CS 61C Quest Notes

Jeffrey Shen 303455557

Bits

- N bits is at most 2^n things
- Numerals: Binary (2), Decimal (10), Hexadecimal (16)
- ASCII: for all characters in English Language, 7 bits

Bit Conversions

Decimal	Hex (0x...)	Binary (0b...
00	0	0000
01	1	0001
02	2	0010
03	3	0011
04	4	0100
05	5	0101
06	6	0110
07	7	0111
08	8	1000
09	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Binary → Decimal, Hex → Decimal

- Add up powers of 2/16

0b101

$1\cdot2^2+0\cdot2^1+1\cdot2^0=5$

Binary ↔ Hex

1. Pad left 0s to make multiple of 4
2. Read off groups of 4, using table
3. Drop any leading 0s

Decimal → Binary, Decimal → Hex

1. From left to right, find largest power of 2,16
2. If it fits, subtract and repeat w/ next digit

Overflow: number is too large to be represented, positive or negative

Bit Operations

&	AND	x & y	extract bits, check neither are 1, turn off bits
|	OR	x | y	combine w/ mask, check either are 1, turn on bi
^	XOR (not equal)	x ^ y	flip bits w/ mask
~	complement (flip)	~x	flip bits
<<	shift left	x << n	multiply by 2
>>	shift right	x >> n	divide by 2

1<<n is n 0s to righ of 1

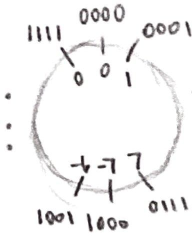
Number Representations

N bits

① Sign and Magnitude $[-(2^{N-1}-1), 2^{N-1}-1]$

Negation: Leftmost bit is sign bit, 0:positive 1:negative

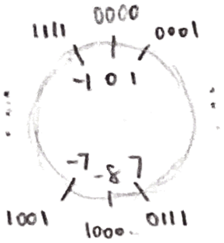
- 2 zeros



② One's Complement $[-(2^{N-1}-1), 2^{N-1}-1]$

Negation: Flip the bits

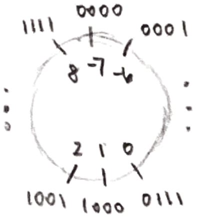
- 2 zeros, leftmost bit tells sign



③ Two's Complement $[-2^{N-1}, 2^{N-1}-1]$

Negation: Flip bits and add one

④ Bias Notation $[b, 2^{N}-1+b]$



$b = -7$

- Shift on unsigned notation

$$x + b = n$$

x ↑ unsigned, b ↑ bias, n ↑ number represented

5) Unsigned $[0, 2^{N}-1]$

- No negative numbers, max at $2^{N}-1$

C

- Allows us to exploit underlying architecture, created in 1970s, C99
- Files first pass through C Pre-Processor where macros replace functions

Variable types

Ex)			
char:	8 bits	(1 byte)	
int:	32 bits	(4 byte)	
int *:	32 bits depending on machine		[index into memory array]

False values: '\0', 0, NULL

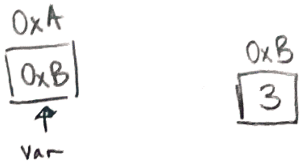
Pointer Arithmetic: increment in sizeof(pointer_type)

Structs: structured groups of variables

- sizeof: returns number of bytes

Pointers (type *var)

- Stores an address
- dereference operator (*): gets value at address
- C passes parameter by value, pass pointer
- If a variable is not initialized, it holds garbage
- Arrow Notation: var → x same as (*var).x



&var = 0xA
var = 0xB
*var = 3

Memory Allocation

malloc(byte-size): returns void * pointer, initialzes with default garbage

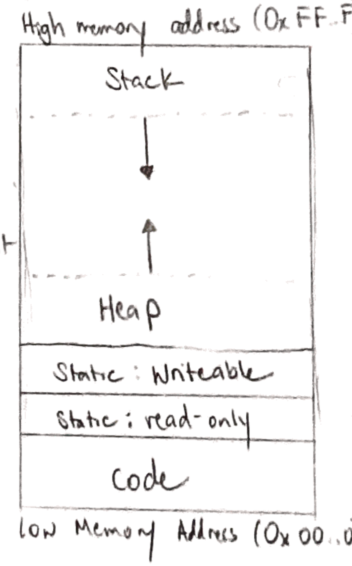
free(ptr): must free any malloc'ed point, once

realloc(ptr, size): reallocate memory

- After allocating memory, check if pointer is NULL

Memory Management

1. Stack: function local variables, strings allocated as arrays (LIFO)
2. Heap: dynamically allocated memory (malloc, calloc, realloc)
   Not necessarly contigous, fragmentation is an issue, circular Linked List
3. Static: global variables, statically allocated strings, basically permanent memory
4. Code: machine instructions



