0 \\ \hline & 1 & 0. & 1 & 1 & 0 & 1 & 1 & 0 & \times 2^{0+3} \end{array}$$

$$1.011011000 \times 2^4 \quad \rightarrow \quad (-1)^0 \times 1.011011000 \times 2^{7-3} \quad \rightarrow \quad S = 0 \quad E = 111 \quad F = 0110$$

We notice that our E is at infinity so we remove F and say that the result is positive infinity

$$01110000$$

Decimal and Floating Point Conversion

Example: convert 10111100 to decimal using the 8-bit model

$$S = 1 \quad E = 011 \quad F = 1100 \quad B = 3$$

$$\begin{aligned} 1 \times 2^{-1} &= 0.5 \\ 1 \times 2^{-2} &= 0.25 \\ 0 \times 2^{-3} &= 0 \\ 0 \times 2^{-4} &= 0 \end{aligned}$$

$$1.1100 \quad \rightarrow \quad 1.75$$

$$(-1)^S \times 1.F \times 2^{E-B} \quad \rightarrow \quad (-1)^1 \times 1.1100 \times 2^{3-3} \quad \rightarrow \quad -1 \times 1.75 \times 1 \quad \rightarrow \quad -1.75$$

Example: convert -8.75_{10} into our 8-bit model

$$8/2 = 4 \ R \ 0$$
$$4/2 = 2 \ R \ 0 \qquad\qquad 0.75 \times 2 = 1 + 0.5$$
$$2/2 = 1 \ R \ 0 \qquad\qquad 0.5 \times 2 = 1 + 0$$
$$1/2 = 0 \ R \ 1$$
$$0.75_{10} = 0.11_2$$
$$8_{10} = 1000_2$$

$$-1000.11 \quad \rightarrow \quad S = 1, \quad E - B = 3, \quad 1.00011$$

Since $B = 3$ we know that $E = 6_{10} = 110_2$
We are also only able to take 4 digits of the binary fraction so we just truncate the last bit

$$S = 1 \quad E = 110 \quad F = 0001 \quad \rightarrow \quad -9.75_{10} \approx 11100001$$

Example: convert -0.000101_2 into our 8-bit model ($B = 3$)

$$-0.000101 \quad \rightarrow \quad 1.01 \times 2^{-4} \qquad\qquad -4 < 1 - B \quad \rightarrow \quad \text{subnormal}$$

Since the value is subnormal we instead use the exponent $1 - B = -2$ and set $E = 0$

$$-0.000101 \quad \rightarrow \quad 0.0101 \times 2^{-2}$$

$$(-1)^1 \times 0.0101 \times 2^{1-B} \quad \rightarrow \quad S = 1 \quad E = 000 \quad F = 0101 \quad \rightarrow \quad 10000101$$

Blocks of Bits and Endianness

- Byte (8 bits)
  - two's complement range: $-128 \cdots 127$
  - unsigned binary range: $0 \cdots 255$
- Word (32 or 64 bits)
  - Little-endian: least-significant byte first (Intel uses this)
    - can start math right away (don't have to go to end)
  - Big-endian: most-significant byte first
    - the "natural" way of writing a number (leftmost digit is biggest)

- Byte Order
  - Different computers may use different endianness
  - similar challenge for bits but irrelevant since bits are not addressable
  - CS230 will use big-endian
- **Example:** converting big-endian 32-bit word `0x01FAB352` to little-endian
  - Break up into $32/8 = 4$ bytes then reverse the order

    `0x01FAB352` → `0x01 0xFA 0xB3 0x52` → `0x52 0xB3 0xFA 0x01`
  - if we want to send this to another computer then convert to binary

    01010010 10110011 11111010 00000001
- For larger than 32 or 64 bit numbers
  - programming libraries offer big integer types but more costly (more complex data structures)
  - do operations in software rather than hardware

Characters

ASCII

American Standard Code for Information Interchange (ASCII) assigns each character to a number (each character can be stored in 7-bits (`7F` $= 1111111_2 = 127_{10}$) within a byte)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
20		!	“	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Example: `0x0077696E` converts to [NUL]win

Unicode

Unicode provides over 100,000 code points or characters, symbols, etc

- code point range: U+0000 $\cdots$ U+10FFFF or $2^{16} + 2^{20} \approx 1$ million possible code points
- UTF: Unicode Transformation Format
- UTF-32: direct 4-byte encoding of code points
- UTF-16: 2-byte encoding for most code points (sometimes special prefix indicating 4-byte)
- UTF-8: 1 to 4 bytes with first byte compatible with ASCII

For UTF-8 the number of following bytes to take depends on starting bits of the first byte

1st Byte	2nd Byte	3rd Byte	4th Byte	Number of Free Bits	Maximum Expressible Unicode Value
0xxxxxxx				7	007F hex (127)
110xxxxx	10xxxxxx			(5+6)=11	07FF hex (2047)
1110xxxx	10xxxxxx	10xxxxxx		(4+6+6)=16	FFFF hex (65535)
11110xxx	10xxxxxx	10xxxxxx	10xxxxxx	(3+6+6+6)=21	10FFFF hex (1,114,111)

The free bits are gathered up into one long binary number before being converted to hex

The MIPS Assembly Language

Machine vs Assembly Code

Machine Code

- Binary code (0s and 1s) that is directly executed by the processor
- Program is a series of instructions
  - opcode (operation code) + operands (arguments)
  - opcode say what action and operands say what to perform action to

Assembly Code

- Human-readable "programming language" (much simpler than Racket/Python/etc)
- Almost directly mapping to machine code
- Assembler turns it into machine code (process is "assembling" rather than "compiling")

MIPS Assembly Language Introduction

MIPS: Microprocessor without Interlocked Pipeline Stages

- There exists multiple revisions, systems, and compiler (not just one standard MIPS)
- We use simplified version in CS230
- Each instruction is 32 bits = 4 bytes = 1 word
- Arithmetic instructions operate on registers
  - 32 registers numbered $0 to $31
  - register $0 always equals 0
- Instructions have up to 3 operands
  - 1st is destination, 2nd and 3rd are sources
  - source and destination can be the same

MIPS Emulation in CS230

To assemble the program we do

`/u/cs230/pub/binasm < in.asm > out.mips`

There are several emulators in `/u/cs230/pub/` (also called simulators or frontends)

- `noargs <mips-file>`
  run code without user input
- `twoints <mips-file>`
  enter two integers to be stored in `$1` and `$2`
- `array <mips-file>` and `dumparray <mips-file>`
  enter an array of integer numbers to be stored in `$1` and length in `$2`

