Number Representation

2's Complement hex to decimal

Ex. 0x52 = 0b01010010 → 0b10101101 + 1 = 0b10101110

→ 128 - (32 + 8 + 4 + 2) = 82

Two's complement addition

(-2) + (-4) = (-6) → 1110 + 1100 = 11010

→ leading 1 means negative → flip bits = 0101 → add 1 = 0110

Counting

Unsigned binary → Count = 2^n

- Range of unsigned representation is [0, 2^n - 1]
- Negative: [-(2^(n-1) - 1), -0], Positive: [+0, 2^(n-1) - 1]
- Together: [-(2^(n-1) - 1), 2^(n-1) - 1]

One's complement

- Zero: 00...0 and 11...1 = ± 0
- Negative: [-(2^(n-1) - 1) , 0], Positive: [0, 2^(n-1) - 1]
- Together: [-(2^(n-1) - 1), 2^(n-1) - 1]

Two's complement

- Like One's complement, but shift negative numbers by 1 → get rid of 2 zeros
- Zero: 00...0
- Negative: [-2^(n-1) , 0], most negative: 100...0
- Positive: [0, 2^(n-1) - 1], most positive: 011...1
- Together: [-2^(n-1), 2^(n-1) - 1]

Two's complement overflow will never occur when adding two numbers of opposite signs.

Bias N = -(2^(n-1) - 1)

- Non-biased encoding → biased encoding (Subtract the bias)
- Biased encoding → non-biased encoding (Add the bias)
  - Zero: 011...1
  - Most negative: 00...0 (-(2^(n-1) - 1)), Most positive: 11...1 (2^(n-1))
- Range of numbers can be represented: [B, 2^n - 1] → it's like unsigned notation([0, 2^n - 1]) + bias
- IEC standard prefixes

Kibi - Ki - 2^10

Mebi - Mi - 2^20

Gibi - Gi - 2^30

Tebi - Ti - 2^40

Pebi - Pi - 2^50

Exbi - Ei - 2^60

Zebi - Zi - 2^70

Yobi - Yi - 2^80

- 2^{XY}
  - X tells us the prefix
  - Y tells us the number that precedes the prefix (ex: 2 Ti)

X = 0 => no prefix	Y = 0 => 1
X = 1 => kibi	Y = 1 => 2
X = 2 => mebi	Y = 2 => 4
X = 3 => gibi	Y = 3 => 8
X = 4 => tebi	Y = 4 => 16
X = 5 => pebi	Y = 5 => 32
X = 6 => exbi	Y = 6 => 64
X = 7 => zebi	Y = 7 => 128
X = 8 => yobi	Y = 8 => 256

• Write the following in IEC notation: 2^{75} • X = 7 => zebi • Y = 5 => 32 • Answer: 32 Zi	• Write the following as a power of 2: 64 Ti • Ti => X = 4 • Y = log2(64) = 6 • Answer: 2^{46}

C Intro - Basics

C Pre-Processor (CPP) (begin with #) → #define is string replacement

- C source files first pass through CPP before compiler sees code: Foo.c → CPP → foo.i → compiler

What evaluates to FALSE in C? → 0 and NULL

C Intro - Pointers, Arrays, Strings

- sizeof(void) = word size = **4 bytes**
- * → value of the pointer, & → address of a value
- Pointers to functions → *int (*fn) (void *, void *) = &foo;*
  - (*fn) (x, y) call the function
- Null pointers (0s) → Cannot read or write a null pointer, cannot dereference (*p) a null pointer
- Pointers and Structures → Passed by **reference**
- sizeof() operator → **Not a function**, is a static compile-time operation
- If we want a function to change a pointer:
- Segmentation Fault (**SIGSEGV**): invalid access to a valid memory
- Bus Error (**SIGBUS**): access to an invalid address

Copy a string

- a = malloc (sizeof (char) * (strlen (b) **+ 1**));
  strcpy (a, b) or strncpy (a, b, strlen (b) + 1); → strncpy safe, but doesn't copy null terminator

What would strlen(a) return?