NOTE: - memory of pointers and what they point to may be in different memory

- char* s1 = "cs61c" s1[0] is in static, read-only
- char s2[] = "cs61c" s2[0] is in stack

Tips

- strlen(char*): # chars(bytes) in string not including terminat
- Big Endian: lowest address on left
- Little Endian: lowest address on righ
- "\n" after print
- arr is pointer to first element in array

C Problems

- Draw out diagram w/ addresses
- Check if malloc or parameters are NULL

CS 61C Midterm Notes Jeffrey Shen

Floating Point Representation

Single Precision Representation (32 bit)

(IEEE Standard)

31	30 23	22 0
Sign	Exponent	Significand / Mantissa
1 bit	8 bits	23 bits

$$\text{Value} = (-1)^{\text{sign}} \times 2^{\text{exp}+\text{bias}} \times (1 + \text{Significand})$$

Denorm: $\text{Value} = (-1)^{\text{sign}} \times 2^{\text{bias}+1} \times (0.\text{significand})$

- Allows smallest value a~ 2^{-149}

Exponent	Significand	Object
0	0	0
0	nonzero	Denorm (+/-)
1-254	any	+/- floating pt #
255	0	+/- ∞
255	nonzero	NaN (+/-)

RISC-V

Instructions: [operation] [destination], [source 1], [source 2]

- RISC-V is little Endian - lowest byte on right w/ lowest address
- Program Counter: Internal Register holding byte address of next instruction
- Pseudo-instructions: Shorthand syntax for common assembly idioms (mv, li, nop)

Function Calls

Steps

1. Put arguments in registers for function (a0-a7)
2. Transfer control to function (jal)
3. Acquire local storage resources (prologue)
4. Perform desired task of function
5. Put return value into register and release local storage
6. Return control to point of origin (ret)

Calling Convention

In function (callee), at prologue, epilogue

Save: sp

S registers (s0-s11)

ra ?

Before calling function (caller), save:

ra

t registers (t0-t6)

a registers (a0-a7)

Ex

```
Prologue:  addi  sp, sp, -8
           sw    s1, 4(sp)
           sw    s0, 0(sp)

Epilogue:  lw    s0, 0(sp)
           lw    s1, 4(sp)
           addi  sp, sp, 8
           jr    ra
```

Instruction Formats

R: register register arithmetic

I: immediate arithmetic, loads

S: Stores

B: Branches

U: Upper Immediate

J: Jumps

R-Format (Register)

31 25	24 20	19 15	14 12	11 7	6 0
funct 7	rs2	rs1	funct 3	rd	opcode
7	5	5	3	5	7

Operations : add, sub, sll, slt, sltu, xor, srl, sra, or, and

I-Format (Immediate)

Immediate values
(-2048, 2047)

31 20	19 15	14 12	11 7	6 0
imm [11:0]	rs1	funct 3	rd	opcode
12	5	3	5	7
[offset]	[base]	[width]	[dest]	[LOAD]

Operations : Immediates : addi, slti, sltiu, xori, ori, andi, slli, srli, srai

Loads : lb, lh, lw, lbu, lhu

Jumps : jalr

S-Format (Stores)

sw src, offset(base)

31 25	24 20	19 15	14 12	11 7	6 0
imm [11:5]	rs 2	rs1	funct3	imm [4:0]	opcode
7	5	5	3	5	7
offset	src	base	width	offset	STORE

Operations : sb, sh, sw

B-Format (Branches)

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm [10:5]	rs 2	rs1	funct 3	imm [4:1]	imm [11]	opcode
1	6	5	5	3	4	1	7

Implicit imm [0] = 0

PC = PC + offset

Operations: beq, bne, blt, bge, bltu, bgeu

U-Format (Upper Immediate)

31 12	11 7	6 0
imm [31:12]	rd	opcode
20	5	7

li will handle case of sign extend

Operations : lui, auipc

J-Format (Jumps)

