CS 33 Study Guide

*** This guide aims to solidify important concepts for CS 33. For practice we highly recommend reviewing the many examples covered in lectures, discussions, homeworks, labs, and LA worksheets and workshops. Consequently, this guide does not have any examples or practice problems. It is meant to be a GUIDE.

Bits, Bytes, and Integers

1. Representing Information as Bits

- Everything is bits! They are just interpreted differently based on data type.
- Used to execute instructions or to represent/manipulate data.
- Understand binary and hex representation and to/from decimal.

2. Bit Level Manipulations

- Fill in the truth tables for the following bitwise operations:

AND

A	B	A & B
0	0	
0	1	
1	0	
1	1	

OR

A	B	A | B
0	0	
0	1	
1	0	
1	1	

XOR

A	B	A ^ B
0	0	
0	1	
1	0	
1	1	

- Understand the difference between ~ and ! for 32 bit numbers.
- Understand the difference between arithmetic and logical left and right shifts.
    - HINT: one is used for unsigned numbers and the other is used for signed numbers.
    - What mathematical operations do shifts perform? Are they always precise computations? HINT: think about rounding.

3. Integers

- Know the difference between signed and unsigned integers.
    - Unsigned integers are represented in ______ complement form.
    - Fill in the definitions below and know their binary representations:
        - Smallest unsigned number we can represent: __________
        - Largest unsigned number we can represent: __________
        - Smallest signed number we can represent: __________
        - Largest signed number we can represent: __________
    - Complete the following identities:
        - ~x + 1 == ________
        - TMAX + 1 == ________
        - TMAX + TMIN == _______

- Understand how casting between signed and unsigned numbers works.
- What happens when there is a mix of signed and unsigned integers in an arithmetic operation or comparison expression?

- Understand the conditions for overflow and underflow.

Studying Resources Checkpoint Checklist #1

- ☐ Lecture 1 Notes & Examples - Bits and Bytes
- ☐ Lecture 2 Notes & Examples - Integers
- ☐ Data Lab
- ☐ Homework #1
- ☐ Week 1 LA Worksheet

Machine Level Programming

1. **Assembly Basics**

- Perform arithmetic functions on either registers, memory, or literals (immediates).
- Transfer data from memory into a register and back.
- Transfer control and change instruction execution (unconditional jumps, conditional branches).
- x86 registers:

63	31		15 … 8	7 … 0	
%rax	%eax	%ax	%ah	%al	Return value
%rbx	%ebx	%bx	%bh	%bl	Callee saved
%rcx	%ecx	%cx	%ch	%cl	4th argument
%rdx	%edx	%dx	%dh	%dl	3rd argument
%rsi	%esi	%si		%sil	2nd argument
%rdi	%edi	%di		%dil	1st argument
%rbp	%ebp	%bp		%bpl	Callee saved
%rsp	%esp	%sp		%spl	Stack pointer
%r8	%r8d	%r8w		%r8b	5th argument
%r9	%r9d	%r9w		%r9b	6th argument
%r10	%r10d	%r10w		%r10b	Callee saved
%r11	%r11d	%r11w		%r11b	Used for linking
%r12	%r12d	%r12w		%r12b	Unused for C
%r13	%r13d	%r13w		%r13b	Callee saved
%r14	%r14d	%r14w		%r14b	Callee saved
%r15	%r15d	%r15w		%r15b	Callee saved

***Don't forget %rip.**

- Fill in the following table:

	char	short	int	long	char*	float	double
data type		word					
suffix	b						
size (bytes)						4	

- **MOV**

Instruction	Operand(s)	Effect	Description
mov	S, D	D ← S	Move
movb	S, D	D ← S	Move byte
movw	S, D	D ← S	Move word
movl	S, D	D ← S	Move double word
movq	S, D	D ← S	Move quad word
movabsq	I, R	R ← I	Move absolute quad word

  - Understand the difference between mov and lea.
  - Understand the addressing mode for mov and lea. What do the parentheses indicate in both cases?

D(Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

- D: Constant "displacement" 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for `%rsp`
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

  - What are the possible source and destination operand type combinations?
    - HINT: think registers, memory, and immediates.