Arithmatic Operations

Immediate Addition

```
addi $t, $s, i
```

- set `$t` to sum of `$s` and value `i`
- value `i` can be in negative, decimal, hex (Ex: -4 or `0xAA`)
- often used to initialize registers: `addi, $t, $0, i`

Example: `addi $1, $2, 14` (`$1` becomes value in `$2` + 14)

Addition and Subtraction

`add $d, $s, $t`

make `$d` equal to the sum of `$s` and `$t`

`sub $d, $s, $t`

make `$d` equal to `$s` minus `$t`

Multiplication and Division

special registers `hi` and `lo` (accessed in different way than general purpose registers `$0 .. $31`)

`mult $s, $t`

- multiply values in registers `$s` and `$t` and place result in `hi:lo`
  - `lo`: contains the result of the multiplication
  - `hi`: contains the overflow (for CS230 we can ignore)

`div $s, $t`

- divide values in registers `$s` with `$t` and place result in `hi:lo`
  - `lo`: contains integer quotient
  - `hi`: contains the remainder

`mfhi $d`

copy contents of `hi` to `$d` (Move From HI)

`mflo $d`

copy contents of `lo` to `$d` (Move From LO)

Conditional Execution

Program Counter (PC)

- MIPS word size is 32-bits (each instruction is converted to 4 bytes of machine code)
- The Program Counter (CP) is the address of the current instruction's byte
  - always incremented by 4 bytes each instruction
  - `beq` and `bne` does `PC = PC + (i * 4)` when condition is met

Conditional Branch

`beq $s, $t, i`

compare `$s` and `$t` (Branch if Equal) and if equal, skip i instructions (i can be negative)

`bne $s, $t, i`

compare `$s` and `$t` (Branch if Not Equal) and if not equal, skip i instructions (i can be negative)

Rather than setting a number for `i` we can use a label instead

```
    addi $1, $0, 10
loop:
    addi $1, $1, -1
    bne $1, $0, loop
    jr $31
```

In this example we use `loop` instead of -2 (the assembler will turn it into -2 for us)

Comparison

`slt $d, $s, $t`

- compare `$s` and `$t` (Set Less Than)
- when `$s < $t` set `$d=1` otherwise set `$d=0`

Memory and Input/Output

Memory Model

- Byte **addressable**: $2^{32} = 4294967296$ addresses with each address storing 1 byte
- Word **aligned** (word referenced): access word by word (multiples of 4 bytes)

Constants

- Loading a constant can done with `addi` but limited to size of 16 bits
- For larger values use `lis` (Load Immediate and Skip)
  - loads next word (32 bits) into register and skip
  - Example: load value `0xA3257CE2` into `$4`

```
lis $4
.word 0xA3257CE2
```

Memory Access

`lw $t, i($s)`

- Load Word from address `$s+i` into `$t`
- `$s+i` must be word-aligned (`i` is a multiple of 4)

`sw $t, i($s)`

- Store Word from `$t` into the address `s+i`
- `$s+i` must be word-aligned (`i` is a multiple of 4)

Input/Output

We have two special addresses for I/O

- Input Memory Address: `0xFFFF0004`
  - each `lw` reads a new char ("Magic" address)
  - only sent to program once you press "Enter"
  - only works for keyboard input (`<` input redirection may not work)
- Output Memory Address: `0xFFFF000C`
  - bytes written to this address by `sw` will appear on screen
  - ASCII encoding, only the least significant 7 bits

Arrays

Arrays store a fixed length sequence of values called *elements* or *items*

- two components
  - address of the first element
  - number of elements (size of array)
- We use a memory range as the data area (cannot iterate registers, but can iterate addresses)
- direct access and iteration (as location of elements is just a calculation)
- "Raw" array with no built-in boundary checking

Selection Sort

A simple but slow sorting algorithm that sweeps through array finding the smallest element and moving to the end of the sorted part.

An implementation in Python:

```
def SelectionSort(A, length):
    for i in range(0, length-1):
        minpos = i
        minimum = A[i]
        for j in range(i+1, length):
            if A[j] < minimum:
                minpos = j
                minimum = A[j]
        temp = A[i]
        A[i] = A[minpos]
        A[minpos] = temp
```

An implementation in MIPS:

```
; $1 - array start
; $2 - array length
; $8 - array end
; i as $9 and j as $12
; minpos in $10 and minimum in $11
; $12 and $13 for temp

    addi $8, $0, 4      ; set up array end
    mult $2, $8
    mflo $8
    add $8, $8, $1
    addi $9, $1, 0      ; initialize $9
oloop:  addi $10, $9, 0 ; set minpos $10
    lw $11, 0($10)      ; set minimum $11
    addi $12, $9, 0     ; initialize $12
iloop:  lw $13, 0($12)  ; set up compare value
    slt $14, $13, $11
    beq $14, $0, notmin
    addi $11, $13, 0    ; set new minimum
    addi $10, $12, 0    ; set new minpos
notmin: addi $12, $12, 4
    bne $12, $8, iloop  ; loop over $12
    lw $14, 0($9)       ; load front value
    sw $14, 0($10)      ; store at minpos
    sw $11, 0($9)       ; store min in front
    addi $9, $9, 4
    bne $9, $8, oloop   ; loop over $9
```

Subroutines

Calling

`jal x`

- Jump And Link
- copys the current `PC + 4` into `$31` and sets `PC = x` (where `x` is a label)

Indirect Calling

`jalr $q`

- Jump And Link Register
- copy current `PC + 4` into `$31` and set `PC` to contents of `$q`
  - load the address of the label into `$q` with `lis` and `.word x` before calling
  - value in `$q` is called a **function pointer** and allows functions to be passed as parameters

Returning

`jr $s`

- Jump Register
- set `PC` to `$s` (usually do `jr $31`)
  - `$31` holds the **return adddress**
  - `$31` starts with the **outer return address** to exit the entire program

Calling and Returning

```
; example function
addTwoNumbers:
    add $2, $4, $5
    jr $31

; call function
jal addTwoNumbers

; call function indirectly
lis $1
.word addTwoNumbers
jalr $1
```

Example: Print String

- Print NUL-terminated string
- Use register conventions (`$4...$7` for arguments `$8...$15` for local variables)

```
; pr_str: prints out a null-terminated string
; input:  $4
; locals: $8, $9, $10

pr_str: lis $8              ; set $8 for output
        .word 0xffff000c
        addi $9, $4, 0      ; copy value into $9
loop:   lw $10, 0($9)       ; load character
        beq $10, $0, end    ; NUL? -> end
        sw $10, 0($8)       ; output character
        addi $9, $9, 4      ; increment $9
        beq $0, $0, loop    ; loop
end:    addi $10, $0, 0xA   ; print LF
        sw $10, 0($8)
        jr $31              ; return
```