			12	11 7	6 0
imm[20]	imm [10:1]	imm[11]	imm [19:12]	rd	opcode
1	10	1	8		

jal: rd = PC+4
PC = PC + offset

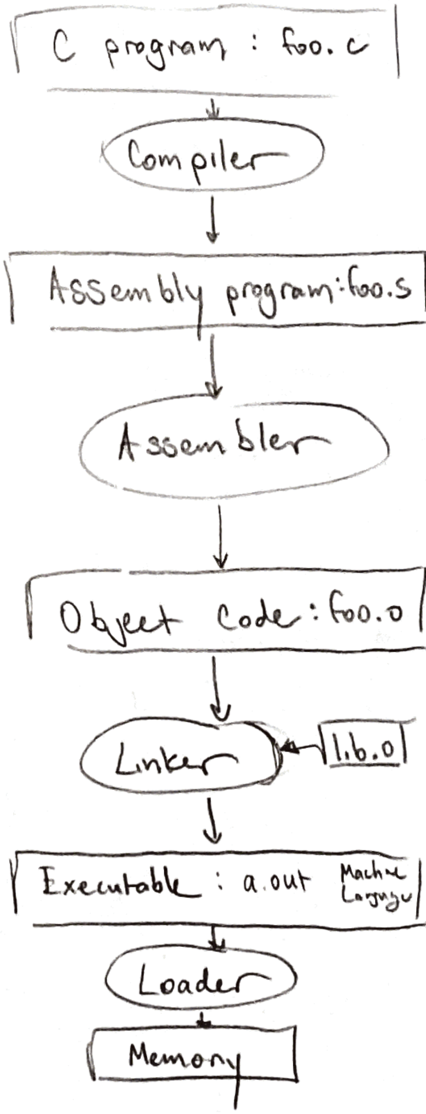
jalr: rd = PC+4
PC = rs1+ imm

implicit 0

Operations : jal

Compiler, Assembler, Linker, Loader (CALL)

- Interpret a high level language when efficiency not critical to exec other programs
- Translate to lower level language to increase performance, hide source



Compiler

High level language → Assembly Language (C → RISC-V)

- Translates language

Assembler

Assembly language → object files (RISC-V → file.o)

- Replace pseudo-instructions
- Two passes to determine offset between jumps
- Reads and uses directives (.text, .data, .string, .globl)
- Produces relocation tables (to fill in later when you link, in other files) and symbol tables (list of items in this file that may be used/referenced)

Linker

object files → executable code (file.o → a.out)

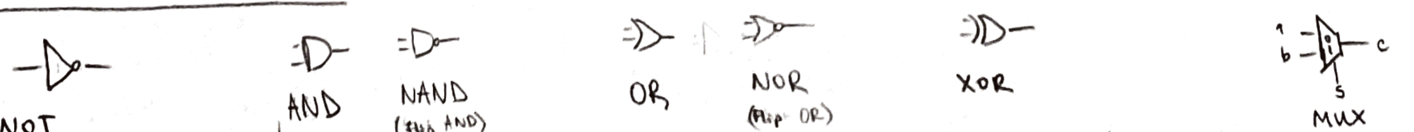
- Fulfills missing labels in relocation symbol tables
- Combines object files into binary executable

Loader

executable file → program run

- Reads header and creates memory space, set up for execution

Combinational Logic



NOT: flips input

AND: 1: both inputs 1, 0: otherwise

OR: 1: at least one input 1, 0: otherwise

XOR: 1: inputs different, 0: otherwise

$x\bar{y} + \bar{x}y$ or $(x+y)(\bar{x}+\bar{y})$

MUX: pick among inputs, S input selects one of 2^s

Boolean Algebra

+ : OR

* : AND

$x + yz = (x+y)(x+z)$ distribution

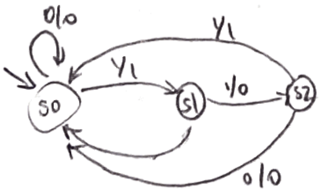
$xy + x = x$ $(x+y)x = x$ uniting theorem

$\overline{xy} = \bar{x}+\bar{y}$ $\overline{(x+y)} = \bar{x}\cdot\bar{y}$ DeMorgan's Law

Canonical Forms : sum of products

- $y = \bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c + a\bar{b}\bar{c} + ab\bar{c}$

Can find simplified using truth table



Finite State Machines : number of states, transitions

- State denoted as start state, one arrow for input
- X/Y where input X, then output Y following arrow

Sychronous Digital Systems

Registers : Controlled by clock, storage object

Value updated at "rising edge of the clock" otherwise constant

- Setup time : time before rising edge where input must be stable
- Hold time : time after rising edge where input must be stable
- Clock-to-Q Delay : time it takes for register's input to become its output after rising edge

Critical Path : longest delay between state elements

$$t_{clk} \geq t_{clk\text{-}to\text{-}q} + t_{logic} + t_{setup}$$

↑ longest logical path

Hold time requirement : $t_{clk\text{-}to\text{-}q} + t_{logic, shortest} > t_{hold}$