- **MOVZ**

Instruction	Operand(s)	Effect	Description
movz	S, R	R ← zero_extend(S)	Move w/ zero extension
movzbw	S, R	R ← zero_extend(S)	Move w/ zero extend byte to word
movzbl	S, R	R ← zero_extend(S)	Move w/ zero extend byte to double word
movzwl	S, R	R ← zero_extend(S)	Move w/ zero extend word to double word

movzbq	S, R	R ← zero_extend(S)	Move w/ zero extend byte to quad word
movzwq	S, R	R ← zero_extend(S)	Move w/ zero extend word to double word

- *__movzlq__ doesn't exist ⇒ happens automatically; movl having register as destination

- **MOVS**

Instruction	Operand(s)	Effect	Description
movs	S, R	R ← sign_extend(S)	Move w/ sign extension
movsbw	S, R	R ← sign_extend(S)	Move w/ sign extend byte to word
movsbl	S, R	R ← sign_extend(S)	Move w/ sign extend byte to double word
movswl	S, R	R ← sign_extend(S)	Move w/ sign extend word to double word
movsbq	S, R	R ← sign_extend(S)	Move w/ sign extend byte to quad word
movswq	S, R	R ← sign_extend(S)	Move w/ sign extend word to quad word
movslq	S, R	R ← sign_extend(S)	Move w/ sign extend double word to quad word

- *__cltq__ - move w/ sign extend %eax (double word) to %rax (quad word); same as movslq %eax, %rax

2. ARITHMETIC AND LOGICAL OPERATIONS

- **LEAQ**

leaq	S, D	D ← &S	Load effective address

- **UNARY**

inc	D	D ← D+1	increment
dec	D	D ← D-1	decrement
neg	D	D ← -D	negate
not	D	D ← -D	complement

- **BINARY**

add	S, D	D ← D+S	add
sub	S, D	D ← D-S	subtract
imul	S, D	D ← D*S	multiply
xor	S, D	D ← D^S	xor
or	S, D	D ← D|S	or
and	S, D	D ← D&S	and

- **SHIFT**

sal	K, D	D ← D << K	left shift
shl	K, D	D ← D << K	left shift
sar	K, D	D ← D $>>_A$ K	arithmetic right shift
shr	K, D	D ← D $>>_L$ K	logical right shift

- **SPECIAL ARITHMETIC**

imulq	S	R[%rdx]:R[%rax] ← S*R[%rax]	signed full multiply
mulq	S	R[%rdx]:R[%rax] ← S*R[%rax]	unsigned full multiply
cqto		R[%rdx]:R[%rax] ← sign_extend(R[%rax])	convert to oct word
idivq	S	R[%rdx] ← R[%rdx]:R[%rax] mod S	signed divide
divq	S	R[%rdx] ← R[%rdx]:R[%rax]/S	unsigned divide

3. CONTROL

- Examples to recall:
    - Jump tables.
    - If-else.
    - Loops (while, do-while, for).
    - Switch statements.

- **CONDITION CODES**

Flag	Name	Description
CF	carry flag	generated a carry out of MSB (detect overflow)
ZF	zero flag	yielded zero
SF	sign flag	yielded negative value
0F	overflow flag	two's complement overflow (positive or negative)

- **CMP**

cmp	S_1, S_2	S_2 - S_1	compare
cmpb	S_1, S_2	S_2 - S_1	compare byte
cmpw	S_1, S_2	S_2 - S_1	compare word
cmpl	S_1, S_2	S_2 - S_1	compare double word
cmpq	S_1, S_2	S_2 - S_1	compare quad word

- **TEST**

test	S_1, S_2	S_2 & S_1	test
testb	S_1, S_2	S_2 & S_1	test byte
testw	S_1, S_2	S_2 & S_1	test word
testl	S_1, S_2	S_2 & S_1	test double word
testq	S_1, S_2	S_2 & S_1	test quad word

- **SET**

Instruction	Operand(s)	Synonym	Effect	Condition
sete	D	setz	D ← ZF	equal/zero
setne	D	setnz	D ← ~ZF	not equal/not zero
sets	D		D ← SF	negative
setns	D		D ← ~SF	nonnegative
setg	D	setnle	D ← ~(SF^OF) & ~ZF	greater (signed >)
setge	D	setnl	D ← ~(SF^OF)	greater or equal (signed >=)
setl	D	setnge	D ← SF^OF	less (signed <)
setle	D	setng	D ← (SF^OF) | ZF	less or equal (signed <=)
seta	D	setnbe	D ← ~CF & ~ZF	above (unsigned >)
setae	D	setnb	D ← ~CF	above or equal (unsigned >=)
setb	D	setnae	D ← CF	below (unsigned <)
setbe	D	setna	D ← CF | ZF	below or equal (unsigned <=)