Stack

- last-in first-out queue
- stack grows downward in memory (Ex: `addi $30, $30, -4`)
- convention is to use `$29` for the bottom of the stack however CS230 will use `$30`

Save to stack - *push*

- decrement stack pointer to make room
- copy value to stack point memory location

```
addi $30, $30, -4
sw $x, 0($30)
```

Restore from stack - *pop*

- copy value from stack pointer memory location
- increment stack pointer to free up space

```
lw $x, 0($30)
addi $30, $30, 4
```

Multiple Push/Pop

- Save three registers onto the stack

```
addi $30, $30, -12
sw $3, 0($30)
sw $4, 4($30)
sw $5, 8($30)
```

- Restore the second one and discard the others

```
lw $7, 4($30)
addi $30, $30, 12
```

Remember to always release all stack memory! (restore original value of `$30`)

Argument Passing and Register Conventions

We use registers and the stack to to pass arguments onto subroutines
(first 4 arguments in registers and rest in stack)

Within MIPS the conventions for registers is

`$1`	assembler temporary
`$2`, `$3`	function results
`$4 .. $7`	function arguments
`$8 .. $15`, `$24`, `$25`	temporary
`$16 .. $23`	saved temporary
`$26 .. $27`	OS kernel

- temporary: callee can change in any way (caller saves)
- saved temporary: callee must restore if they want to modify (callee saves)
- **Always clean up the stack before** `jr $31`

Hardware Support

There is often support from hardware or assembler

- reserving the stack area and save registers (also hardware stacks)
- register windows (MIPS does not have)
  - multiple instances (windows) of callee-saved registers
  - fast but limited to fixed small number of windows

Example: Print Integer

- Extract digits
  - check for negative number
  - use modulo 10 and remainder to get digits
  - convert digits to ASCII
- Use stack to reorder digits

```
; pr_int: prints out an integer value
; input:  $4
; locals: $8, $9, $10, $11

pr_int: lis $8              ; set up $8 for output
        .word 0xffff000c
        addi $9, $4, 0      ; set up value $9
        addi $11, $0, 0     ; set up counter $11
        slt $10, $9, $0     ; check for negative
        beq $10, $0, comp
        addi $10, $0, 0x2D  ; print minus sign
        sw $10, 0($8)
        sub $9, $0, $9      ; make $9 positive

comp:   addi $11, $11, 1    ; increment counter
        addi $10, $0, 10
        div $9, $10         ; divide by 10
        mfhi $10
        addi $30, $30, -4   ; remainder on stack
        sw $10, 0($30)
        mflo $9             ; result to $9
        bne $9, $0, comp    ; restart loop

output: lw $10, 0($30)      ; start from stack
        addi $30, $30, 4
        addi $10, $10, 0x30 ; convert to ASCII
        sw $10, 0($8)       ; output
        addi $11, $11, -1   ; decrement counter
        bne $11, $0, output ; restart loop
        addi $10, $0, 0xA   ; print LF
        sw $10, 0($8)
        jr $31              ; return
```

Recursion

- Mathematical programming technique (divide large problem into small ones)
- Needs stack memory area per subroutine

Example: Factorial

```
; $4 for input
; $2 for ouput
; save only $4 during recursion

        addi $2, $0, 1     ; basic result
fac:    slt $3, $0, $4     ; $4 > 0 ?
        beq $3, $0, end    ; else: finish
        addi $30, $30, -8  ; save $31 & $4
        sw $31, 4($30)
        sw $4, 0($30)
        addi $4, $4, -1    ; decrement $4
        jal fac ; recursion
r:      lw $31, 4($30)     ; restore $31 & $4
        lw $4, 0($30)
        addi $30, $30, 8
        mult $2, $4        ; multiply
        mflo $2            ; store result
end:    jr $31             ; return
```

Iteration is still easier

```
        addi $2, $0, 1        ; basic result
        slt $3, $0, $4        ; $1 > 0 ?
        beq $3, $0, end       ; finish
        addi $3, $4, 0        ; use $3 as temp
loop: mult $2, $3             ; product
        mflo $2               ; store result
        addi $3, $3, -1       ; decrement $3
        bne $3, $0, loop      ; loop, if not 0
end:  jr $31                  ; return
```

Example: Fibonacci

$$f(0) = 0 \qquad f(1) = 1 \qquad f(n) = f(n-1) + f(n-2)$$

This can be written in Python as

```
def fib(n):
    if n < 2:  # base case
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

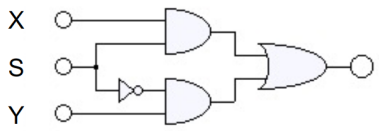
This can be written in MIPS as

```
; input:  $4
; output: $2
; locals: $8

fib:  addi $2, $4, 0    ; default output = input
    addi $8, $0, 1
    slt $8, $8, $4      ; 1 < input
    beq $8, $0, end     ; else: finish

    addi $30, $30, -8   ; save return & input
    sw $31, 4($30)
    sw $4, 0($30)

    addi $4, $4, -1     ; compute f(n-1)
    jal fib             ; result in $2
    addi $30, $30, -4   ; store result on stack
    sw $2, 0($30)
    addi $4, $4, -1     ; compute f(n-2)
    jal fib             ; result in $2
    lw $8, 0($30)       ; restore f(n-1) as $8
    addi $30, $30, 4
    add $2, $2, $8      ; add f(n-1)+f(n-2)

    lw $31, 4($30)      ; restore return & input
    lw $4, 0($30)
    addi $30, $30, 8