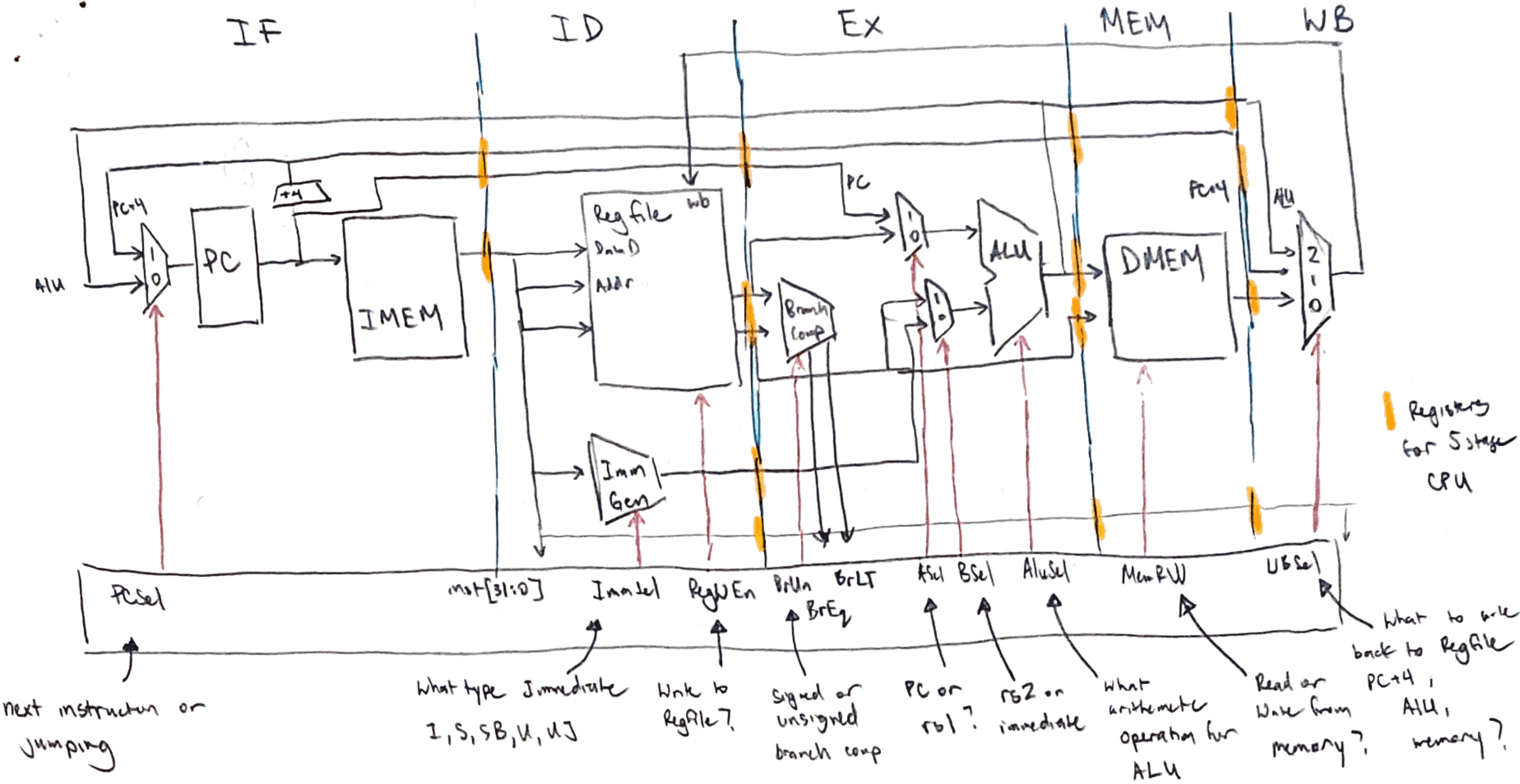
Datapath

Processor (CPU): implemented directly in hardware (ISA)

- Datapath : contains hardware to perform operations
- Control : part of processor telling datapath what needs to be done

5 Stages of Data path

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execute - ALU (EX)
4. Memory Access (MEM)
5. Write Back to Register (WB)


IF
ID
EX
MEM
WB
ALU
PC+4
1
0
PC
+4
IMEM
Reg file
wb
DataD
Addr
Imm Gen
PC
Branch Comp
1
0
ALU
DMEM
PC+4
ALU
2
1
0
Registers for 5 stage CPU
PCSel
inst[31:0]
ImmSel
RegWEn
BrUn
BrEq
BrLT
ASel
BSel
AluSel
MemRW
WBSel
next instruction or jumping
What type Immediate I, S, SB, U, UJ
Write to Regfile?
Signed or unsigned branch comp
PC or rs1?
rs2 or immediate
What arithmetic operation for ALU
Read or Write from memory?
What to write back to Regfile PC+4, ALU, memory?