- **JUMP**

jmp	Label		|	direct jump
jmp	*Operand		|	indirect jump
je	Label	jz	ZF	equal/zero
jne	Label	jnz	~ZF	not equal/not zero
js	Label		SF	negative
jns	Label		~SF	nonnegative
jg	Label	jnle	~(SF^OF) & ~ZF	greater (signed >)

jge	Label	jnl	~(SF^OF)	greater or equal (signed >=)
jl	Label	jnge	SF^OF	less (signed <)
jle	Label	jng	(SF^OF) | ZF	less or equal (signed <=)
ja	Label	jnbe	~CF & ~ZF	above (unsigned >)
jae	Label	jnb	~CF	above or equal (unsigned >=)
jb	Label	jnae	CF	below (unsigned <)
jbe	Label	jna	CF | ZF	below or equal (unsigned <=)

- **CONDITIONAL MOVES (CMOV)**

cmove	S, R	cmovz	ZF	equal/zero
cmovne	S, R	cmovnz	~ZF	not equal/not zero
cmovs	S, R		SF	negative
cmovns	S, R		~SF	nonnegative
cmovg	S, R	cmovnle	~(SF^OF) & ~ZF	greater (signed >)
cmovge	S, R	cmovnl	~(SF^OF)	greater or equal (signed >=)
cmovl	S, R	cmovnge	SF^OF	less (signed <)
cmovle	S, R	cmovng	(SF^OF) | ZF	less or equal (signed <=)
cmova	S, R	cmovnbe	~CF & ~ZF	above (unsigned >)
cmovae	S, R	cmovnb	~CF	above or equal (unsigned >=)
cmovb	S, R	cmovnae	CF	below (unsigned <)
cmovbe	S, R	cmovna	CF | ZF	below or equal (unsigned <=)

Studying Resources Checkpoint Checklist #2

- [ ] Lecture 3 Notes & Examples - Machine-Level Programming I: Basics
- [ ] Lecture 4 Notes & Examples - Machine-Level Programming II: Control
- [ ] Week 2 LA Worksheet

4. Procedures

- **PUSH and POP**

Instruction	Operand(s)	Effect	Description
pushq	S	R[%rsp] ← R[%rsp-8] M[R[%rsp]] ← S	Push quad word

popq	D	D ← M[R[%rsp]] R[%rsp] ← R[%rsp]+8	Pop quad word

- Understand the stack (which direction it grows, what affects the stack):
    - During a procedure call.
    - During pop, push, call, and ret instructions.
    - How and which registers interact with it.
    - In context of caller and callee saved registers.

5. **Data**

- Arrays
    - How to find elements in 1-dimensional, multi-dimensional, and multi-level arrays from a hex dump.
        - HINT: Think Magic 8 Ball and Midterm #3.
- Structs and Unions
    - How to find the size structs and unions.
        - Remember alignment rules.
    - How to find elements in structs and unions from a hex dump.

Studying Resources Checkpoint Checklist #3

- ☐ Lecture 5 Notes & Examples - Machine-Level Programming III: Procedures
- ☐ Lecture 6 Notes & Examples - Machine-Level Programming IV: Data
- ☐ Bomb Lab
- ☐ Homework #2
- ☐ Homework #3
- ☐ Week 3 LA Worksheet

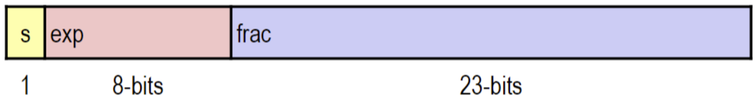
6. **Advanced Topics**

- Understand the uses and locations of the stack, heap, data, and text sections of memory.
- What is buffer overflow, when does it occur, and how does it work?
    - What are some protections against BOF?
    - What is a ROP attack and what challenges does it overcome?

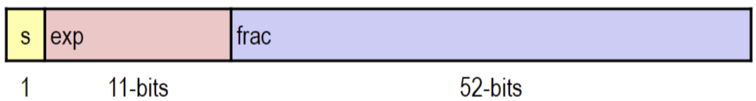
Floating Point

- Understand how to convert IEEE floating point standard format to and from decimal.