end:  jr $31            ; return
```

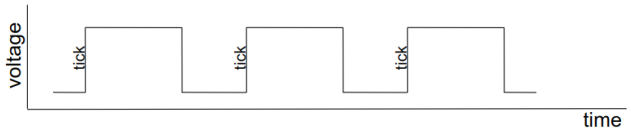
CPU Instruction Proccessing

Definitions

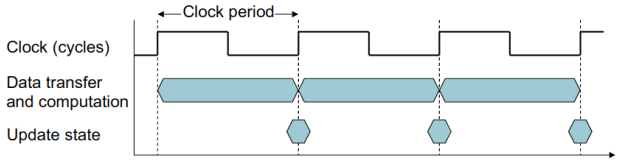
- **Multiplexor**: forwards X or Y signal depending on S (one of the simplest control elements)



- **Clock Cycle**: the heart beat of the computer (measured in time per cycle)



- **Tick**: the rising edge of the clock cycle
  - **Note**: electrial signals are not infinitly fast so there will be gate delays



- **CPU Clocking**: a cycle of working, updating, working, updating, ...
  - we split instructions up into pipeline stages with one stage per clock cycle
- **Clock Frequency**: inverse of clock period and measured in cycles per second or Hertz (Hz)

$$\text{THz} = 10^{12}\,\text{Hz} \qquad \text{GHz} = 10^{9}\,\text{Hz} \qquad \text{MHz} = 10^{6}\,\text{Hz} \qquad \text{KHz} = 10^{3}\,\text{Hz}$$

  - **Note**: 1ns per cycle is 1GHz and 1ps per cycle is 1THz
- **Cycles Per Instruction (CPI)**: determined by the **instruction set architecture** (ISA) which varies between CPUs/programs (divide clock cycles taken by instruction count)
- **Instruction Count**: number of instructions executed by program (determined by program, ISA, and compiler) (every instruction is 5 cycles for MIPS)
- **Elapsed Time**: total time including processing, wait, and idle (perceived performance)

- **CPU Time**: actual time spent processing instructions (user time and system time)

$$\text{CPU Time} = \text{instruction count} \times \frac{\text{clock cycle count}}{\text{instruction count}} \times \frac{1}{\text{clock freq}}$$

$$\text{CPU Time} = \text{instruction count} \times \text{CPI} \times \text{clock period}$$

MIPS Pipeline Stages

- **IF** (Instruction Fetch): retrieve instruction from memory
- **ID** (Instruction Decode): decode instruction and load register values needed
- **EX** (EXecute): execute the instruction using the arithmetic logic unit (ALU)
- **MEM** (MEMory access): modify or read memory
- **WB** (Write Back): write results back to registers

MIPS Pipeline Stages Details

- **IF - Instruction Fetch**
  - load the 32-bit value from the address in memory stored in `PC` and pass it to **ID**
  - increment `PC` by 4
- **ID - Instruction Decode**
  - receive 32-bit binary instruction from **IF**, decode it
  - pass instruction, registers, and immediate values to **EX**
  - if instruction is a branch and the branch condition is met then update `PC`
- **EX - EXecute**
  - receive 32-bit register contents, the instruction, and destination register (the instruction for `lw` and `sw` is the addition of the offset)
  - use the ALU (Arithmetic Logic Unit) to do the math for the instruction
  - pass 32-bit result of the math, destination register, and instruction to **MEM**
- **MEM - MEMory access**
  - receive 32-bit result of the math, destination register, and instruction
  - if instruction is `lw` or `sw` then laod or store memory (otherwise just pass on values)
  - pass 32-bit math result or value loaded (for `lw`), dest register, and instruction to **WB**

- **WB - Write Back**
  - receive 32-bit math result or loaded value, destination register, and instruction
  - if instruction is not `sw` put result or loaded value into destination register

Pipelining

- Analogy: if you have multiple load of laundary you can dry while one load is washing (compared to washing and drying the first load before starting on the second)
- the speedup is increased throughput (latency for each instruction is unchanged)
- max speedup when all stages are balanced (they all take the same amount of time)

$$\text{time between instructions}_{\text{pipelined}} = \text{time between instructions}_{\text{serial}} \ / \ \#\text{ of stages}$$

MIPS instructions are not completely independent so some *hazards* may arise

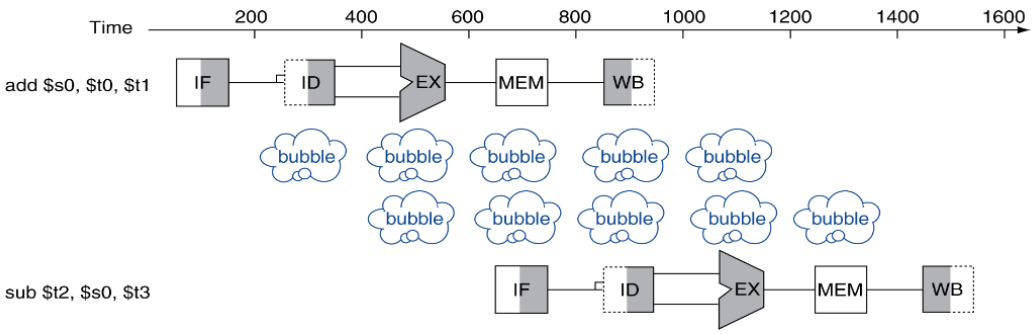
- Structural: when two (or more) instructions require the same resource (Ex: ALU) (instructions/stages must be executed in series rather than parallel)
- Data: when instructions modify data in different stages of the pipeline (if ignored can result in race conditions)
- Control: occurs when the pipeline makes wrong decisions for its branch prediction (brings instructions into pipeline that must be discarded)

We can just assume that MIPS will always avoid structural hazards

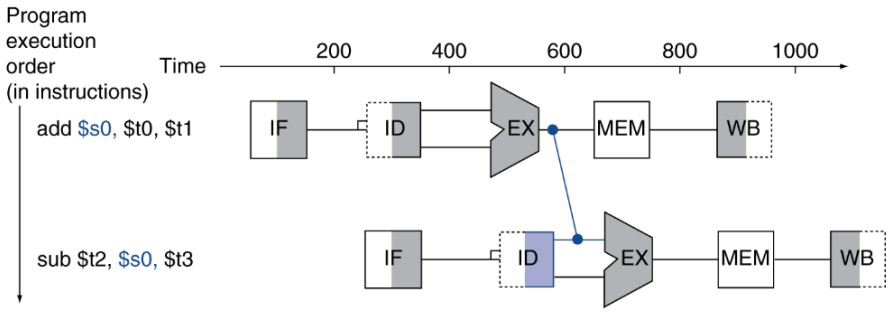
Data Hazards

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

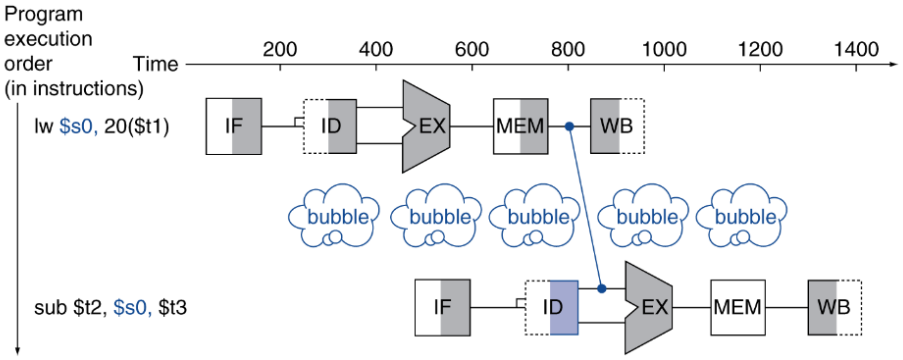
For this example the earliest can run `sub` is when its **ID** lines with **WB**
(this allows the first instruction to write it's results by **WB** before the second reads it in **ID**)



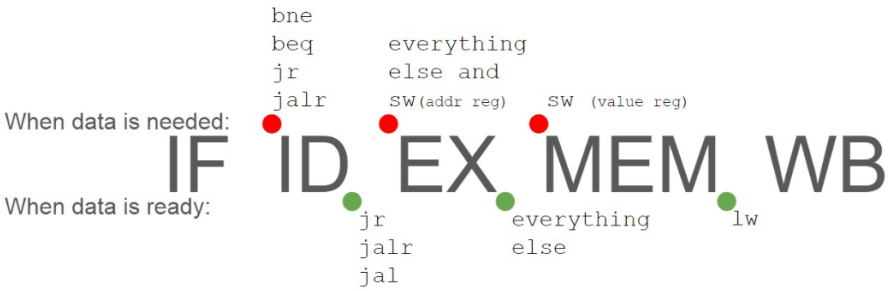
However if we use Forwarding we connect **WB** output to the **WB** input



Note that if the previous instruction is `lw` or `sw` the data is only ready at the **MEM** stage