Critical Path for Single cycle $= t_{clk\text{-}to\text{-}q} + t_{IMEM} + t_{REG} + t_{mux} + t_{ALU} + t_{DMEM} + t_{mux}$

Clock frequency $= \frac{1}{\text{critical path}}$

Pipelining : Add registers between stages to speed up

Latency : time for 1 instruction to finish

Throughput : # instructions processed per unit of time

- Pipeling increases throughput but also latency

Pipelining Hazards

1. Structural Hazards
   - more than one instruction needs to use resource

   caused by : register file ID, WB , Memory IMEM , DMEM

   Solved by : hardware

2. Data Hazards
   - data dependences between instructions

   caused by : instruction reads register before prev finished writing

   solved by : 1. Forwarding : result of EX, MEM sent to EX for next

   2. Stalls (lw) nop to stall

3. Control Hazards

- Jump and branch and unsure of next PC

caused by : jump and branch instructon

solved by : 1. Branch predicton : predict where to go from prev
kill instructon if not taken

2. Stall

Double Pumping : allows writing and reading from reg file in one stage

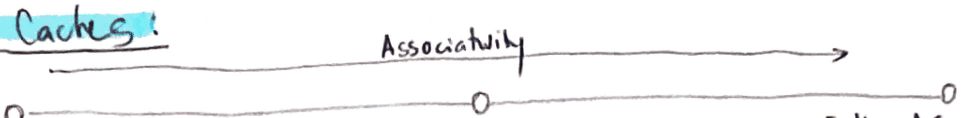
CS 61C Final Notes Jeffrey Shen

Caches

Caches store information closer to processor for easy access

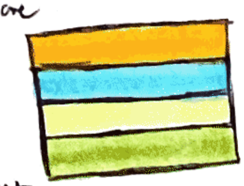
Organize data by: 1) Spatial Locality - locality w/ respect to location
2) Temporal Locality - locality w/ respect to time

Types of Caches:



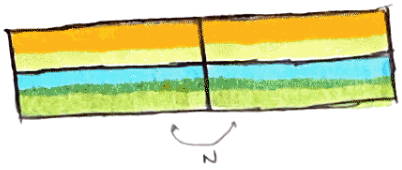
Direct Mapped

Each memory address associated with one possible block within a cache
1 compare



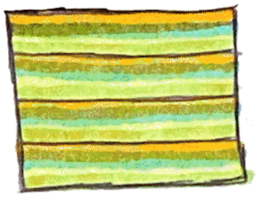
N-Way Associative

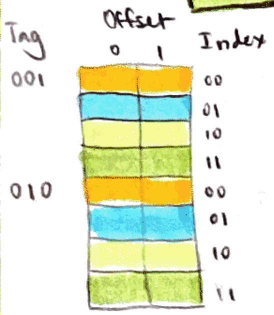
Each set is fully associative with N blocks in it
N compares



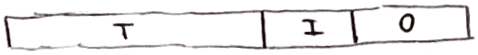
Fully Associative

Block can go in any row
Many compares





Address:



Tag

Distinguish different blocks that use the same index

Bits: Address bits - Index bits - Offset bits

Index

Specifies set that memory will be in

Bits: $\log_2(\#\text{sets})$

$\#\text{sets} = \#\text{blocks}/N$ ← blocks per set

$\#\text{blocks} = \text{Cache Size}/\text{Block Size}$

Offset

Location of the byte in the block

Bits: $\log_2(\text{blocksize (bytes)})$

Cache size:
index (# blocks) × offset (B/block)
$2^{H+W} = 2^H \times 2^W$

Write Policy

1) Write Through: Write updates both cache and memory, Slower but keeps memory up-to-date and consistent

2) Write Back: Only cache updates, but marks as dirty bit, when block evicted, update memory faster but worse at keeping memory coherent