Single precision: 32 bits



Double precision: 64 bits



- Understand the significance of the following (-1)^S * M * 2^E.
    - What does S signify?
    - E = exponential_field - ______
    - M = 1.________
- Recall these concepts in the context of floating point: normalized, denormalized, NAN, infinity.
- Understand the complexities of rounding, casting, and arithmetic with floating point.
    - HINT: Look at floating point puzzles.

Studying Resources Checkpoint Checklist #4

- [ ] Lecture 7 Notes & Examples - Machine-Level Programming V: Advanced Topics
- [ ] Lecture 8 Notes & Examples - Floating Point
- [ ] Attack Lab
- [ ] Homework #4
- [ ] Week 4 LA Worksheet

Program Optimization

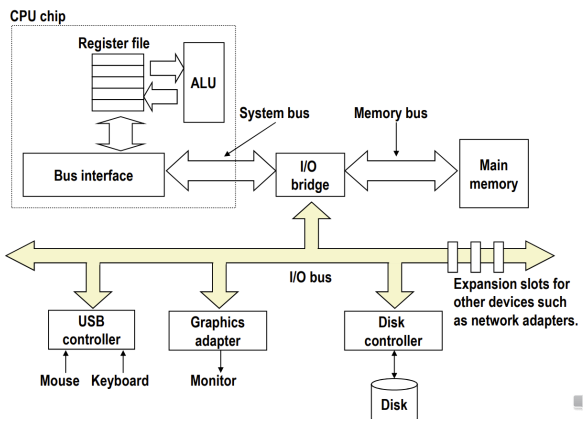
- Understand the following techniques:
    - Code Motion.
    - Strength Reduction.
    - Common Subexpressions.
- Understand the following Optimization Blockers and how to overcome them:
    - Procedure Calls.
    - Memory Aliasing.
- Instruction-Level Parallelism
    - Understand the idea of pipelining.
    - Understand the following techniques:
        - Loop Unrolling.
        - Reassociation.
        - Separate Accumulators.

Studying Resources Checkpoint Checklist #5

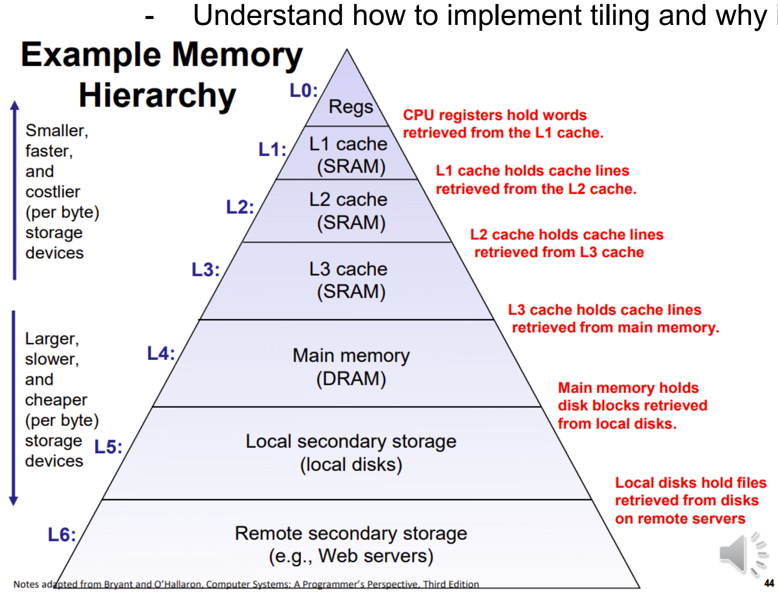
- ☐ Midterm
- ☐ Lecture 9 Notes & Examples - Program Optimization
- ☐ Homework #5
- ☐ Week 5 LA Worksheet

Memory Hierarchy and Cache

- What are the characteristics of volatile and non-volatile memories?
- Understand the flow of read and write operations from and to main memory and disk.



- What is locality (temporal, spatial) and what does it help fix?
- What is a cache and what is its role in the memory hierarchy?
    - Understand the benefits to using caches.
    - Understand when cache hits and misses occur.
    - What are some ways we can write cache-friendly code?
        - How does loop order help exploit cache locality?
        - Understand how to implement tiling and why it exploits cache locality?