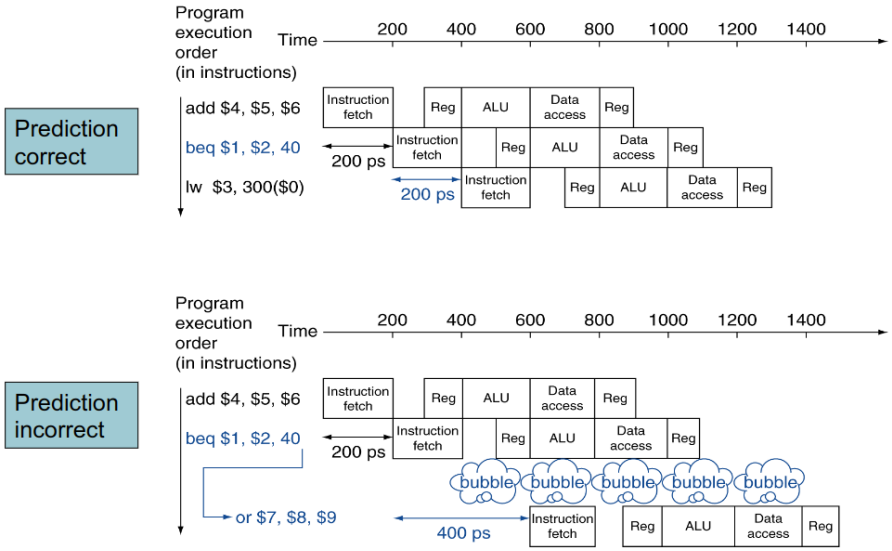


The forwarding connections diagram



Control Hazards

When branching we only know what the next instruction is after the **ID** stage



Thus when the pipeline makes the wrong choice for its branch prediction we stall for one cycle (if we don't make a prediction we would always need to stall for one cycle)

- Static branch prediction:
  - predicts the brackwards branch is always taken (Ex: `beq $x $y -10`)
  - predicts the forwards branch is never taken (Ex: `beq $x $y 10`)
- Dynamic branch prediction:
  - hardware measures branch behaviour and attempts to predict future with history

RISC vs. CISC

Reduced Instruction Set Computer (RISC)

- emphasis on software, single-clock, reduced instruction only
- uniform instruction format with simple addressing modes
- typically larger code sizes with few data types in hardware
- direct execution of machine code, single result write at end

Complex Instruction Set Computer (CISC)

- emphasis on hardward, multi-cycle, complex instructions
- variable length instruction format with complex addressing and memory access
- small code size with complex data types in hardware
- hybrid assembly language: microcode (indirect execution)

Memory and Caching

Memory Hierarchy

Going down the memory hierachy: memory gets cheaper, bigger, slower, and further from the CPU

$$\text{Registers} \rightarrow \text{Cache} \rightarrow \text{Main Memory} \rightarrow \text{Disk} \rightarrow \text{Network Drives} \rightarrow \text{Archive (tape, etc)}$$

- **Registers**: very expensive, very low latency, very high throughput, not persistent
- **Main Memory**: cheap, some latency (100x slower than registers), not persistent
  - address sent to memory controller which responds with value at that address
- **Disk**: very cheap, very large latency (1000x slower than registers), persistent
- **RAM**: internally a 2D matrix but uses index-based access via memory bus
- **Static vs Dynamic RAM**
  - **Static RAM (SRAM)**: used for caches
    - not persistent, multiple transistors per bit, expensive, quite fast
  - **Dynamic RAM (DRAM)**: used for main memory
    - single transistor per bit, cheaper, but slower

In 2008 this was the typical performance and cost of memory

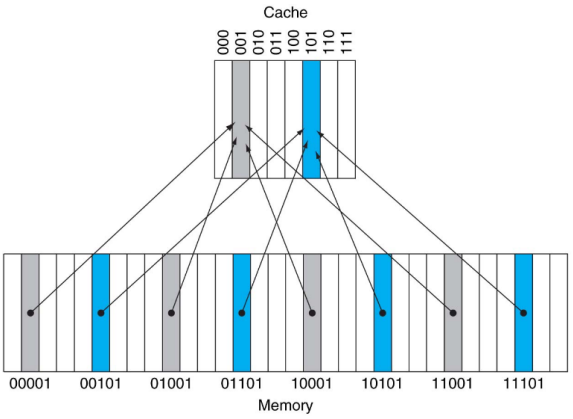
Technology	Access Time	$/GB
SRAM	0.5-2.5ns	$2000-$5000
DRAM	50-70ns	$20-$75
Disk	5,000,000-7,000,000ns	$0.20-$2

- **Locality**: minimizing stalls as slower memory is accessed by keeping a *working set*
  - **Temporal Locality**: data used recently will likely be used again soon (loop)
  - **Spatial Locality**: data close will likely be used soon (iteration through array)

- **cache hit**: data was found in cache
- **cache miss**: data was not found in cache (need to get from main memory)
- **hit time**: time it takes to fetch from cache
- **miss penalty**: time it takes to copy data to cache
  - total time for miss is sum of hit time and miss penalty
- **block**: collection of sequential bytes in memory

Direct-Mapped Cache

- Assume M blocks of cache memory each of size B and a request for address p
- Mapping for cache block is $c = (p/B) \mod M$



- 32 blocks of memory ($p = 0, \ldots, 31$)
- 8 blocks of cache ($M = 8$ and $B = ?$)
- $c = 3$ lowest bits of the block address

Example: say $B = 4$ and $M = 8 = 1000_2$ then 1110111_2 has

- block address: 1110111_2 / $2^2 = 11101_2$
- cache address: $11101_2 \mod 1000_2 = 101_2$

copy all the bytes 11101XX and tag with 11 then place into cache address 101

- **Tag**: used to identify which block of main memory was placed in the cache (appending tag with cache address gives memory block address)
- **Valid Bit**: used to tell if cache block actaully has valid data (1 if data is loaded, 0 if empty)

Block Size Considerations

- Larger blocks should reduce miss rate (due to spatial locality)
- However fewer cache blocks could mean increased miss rates
- larger blocks will have a greater miss penalty

Writing

- Write through: update both cache and main memory (each write operation takes longer)
- Write back: only update cache and mark block as *dirty*
  (update main memory on eviction or in background)
- Write Buffer: dedicated buffer for dirty blocks

Associative Caches

- Fully associative: any block may go to any cache entry
  - requires all entries to be search at once to find correct block (expensive hardware)
- n-way set associative: each set contains n entries
  - the set to place in is determined by: (block num) mod (num of sets)
  - only need to search for entries within a given set (cheaper)

Associative caches use the replacement policy of prefer empty otherwise evict *least recently used* (LRU) (direct mapped does does not have any choices to make)

Associativity Example

Comparing 4-block caches with memory block access sequence: 0, 8, 0, 6, 8

- directly mapped

Block Address	Cache Index	Hit/Miss	Cache Content After Access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