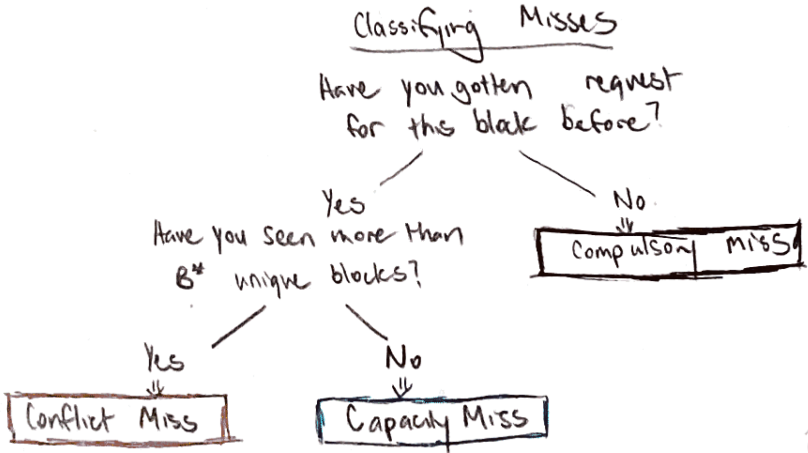
Cache Misses

1) Compulsory: first time bringing in block

2) Conflict: two blocks mapped to same index, not enough room

3) Capacity: Cache not big enough to hold every block

4) Coherence: caused by coherence traffic w/ other


Classifying Misses
Have you gotten request for this block before?
Yes
No
Compulsory Miss
Have you seen more than B* unique blocks?
Yes
No
Conflict Miss
Capacity Miss

Average Memory Access Time (AMAT)

AMAT = Hit Time + Miss Rate * Miss Penalty

↑ Can recursively find Miss Penalty for next level

Hit Rate: fraction of accesses that hit the cache

Miss Rate: 1 − Hit Rate

Miss Penalty: time to replace block from level level in hierarchy

Hit Time: Time to access cache memory

Cache has valid bit to indicate if tag entry is valid

2^{xy} means...

x=0	--	y=0	1
x=1	kibi ~ 10^3	y=1	2
x=2	mebi ~ 10^6	y=2	4
x=3	gibi ~ 10^9	y=3	8
x=4	tebi ~ 10^{12}	y=4	16
x=5	pebi ~ 10^{15}	y=5	32
x=6	exbi ~ 10^{18}	y=6	64
x=7	zebi ~ 10^{21}	y=7	128
x=8	yobi ~ 10^{24}	y=8	256
		y=9	512

Block Replacement Policies

1) Least Recently Used (LRU)

Cache out block that was accessed least recently

+: temporal locality −: complicated hardware, time to keep track

2) FIFO

Track initial order, ignore accesses

3) Random

Choose block at random to cache out

Virtual Memory

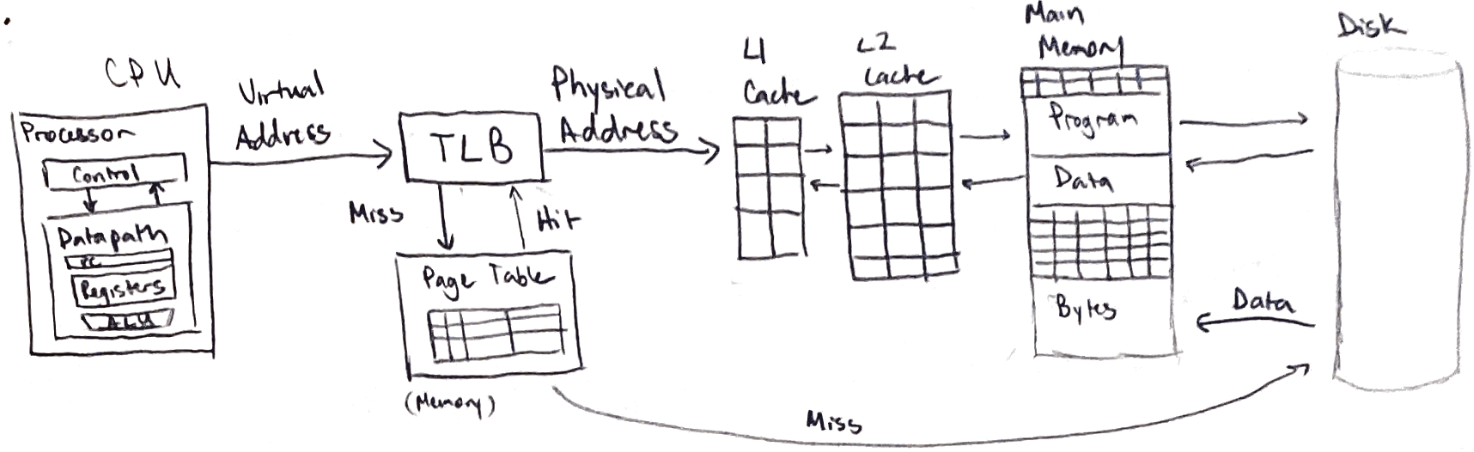
Virtual Memory gives the illusion of infinite memory for applications, prevents one process from accessing another process's memory.