Studying Resources Checkpoint Checklist #6

- [ ] Lecture 10 Notes & Examples - The Memory Hierarchy
- [ ] Lecture 11 Notes & Examples - Cache Memories
- [ ] Parallel Lab
- [ ] Homework #6
- [ ] Week 6 LA Worksheet

Parallelism

- Understand the following approaches to parallelism:
    - Domain Decomposition.
    - Task Decomposition.
    - Pipelining.
- Understand the fork-join model.
- What is a lock? Why is it helpful? Why is it not helpful?
    - Lock ______ not _______.
- What are the following and under what circumstances do they occur? What do they do?
    - Race Conditions.
        - How can we avoid race conditions?
    - Mutual Exclusion.
    - Incremental Allocation.
    - No Pre-Emption.
    - Circular Waiting.
    - Deadlock.
        - How can we avoid deadlock?
- OpenMP

1. **Parallel Region**

- **#pragma omp parallel**
    - grand parallelization region with optional work-sharing constructs defining more specific splitting of work and variables amongst threads

2. **Worksharing Constructs**

- Must be enclosed in a parallel region construct and have implicit barriers.
- **#pragma omp parallel for**
    - parallelize a for loop by breaking apart iterations into chunks
- **#pragma omp parallel sections** {
    #pragma omp section { }
    #pragma omp section { } .... }
    - parallelized sections of code with each section operating in one thread
- **#pragma omp single** { }
    - only one thread will execute the section
- **#pragma omp for**
    - parallelize a for loop by breaking apart iterations into chunks

***NOTE: #pragma omp parallel for** and **#pragma omp parallel sections** can be used in place of the parallel region construct containing #pragma omp for and #pragma omp sections respectively.

3. **Synchronization Constructs**
   - **#pragma omp master**
     - only the master thread will execute the following
   - **#pragma omp critical**
     - mutex lock the region
   - **#pragma omp barrier**
     - force all threads to complete their operations before continuing
   - **#pragma omp atomic**
     - like critical, but only works for simple operations and structures contained in one line of code
     - supported operations are ++,--,+,*,- ,/,&,^,<<,>>,| on primitive data types
   - **#pragma omp flush(vars)**
     - force a register flush of the variables so all threads see the same memory
   - **#pragma omp threadprivate(vars)**
     - Applies the private clause to the vars of any future parallelize constructs encountered
4. **Directives and Clauses**
   - **shared(vars)**
     - share the same variables between all the threads
   - **private(vars)**
     - each thread gets a private copy of variables
     - other than the master thread, which uses the original, these variables are not initialized to anything
   - **firstprivate(vars)**
     - like private, but the variables do get copies of their master thread values
   - **lastprivate(vars)**
     - copy back the last iteration (in a for loop) or the last section (in a sections) variables to the master thread copy
   - **default(private|shared|none)**
     - set the default behavior of variables in the parallelization construct
     - shared is the default setting
   - **reduction(op:vars)**
     - vars are treated as private and the specified operation (op, which can be +,*,-,&,|,&,&&,||) is performed using the private copies in each thread
     - the master thread copy (which will persist) is updated with the final value
   - **schedule(static|dynamic|guided)**
     - thread scheduling model
   - **nowait**
     - remove the implicit barrier which forces all threads to finish before continuing in the construct

5. **Using Pragmas with Clauses**

- Not all pragmas can be used with all clauses. Below is a chart to specify which combinations work:

clause	parallel	for	sections	single	parallel for	parallel sections
private	■	■	■	■	■	■
firstprivate	■	■	■	■	■	■
lastprivate		■	■		■	■
shared	■				■	■
default	■				■	■
reduction	■	■	■		■	■
nowait		■	■	■		
num_threads	■				■	■

Studying Resources Checkpoint Checklist #7

- [ ] Lecture 12 + 13 Notes & Examples - Concurrency
- [ ] Week 7 LA Worksheet

Linking

- What are linkers and what do they do? Why are they helpful?
- What are the types of object files?
- Understand static and dynamic linking, their differences, and their advantages and disadvantages.
- Understand the intermediate steps of both types of linking.