- char a = 'f ' → N/A (not a string)
- char a[] = {'f', 'o', 'o', 'b', 'a', 'r'} → N/A (no null terminator)
- char* a = "foo\0bar" → 3

Notes

- (*ip)++ without () is *(ip++) (access ip before increment vs. increment before access)
- ptr = &a[0] is the same as ptr = a
- *(a + i) = a[i] (next element in array)
- &a[i] = a + i
- only print null terminated string
- **n*m matrix** mat = (int**)calloc(n, sizeof(int*));
  for (int i = 0; i<n;i++){mat[i]=(int*)calloc(m, sizeof(int))}

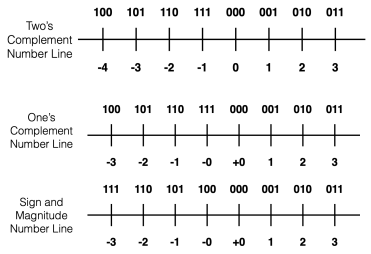
char a[5] = {'a', 'b', 'c', 'd', 'e'}

- a = &a
- a+1 → increment one char over
- &a+1 → increment 5 chars over

```
int x = 5;
int *y = &x;            // y = 0x4000
y = y + 2;              // y = 0x4008
char* z = "bears!"      // z = 0x2000
z = z + 2;              // z = 0x2002
```

Incrementing sizeof(pointer_type)!

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



Memory Management

- **Stack**: Return address, arguments, space for local variables; Stack pointer indicates start of stack frame
- **Heap**: space requested for dynamic data via malloc() or calloc(), resizes dynamically, grows upward
- **Static data**: Global variables and constants declared outside functions;
  - Doesn't grow or shrink, loaded when program starts, can be modified.
- **Code**: loaded when program starts, does not change
- **0x0000 0000** is reserved and unwriteable/unreadable → crush on null pointer access

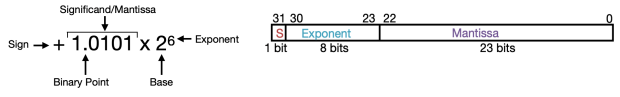
Managing the Heap

- ***void *malloc (size_t n)*** - O(1)
- ***void *calloc (size_t nmem, size_t size)*** - O(N)
- ***void free (void *p)***
- ***void *realloc (void *p, size_t size)***
  - If p is NULL, then behaves like malloc
  - If size is 0, then behaves like free → **realloc (p, 0)**, otherwise it "increases the size"

Bitwise operations

- a >> b: If a is signed: sign extend, If unsigned: zero extend (Not the same as dividing by 2^b due to rounding)
- x & 1 = x; x & 0 = 0; x | 1 = 1; x | 0 = x; x ^ 1 = ~x; x ^ 0 = x
- Turning bits ON → | ; turning bits OFF → &; flipping bits → ^

IEEE 754 Floating-Point Standard



Normalized $(-1)^S \times 1.\text{mantissa} \times 2^{\text{exponent}-127}$

Denormalized $(-1)^S \times 0.\text{mantissa} \times 2^{-126}$

Exponent = 0 and we need to shift the binary point over by 1 to get an implicit leading 0

Exponent (biased notation)

- Range of exponent: [-126, 127]
- Bias formula: $N = -(2^{n-1} - 1)$ → 8 exponent bits
  → bias = -(2^(8-1)-1) = -127

Range of floating-point values

- Smallest positive number we can represent:

0 | 00000001 | 00000000000000000000000

$(-1)^S$ x 1.mantissa x $2^{exponent-127}$

$(-1)^0$ x 1. 00000000000000000000000 x 2^{1-127}

1 x 2^{-126}

2^{-126}

- Largest positive number that we can represent

0 | 11111110 | 11111111111111111111111

$(-1)^S$ x 1.mantissa x $2^{exponent-127}$

$(-1)^0$ x 1.11111111111111111111111 x $2^{254-127}$

.11111111111111111111111 = $2^{-1} + 2^{-2} + \ldots + 2^{-23}$

$\sum_{i=0}^{n-1} 2^i = 2^n - 1$

$2^{-23}(2^{22} + 2^{21} + \ldots + 1)$

$2^{-23}(2^{23}-1) = 1 - 2^{-23}$

Implicit 1 → $1 + 1 - 2^{-23}$

$2 - 2^{-23}$

$(2 - 2^{-23}) \times 2^{127}$

- Positive range $[2^{-126}, (2 - 2^{-23}) \times 2^{127}]$
- Negative range
- Overflow (blue regions), Underflow (red regions)

The only thing that's different is the sign bit, so the range is the same

$[-(2 - 2^{-23}) \times 2^{127}, -2^{-126}]$

Floating Point Step Size

- Next smallest number greater than 2 that we can represent

2	0 | 10000000 | 00000000000000000000001
$(-1)^0$ x 1.0 x 2^1	$(-1)^0$ x 1.00000000000000000000001 x $2^{128-127}$
Exponent = 1 + 127 = 128	$(1+2^{-23}) \times 2$
0 | 10000000 | 00000000000000000000000	$2+2^{-22}$

- If x is the biased exponent and y is the significand
  - Current number: $(1 + y) * 2^{(x-127)}$
  - Next number: $(1 + y + 2^{-23}) * 2^{(x-127)}$
  - Step size = next_num - curr_num

$(1 + y + 2^{-23}) * 2^{(x-127)} - (1 + y) * 2^{(x-127)}$

$= 2^{-23} * 2^{(x-127)} = 2^{(x-150)}$

- The step size increases by a factor of 2 for every time the exponent increases by 1
- The gap between 0 and the smallest positive number is 2^{-126}
- The gap between the smallest positive number and the next smallest positive number is $2^{(x-150)} = 2^{(1-150)} = 2^{-149}$
- There is a large gap between 0 and the smallest positive number due to the requirement of normalization with an *implicit leading one*.

Denorm Range

What is the smallest positive denormalized number that can be represented?	What is the largest positive denormalized number that can be represented?
0 | 00000000 | 00000000000000000000001	0 | 00000000 | 11111111111111111111111
$(-1)^S$ x 0.mantissa x 2^{-126}	$(-1)^S$ x 0.mantissa x 2^{-126}
$(-1)^0$ x 0.00000000000000000000001 x 2^{-126}	$(-1)^0$ x 0.11111111111111111111111 x 2^{-126}
2^{-23} x 2^{-126}	$(1-2^{-23}) \times 2^{-126}$
2^{-149}	$2^{-126}-2^{-149}$

Denorm Step Size

- Current number: $y * 2^{(x-127)}$, Next number: $(y + 2^{-23}) * 2^{-126}$
- Step size = next_num - curr_num

$(y + 2^{-23}) * 2^{-126} - y * 2^{-126} = 2^{-149}$

- The step size is the same for all denorm values because they all have the same exponent

Write $1.5_{10} \times 2^{-129}$ in IEEE-754 Format

Put in normalized binary form: $1.1_2 \times 2^{-129}$ — Exponent is too big for normalized

Put in denorm form: 0.0011×2^{-126}

$(-1)^S$ x 0.mantissa x 2^{-126}

0 | 00000000 | 00110000000000000000000

Floating Point NOT Associative

- Due to rounding errors, (small + big) + big != small + (big + big)

Ex: x = -1.5 x 10^{38}, y = 1.5 x 10^{38}, z = 1.0

x + (y + z)	(x + y) + z
-1.5 x 10^{38} + (1.5 x 10^{38} + 1.0)	(-1.5 x 10^{38} + 1.5 x 10^{38}) + 1.0)
-1.5 x 10^{38} + (1.5 x 10^{38})	0 + 1.0
0	1.0

:ley:EECS Floating Point Addition is not associative!