- 2-way set associative (each set holds up to two blocks, the two sets is a coincidence)

Block Address	Cache Index	Hit/Miss	Cache Content After Access			
			Set 0		Set 1	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

- fully associative

Block Address		Hit/Miss	Cache Content After Access			
0		miss	**Mem[0]**			
8		miss	Mem[0]	**Mem[8]**		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	**Mem[6]**	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

In general, increasing associativity decreses miss rate (but has diminishing returns), in addition it gets more expensive (thus the choice depends on level in hierarchy)

Cache Performance

- **CPU Time**: the program execution time including the memory access
- **Average Memory Access Time (AMAT)**: cache hit time + miss rate × miss penalty
- **Multilevel Caches**
  - Primary (level-1) CPU cache: small, but fast (usually instruction speed)
  - Level-2 cache services misses from L1 cache
    - larger, slower, but still faster than main memory
    - check that L1 cache is a misses before looking at L2
  - Main Memory services L2 caches misses
  - Most modern computers usually have L3 and sometimes L4

CPU Time Example

- Instruction cache miss rate = 2%
- Data cache miss rate = 4%
- Miss penalty = 100 cycles
- Base CPI (ideal cache) = 2
- memory accesses are 36% of instructions

Instruction cache miss: $0.02 \times 100 = 2$ Data cache miss: $0.36 \times 0.04 \times 100 = 1.44$

Actual CPI $= 2 + 2 + 1.44 = 5.44$

Average Memory Access Time (AMAT) Example

- 4ns clock
- hit time in cycles = 1 cycle
- miss penalty = 20 cycles to main memory
- cache miss rate = 5%

AMAT = 4 (1 + 0.05×20) = 8ns

Multilevel Cache Exmaple

- CPU base CPI = 1, clock rate = 4GHz or 1/(4GHz) = 0.25ns
- miss rate = 2%
- 50% of program is lw/sw
- Main Memory access time = 100ns

With only the L1 cache (L1 is instruction speed)

- miss penalty = 100ns/0.25ns = 400 cycles
- effective CPI = $1 + 0.5 \times 0.02 \times 400 = 5$

Adding a L2 cache

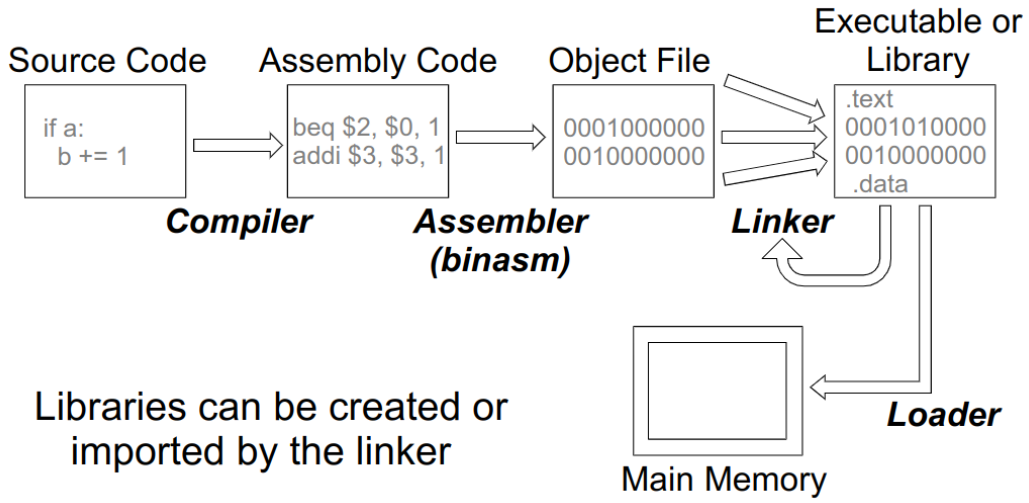
- L2 access time = 5ns

- Global miss rate = 0.5% (chance of missing both L1 and L2 cache)
- Primary miss with L2 hit: penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L2 miss: 100ns main memory access time = 400 cycles
- effective CPI = $1 + 0.5 \times 0.02 \times 20 + 0.5 \times 0.005 \times 400 = 1 + 0.2 + 1 = 2.2$

Build and Execute

Classical Tool Chain

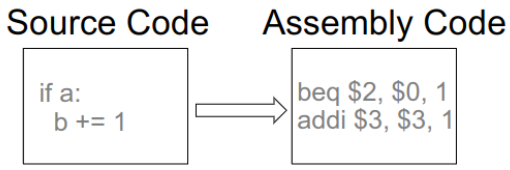
- **Compiler:** translates high level language into assembly program
- **Assembler:** translates assembly program into machine code in object file
- **Linker:** combines multiple object files of machine code into program file
- **Loader:** loads program file into main memory
- **Library:** special object that can added to program file during linking or loading



Other Execution Approaches

- Interpretation
  - exeucte source code directly
  - execute binary code by software
- Byte Code: compile to intermediate binary representation
- Just-In-Time Compilation: compile during runtime

Compiler



- translates the program from the *source* language to *target* language (usually assembly code)
- typically followed by assembler to generate machine code

Basic Compilation Steps

- Scanning: source code to token sequence
- Syntax analysis: token sequence to parse tree
- Semantic analysis: use parse tree to generate symbol table
  (type check the parse tree agsint the symbol table)
- Code generation: parse tree and symbol table to *target* language

Scanner / Tokenizer

- also called "Lexical Analysis"
- convert program text into stream of *tokens*
- some types of tokens
  - keyword: `for`, `while`
  - operator: `+`, `&&`
  - constant: `1000`, `3.5`
  - delimiter: `:`, `;`
  - variable name: `minpos`, `maxtime`
  - subroutine name: `power2`, `print`

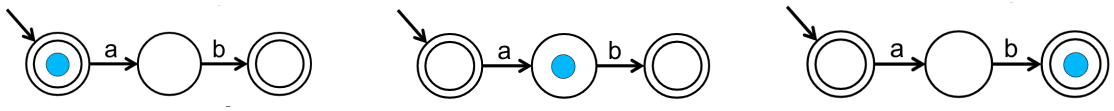
Deterministic Finite Automata (DFA)

Also known as a deterministic finite state machine (FMS)

- finite set of states (exactly one start state, at least one final/accept state)
- finite set of input symbols known as the alphabet (doesn't have to be english alphabet)
- finite set of transitions from one state to another based on input

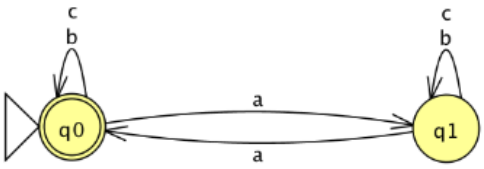
DFA Example

Start state has arrow from nowhere and the final/accept states are double circles



- Consider the input "ab". We begin in the start state "|ab"
- We consume "a" to take the "a" transition to get "a|b"
- We consume "b" to take the "b" transition to get "ab|"
  - Since the marker is at the end of the string we check it we are in an accept state (double circle) and say that this DFA *accepts* the string "ab"
  - string "a" does not land on an accept state so the DFA *rejects* it
  - string "aa" gets stuck on "a|a" so it is also rejected

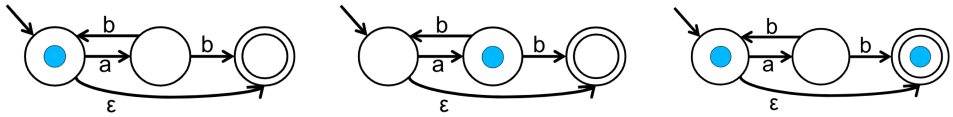