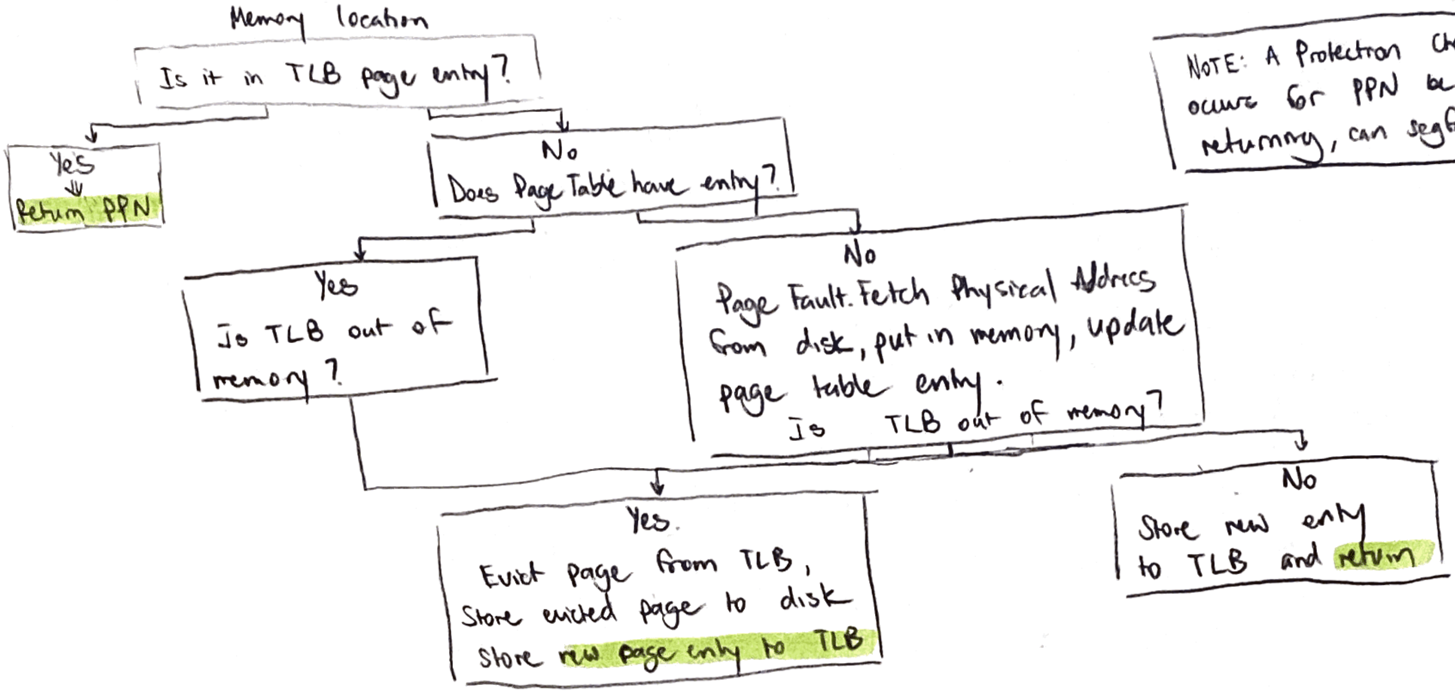
1 Memory > 4 Pages > 128 Blocks > 4096 words > 16,384 bytes

Page Table: table of blocks for each process that maps virtual address to physical address

When switching processes, changes page table by storing in disk and updating "Page Table Base Register" (PTBR) which holds current table address.

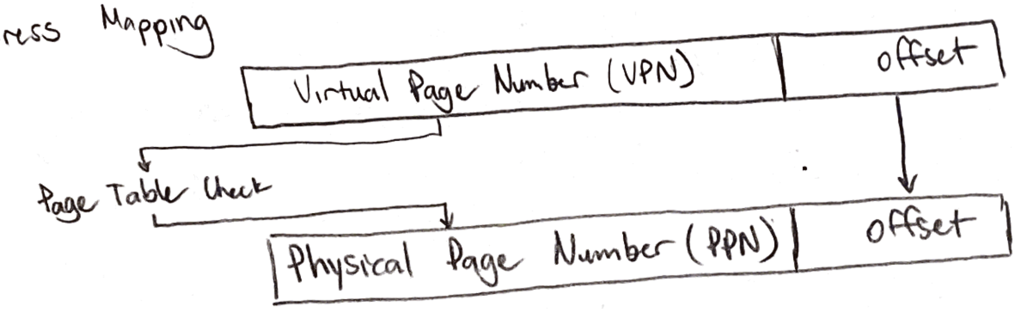
Translation Lookaside Buffer (TLB): hardware which acts as fully associative cache for page table to translate addresses