Exceptions

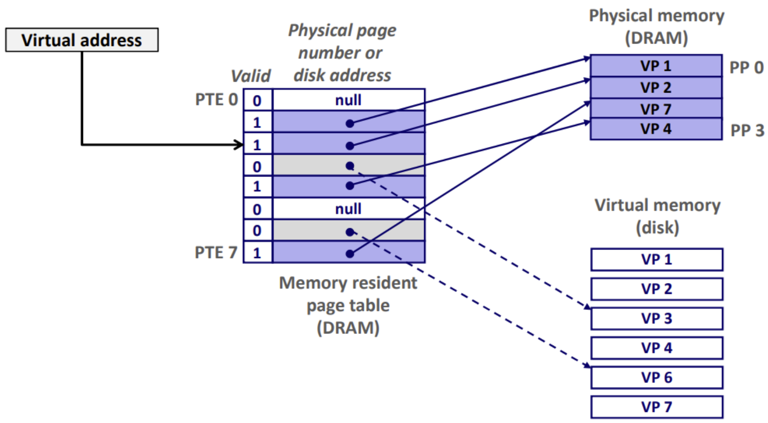
- Processors just read and execute instructions - this is called control flow.
- What is an exception? What is the exception table?
- Understand the following and some examples for each:
    - Asynchronous Exceptions (Interrupts).
    - Synchronous Exceptions (Traps, Faults, Aborts).
- How do system calls work with exception handling?

Studying Resources Checkpoint Checklist #8

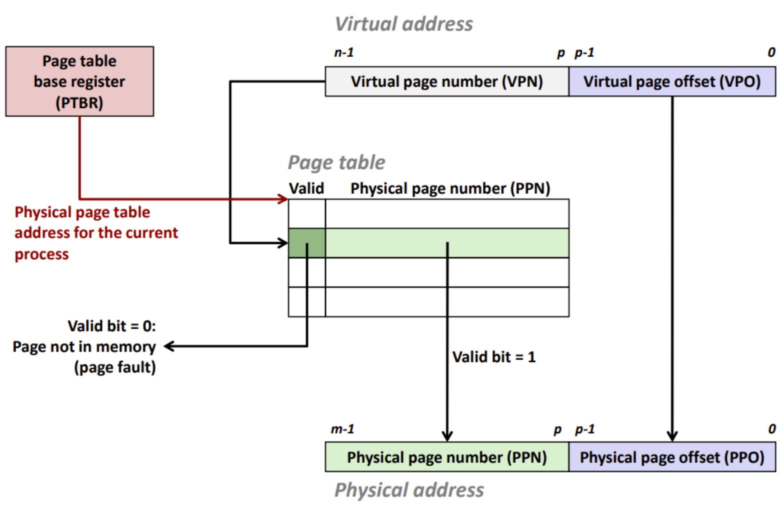
- [ ] Lecture 14 Notes & Examples - Linking
- [ ] Lecture 15 Notes & Examples - Exceptions
- [ ] Week 8 LA Worksheet

Virtual Memory

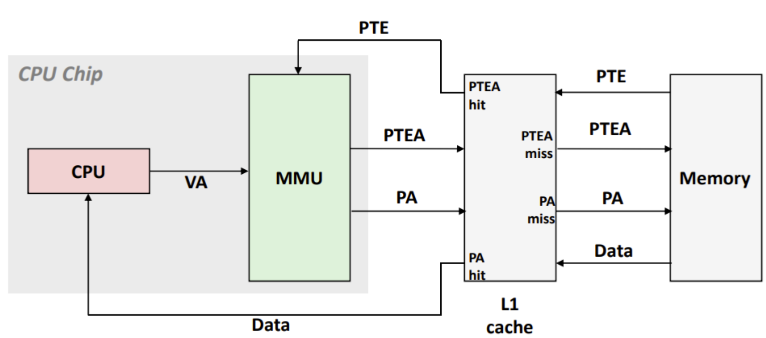
- What is virtual memory? What are its advantages and disadvantages?
- What is the page table? Where does it reside?
- For which pages in the figure below would we get a page hit? Page miss?