Another Example: DFA over the alphabet $\Sigma = \{a, b, c\}$ that accepts strings with an even number of a and any number of b and c



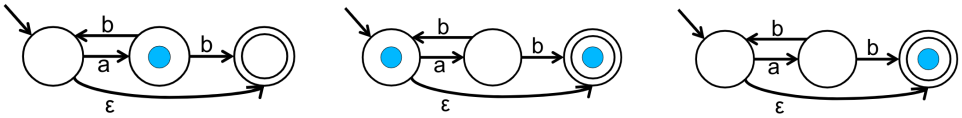
Non-deterministic Finite Automata (NFA)

- NFAs can have more than one transition for some input
  (Ex: $1 \to 2$ and $1 \to 3$ may both consume "a")
  - DFAs can only have one transition per input per state
- NFAs can include an ϵ (empty) transition
  - DFAs cannot change state without consuming input
- NFAs are much easier to design than its equivalent DFA (but harder to evaluate)
- We are always able to create an equivalent DFA from an NFA

NFA Example



- use the input string "|abab"
- after consuming the "a" we have two choices for "a|bab" so we create two clones
- at "ab|ab" we have two clones but notice that the right one has nowhere to go



- kill the right clone and let the left clone to consume "a" to get "aba|b"
- at "abab|" we again have 2 clone
- we say the NFA accepts "abab" since the right clone is in an accept state

Regular Expressions

- The set of all strings accepted by a DFA/NFA is called its *language*
- DFAs/NFAs accept a particular type of language: a *regular language*

Regular expressions (regex) are concise ways to define regular languages

- Σ is the set of all legal characters
- $\varnothing$ is the empty set
- ϵ is the empty string

Basic regex operations

- Alternation

$$R|S = R \cup S$$

- Concatenation

$$RS = \{ab : a \in R \text{ and } b \in S\}$$

- Kleene star

$$R^* = \epsilon \mid R \mid RR \mid RRR \mid \cdots$$

Some basic examples

$a^* = \{\epsilon, a, aa, aaa, \dots\}$ $b|a^* = \{b, \epsilon, a, aa, aaa, \dots\}$ $(h|c)at = \{hat, cat\}$ $hello = \{hello\}$

Other regex operations

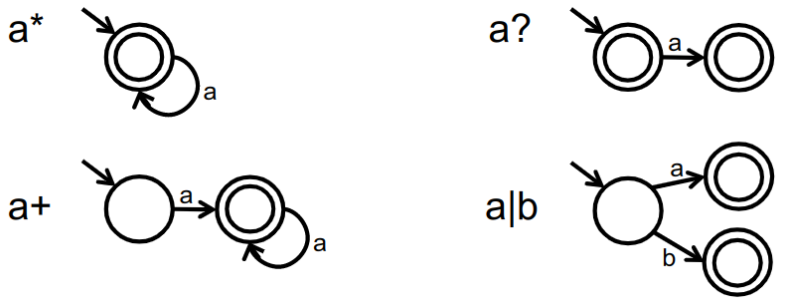
- plus matches one or more: $a+ = \{a, aa, aaa, \dots\}$
- `?` matches one or nothing: $a? = \{\epsilon, a\}$
- square brackets: `[abc]` = `a|b|c` `[a-z]` = any letter from `a` to `z`
- dot matches any single letter: `.at` = `(a|5|j|A|?|...)at`
- the escape character `\` escapes the next character: `\.` = matches a dot

Examples:

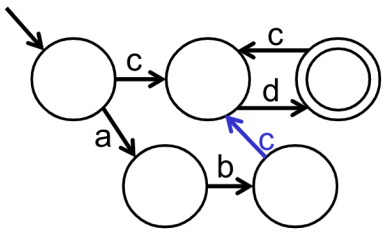
- $ab+ = \{ab, abb, abbb, \dots\}$
- $(h|c)?at = \{hat, cat, at\}$
- matching the different ways to write Händel (Händel, Haendel, Handel, Hendel)

$$[\,\mathrm{Hh}\,]\,(\,\mathrm{ae}\,|\,\mathrm{a}\,|\,\mathrm{e}\,|\,\ddot{\mathrm{a}}\,)\,\mathrm{ndel}$$

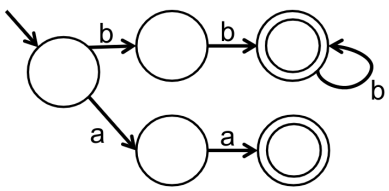
Converting Between Regex and NFAs/DFAs



Regex to DFA: $(ab)?(cd)+$



DFA to Regex: $aa|bb+$ or $aa|bbb^*$



Context-Free Grammer

Definitions

- **terminal/token**: an atomic symbol (number, char, etc)
- **non-terminal/variable**: an abstract component that does not literally appear in the input
  - one of which is designated as the *start symbol*
  - denoted using angle brackets `< ... >`
- **production rules**: expansion of non-terminals into terminals and/or non-terminals
  - usually expand the leftmost non-terminal first (leftmost derivation)

A formal grammer is context free if its production rules can be applied regardless of the symbols that surround it

Arithmetic CFG Example

`<expr>` $\rightarrow$ `<term> <moreTerms>`

`<term>` $\rightarrow$ `<fractor> <moreFractors>`

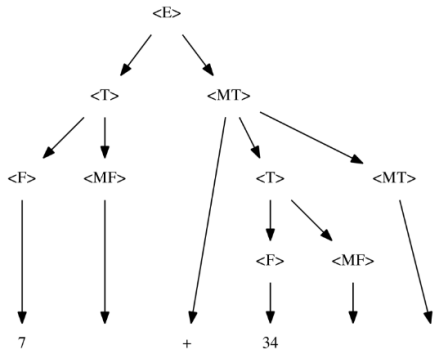
`<term>` $\rightarrow$ `( <expr> ) | int`

`<moreTerms>` $\rightarrow \varepsilon$ `| + <term> <moreTerms> | - <term> <moreTerms>`

`<moreFractors>` $\rightarrow \varepsilon$ `| * <term> <moreFractors> | / <term> <moreFractors>`

A leftmost derivation and parse tree for $7 + 34$ using the rules

<expression>
=> <term> <moreTerms> (R1)
=> <factor> <moreFactors> <moreTerms> (R2)
=> int <moreFactors> <moreTerms> (R4)
=> int <moreTerms> (R8)
=> int + <term> <moreTerms> (R6)
=> int + <factor> <moreFactors> <moreTerms> (R2)
=> int + int <moreFactors> <moreTerms> (R4)
=> int + int <moreTerms> (R8)
=> int + int (R5)



To evaluate the parse tree we used depth first search (DFS)

An alternative and much simplier Arithmetic CFG is

`<expr>` $\rightarrow$ `int | ( <expr> <oper> <expr> )`

`<oper>` $\rightarrow$ `+ | - | * | /`

Language Specification

- good specification → good tree structure → easier evaluation → more automation
- encode associativity (+ vs -) and precedence (+ vs *) in the grammar
- the decision problem of if the grammer has ambiguity is undecidable
  however some ambiguities can be spotted (Ex: same non-terminals in rules)

Tools

- lex: scanner generator (define regular expression rule set)
  - similar tools: flex, quex, etc
- yacc: parser generator (define and associate CFG rules with actions → interpreter)
  - similar tools: byacc, bison, etc

The Assembler

- line by line translation (one assembly instruction to one machine code instruction)
- insert data for .word directive (and possibly other directives)
- ignores commands and blanks lines
- compute and insert address of each label

Instruction Format: Jump