Memory location
Is it in TLB page entry?
Yes
Return PPN
No
Does Page Table have entry?
Yes
Is TLB out of memory?
No
Page Fault. Fetch Physical Address from disk, put in memory, update page table entry.
Is TLB out of memory?
Yes.
Evict page from TLB, Store evicted page to disk
Store new page entry to TLB
No
Store new entry to TLB and return

NOTE: A Protection Check occurs for PPN before returning, can segfault

Address Mapping



Page Table can have valid bit and dirty bit,

I/O and OS

Polling: Computer keeps checking if data is ready to be used in a loop

+: low latency, overhead, devices w/ constant data -: wasteful if infrequent data

Interrupts: OS notifies when device has data, run interrupt handler to process data

+: do work while waiting, can wait for multiple -: if data comes too quickly, overhead

In practice, set interrupt when data arrives and switch to polling

OS 1) schedules threads/processes 2) provides common services (files, network)

3) Abstractions (virtual memory, system calls)

Flynn's Taxonomy

	Data Streams	
Instruction Streams	Single Instruction Single Data (SISD) one thread one data source ex) most simple programs	Single Instruction Multiple Data (SIMD) Same operation applied to multiple pieces of data ex) Intel AVX
	Multiple Instruction Single Data (MISD) N/A not used anymore	Multiple Instruction Multiple Data (MIMD) different operations on diffent data ex) multi core cpu running diffent programs

Data Level Parallelism, SIMD

Can use extra big registers that hold multiple pieces of data at same time

To use SIMD, iterate over groups of 4 elements, load contigous groups of 4, do math on SIMD Registers, store back, tail case on remaining

Thread Level Parallelism

Thread : logical sequence of instructions, each with own execution state
use shared memory, can split or fork itself

Processes: applications with separate memory

Can use Open MP to spawn new threads for parallel

- pragma omp parallel {...} spawns new thread to execute everything between {}
- pragma omp parallel for spawns a few threads to divide up work on iterations of a for loop

Race Condition

Threads share memory, so different threads can access memory at conflicting times resulting in non deterministic output

We want to limit access to shared resource 1 actor at a time

1) Atomic Read and Write : read and write in single instruction

2) pragma omp critical : only one thread can enter section at a time

3) omp reduction(+ : sum) : use reduction to restrict to one doing operation

Cache Coherency

For multi core processors, want to keep cache up-to-date

Each cache tracks state of each block in cache (MOESI)

1) Modified : data is changed, no other cache has a copy, must update memory
2) Owned : has most updated value, others may have copy
3) Exclusive : no modifications to block, no other core has copy
4) Shared : block up to date, other cores may have copy
5) Invalid : block not in cache

Coherence Misses : misses caused by coherence traffic w/ other processors
data moving between processors working together

Amdahl's (Heart breaking) Law

$$\text{Speedup} = \frac{1}{\underbrace{S}_{\text{non sped up portion}} + \frac{\overbrace{1-S}^{\text{sped up part}}}{\underbrace{P}_{\text{factor of speed up}}}} \leq \frac{1}{S} \quad \text{when } P \to \infty$$

MapReduce / Spark

For large datasets, provide simple ways to transform data that a cluster can then distribute work, improving scalability and fault tolerance

Operations :

1) map : apply some function to each element in a data set
2) reduce : combine elements of list together by some operation
3) reduceByKey : combine key/value pairs that share the same key by some operation

Ex)
```
.map(x: (x,1))
.reduceByKey(x, y: x+y)
.reduce( x,y: a if a[i] > b[i] else b)
```

Data Centers / WSC

Response Time / latency: time between start and finish

Throughput / Bandwidth: total amount of work in a given time

Power Usage Effectiveness (PUE): measure of effectiveness, power for cooling etc

PUE = total building power / IT equipment power 1 = perfect

Dependability

Reliability: Mean Time to Failure (MTTF)

Service Interruption: Mean Time to Repair (MTTR)

Mean Time Between Failures (MTBF)

$$MTBF = MTTF + MTTR$$

$$\text{Availability} = \frac{MTTF}{MTTF + MTTR}$$

To improve: ↑ MTTF
↓ MTTR

Annualized Failure Rate (AFR): avg failures per year

Failures in Time (FIT): Rate of failures in 1 billion dence hours

Error Correcting Codes

Protect against soft errors like struck by alpha particles

Hamming distance: difference in # of bits

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverage	P1	✓		✓		✓		✓		✓		✓	
	P2		✓	✓			✓	✓			✓	✓	
	P4				✓	✓	✓	✓					✓
	P8								✓	✓	✓	✓	✓

RAID

Redundant Arrays of Inexpensive Disks: data stored across multiple disks

0) Split data across multiple disks 1) Mirrored: extra copy 2) Bit level striping

3) Byte level striping 4) Block level parity 5) Block parity check