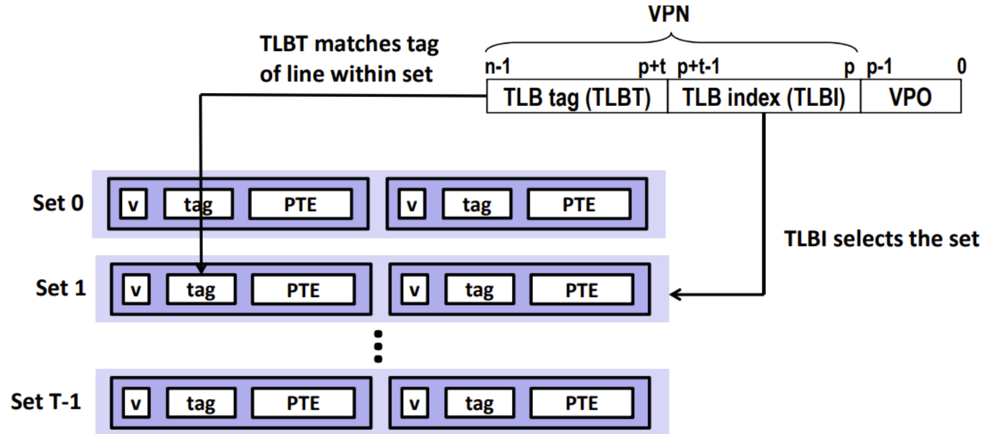
- What happens when we get a page miss? How is it handled?
- How is virtual memory beneficial for memory management?
- How does extending permission bits on page table entries help with memory protection?
- Page Table Address Translation:



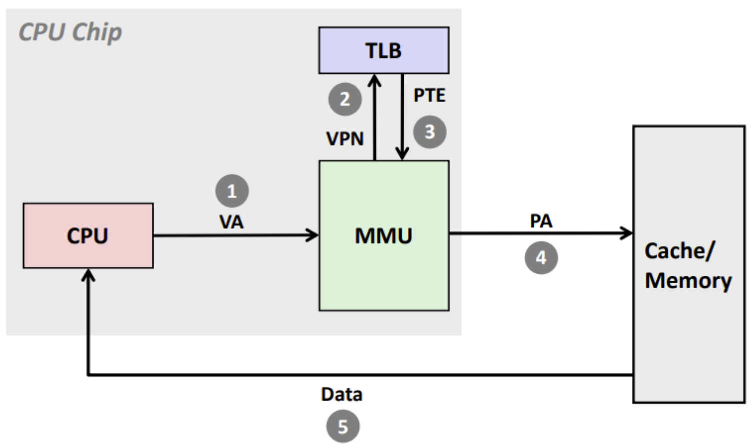
- Page Fetching (no TLB):



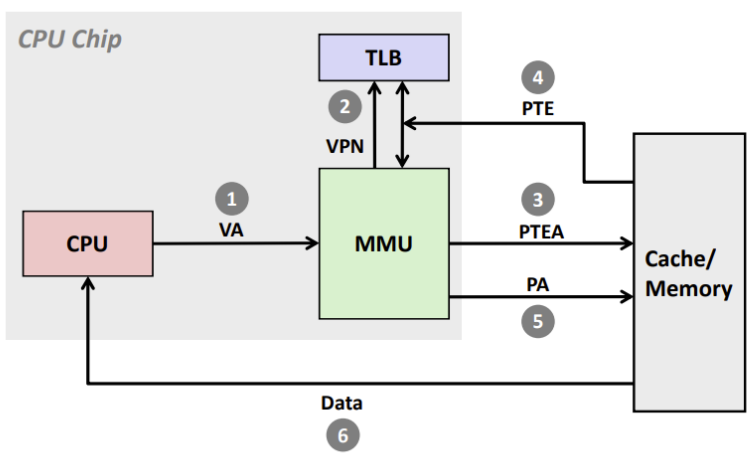
- Page Fetching (TLB):
    - What is the TLB and what does it do? Why is having the TLB beneficial?



    - TLB hit:



    - TLB miss:



- Understand the idea of a multi-level page table.

MIPS

- Understand the differences between RISC and CISC architectures.
- Understand the differences between x86 and MIPS.
    - What size are the registers?
    - How many registers are there?
    - What operand types do the instructions operate on? HINT: think memory, registers, immediates.
    - What length are the instructions?
    - How many instructions are there?
    - How do the addressing modes work?
    - Through what instructions can we interact with memory?
    - What are the differences in syntax? (immediates, registers)