```
oooo ooii iiii iiii iiii iiii iiii iiii
```

- o → opcode
- i → jump target

This is an instruction with large constant operand

- `jump target = high4(PC)+ (i<<2)`
  (highest 4 bits of current PC plus immediate operand multiplied by 4)
- `jal 35528`

  `0000 1100 0000 0000 0010 0010 1011 0010`

Instruction Format: Immediate

```
oooo ooss ssst tttt iiii iiii iiii iiii
```

- o → opcode
- s, t → registers
- i → immediate operand

For instructions iwth register and constant operand

- `lw $1, 24($2)`

  `1000 1100 0100 0001 0000 0000 0001 1000`

Isntruction Format: Register

```
0000 00ss ssst tttt dddd d000 00ff ffff
```

- opcode of `000000`
- s, t, d → registers
- f → function

For instructions with all operands being registers

- `add $1, $2, $3` (add is function `100000`)

  `0000 0000 0100 0011 0000 1000 0010 0000`

- `mult $2, $3` (add is function `011000`) (also notice that `d = 00000`)

  `0000 0000 0100 0011 0000 0000 0001 1000`

Linking

- This combines multiple object files (to avoid compiling the whole program each time)
- resolves external symbols (labels can refer to other object file(s))
- bundles everything to produce a single executable file

Object File Format: Basics

- file header: meta information
- text segment: code
- data segment: static data
- defined external symbols: other objects files can refer to these labels
- undefined external symbols: labels must be found in other object files
- local sybmols (for debugging, relocation)

Relocation

- assembler produces object code starting at 0 so how do we combine multiple of such object files
- relative addresses (`beq $0, $0, 10`) are not a problem
- absolute addresses, static data, must be fixed
  - object file contains list of such code locations
- we adjust actual addresses in object code

Symbol Resolution

- replace symbol names with address
- lables/symbols must be unique (across all linked object files)
- class name and overloading: name mangling
  - C++ exmaple:
    - Before: `int Example::compute(int x, float y);`
    - After: `_ZN7Example7computeEif`

Library

- collection of object files
- with or without preprocessing (internal relocation and name resolution)
- ready for linking with other object files

Loading

- set up memory region(s) for new program
- load executable file from disk (perform late relocation and symbol resolution)
- create and start new process in os

Dynamic Linking

- dynamic linking: relocate and resolve symbols at load time
- dynamic library: combine object code at load time
  (don't add object code to executable file)
- shared library: keep only one copy of object code in memory
  (special memory area or relocatable object code)

Dynamic Shared Library

- dynamic link library (DLL) on Windows
- modification apply to programs (no rebuilding necessary)
- can even rewrite symbols at load time
- slip wrapper between application and library