1. Memory Access

lui	rt, imm	Load Upper Imm.	$rt = imm << 16$
lb	rt, imm(rs)	Load Byte	$rt = \text{SignExt}(M_1[rs + imm_\pm])$
lbu	rt, imm(rs)	Load Byte Unsigned	$rt = M_1[rs + imm_\pm]\ \&\ \text{0xFF}$
lh	rt, imm(rs)	Load Half	$rt = \text{SignExt}(M_2[rs + imm_\pm])$
lhu	rt, imm(rs)	Load Half Unsigned	$rt = M_2[rs + imm_\pm]\ \&\ \text{0xFFFF}$
lw	rt, imm(rs)	Load Word	$rt = M_4[rs + imm_\pm]$
sb	rt, imm(rs)	Store Byte	$M_1[rs + imm_\pm] = rt$
sh	rt, imm(rs)	Store Half	$M_2[rs + imm_\pm] = rt$
sw	rt, imm(rs)	Store Word	$M_4[rs + imm_\pm] = rt$

2. Arithmetic

add	rd, rs, rt	Add	$rd = rs + rt$
sub	rd, rs, rt	Subtract	$rd = rs - rt$
addi	rt, rs, imm	Add Imm.	$rt = rs + imm_\pm$
addu	rd, rs, rt	Add Unsigned	$rd = rs + rt$
subu	rd, rs, rt	Subtract Unsigned	$rd = rs - rt$
addiu	rt, rs, imm	Add Imm. Unsigned	$rt = rs + imm_\pm$

3. Logical

and	rd, rs, rt	And	$rd = rs\ \&\ rt$
or	rd, rs, rt	Or	$rd = rs \vert rt$
nor	rd, rs, rt	Nor	$rd = \sim(rs \vert rt)$
xor	rd, rs, rt	eXclusive Or	$rd = rs\ \hat{}\ rt$
andi	rt, rs, imm	And Imm.	$rt = rs\ \&\ imm_0$
ori	rt, rs, imm	Or Imm.	$rt = rs \vert imm_0$
xori	rt, rs, imm	eXclusive Or Imm.	$rt = rs\ \hat{}\ imm_0$
sll	rd, rt, sh	Shift Left Logical	$rd = rt << sh$
srl	rd, rt, sh	Shift Right Logical	$rd = rt >>> sh$
sra	rd, rt, sh	Shift Right Arithmetic	$rd = rt >> sh$
sllv	rd, rt, rs	Shift Left Logical Variable	$rd = rt << rs$
srlv	rd, rt, rs	Shift Right Logical Variable	$rd = rt >>> rs$
srav	rd, rt, rs	Shift Right Arithmetic Variable	$rd = rt >> rs$

4. **Comparison**

slt	rd, rs, rt	Set if Less Than	rd = rs < rt ? 1 : 0
sltu	rd, rs, rt	Set if Less Than Unsigned	rd = rs < rt ? 1 : 0
slti	rt, rs, imm	Set if Less Than Imm.	rt = rs < imm$_{\pm}$? 1 : 0
sltiu	rt, rs, imm	Set if Less Than Imm. Unsigned	rt = rs < imm$_{\pm}$? 1 : 0

5. **Control**

j	addr	Jump	PC = PC&0xF0000000 | (addr$_0$<< 2)
jal	addr	Jump And Link	$ra = PC + 8; PC = PC&0xF0000000 \| (addr$_0$<< 2)
jr	rs	Jump Register	PC = rs
jalr	rs	Jump And Link Register	$ra = PC + 8; PC = rs
beq	rt, rs, imm	Branch if Equal	if (rs == rt) PC += 4 + (imm$_{\pm}$<< 2)
bne	rt, rs, imm	Branch if Not Equal	if (rs != rt) PC += 4 + (imm$_{\pm}$<< 2)
syscall		System Call	c0_cause = 8 << 2; c0_epc = PC; PC = 0x80000080

Service	Code	Arguments	Result
print integer	1	$a0=integer	Console print
print string	4	$a0=string address	Console print
read integer	5		$a0=result
read string	8	$a0=string address $a1=length limit	Console read
exit	10		end of program

6. **Pseudo-**

bge	rx, ry, imm	Branch if Greater or Equal
bgt	rx, ry, imm	Branch if Greater Than
ble	rx, ry, imm	Branch if Less or Equal
blt	rx, ry, imm	Branch if Less Than
la	rx, label	Load Address
li	rx, imm	Load Immediate
move	rx, ry	Move register
nop		No Operation

Studying Resources Checkpoint Checklist #9

- [ ] Lecture 16 Notes & Examples - Virtual Memory
- [ ] Lecture 17 Notes & Examples - MIPS
- [ ] Homework #7
- [ ] Homework #8
- [ ] Week 9 LA Worksheet
- [ ] Week 10 LA Worksheet