Type	Exponent	Mantissa
Regular Number	1-254	Anything
Zero	All zeros	All zeros
Infinity	All ones (255)	All zeros
NaN	All ones (255)	Nonzero
Denorm	All zeros	Nonzero

Load Byte (lb) and Store Byte (sb)

- Placed into the lowest byte of the destination register and sign extended

x10: xxxx xxxx xxxx xxxx xxxx xxxx **xzzz zzzz**

…is copied to "sign-extend" — This bit — byte loaded

- lbu which zero extends to fill register
- **Loads** are sign extended
  - Destination is 32 bits while data being stored is 8 bits
- **Stores** are NOT sign extended
  - Location that we are putting data is the same size as the data we want to store (8 bits)
  - When store a byte, only the lower 8 bits of the register is copied to memory, so no sign-extension, so no need of store byte unsigned (sbu)

What value is stored in x12 after the following code runs?

`addi x11,x0,0x3F5`
`sw x11,0(x5)`
`lb x12,1(x5)`
	`addi x11,x0,0x3F5`
	`sw x11,0(x5)`
	`lb x12,0(x5)`
		`addi x11,x0,0x8F5`
		`sw x11,0(x5)`
		`lb x12,1(x5)`

x11 = 0b000003F5	x11 = 000003F5	0x85F gets sign extended
Byte 0 = 0xF5	Byte 0 = 0xF5	x11 = FFFFF8F5
Byte 1 = 0x03	0xF5 gets sign extended	Byte 0 = 0xF5
x12 = 0x00000003	x12 = 0xFFFFFFF5	Byte 1 = F8
		x12 = FFFFFFF8

Shifting

- Logical right shift (zeros inserted on the left)
  - Don't use this for negative numbers
- Arithmetic right shift (left bits are sign extended)
  - Right shift positive & even numbers = dividing by 2^n, truncate fractional part
  - Right shift negative & odd numbers = dividing by 2^n, rounding towards negative infinity
- Shifting to compute address

```
slli x12, x11, 2  # compute offset (x12 = x11 * 4)
add x12, x12, x15 # compute address (x12 = arr + offset)
sw x10, 0(x12)
```

Program Counter (PC)

- move to next instruction, processor increments PC by 4 bytes
- Jump instructions (j → 1) store return address, 2) update PC value
- Jal rd, label : rd = return address, PC = PC + offset (20 bits)
- jal x0, L1 → j L1 (pseudo instruction)

Jump and Link Register (JALR) jalr rd, rs, imm

- Only have 20-bit offset, can't jump to everywhere in memory
- Rd = return address, PC = [rs] + imm
- don't need return address when returned: jalr x0, ra, 0
- Caller: jal ra, callee; callee: jalr x0, ra, 0
- Pseudo instruction
  - jalr x0, rs, 0 → jr rs
  - jalr x0, ra, 0 → jr ra → ret

Calling convention

- Temporary registers → saved by caller
- Saved registers → saved by callee

j label == jal x0, label; jal label == jal ra, label

jal jumps to a label; jalr jumps to an address stored in a register

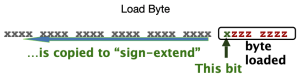
- "Caller" calls a function
  - Saves "caller" saved registers
    - "t" registers (if needed)
    - "a" registers (if needed)
    - ra (if needed)
  - Passes arguments through "a" registers
  - Can expect "s" registers to be saved
- "Callee" is called by another function
  - Handles epilogue and prologue
    - Decrements the stack
    - Saves callee-saved registers ("s" registers)
    - Restores register values before returning and increments the stack
  - Returns value (if any) through "a" registers

R-format
- opcode (7) + funct7 (7) + funct3 (3), rs1 (5), rs2 (5), rd (5)
- opcode: 0110011

I-Format
- imm[11:0] holds values in range [-2048 (base 10), +2047]
- Always sign-extended to 32-bits before use

a = b + c => add a, b, c a = b + 1 => addi a, b, 1
*x = y => sw y, 0(x) y = *x => lw y, 0(x)
int *x = &y => la x, y

Load Instructions



- Load Byte
- Load Byte Unsigned (0-extended)
- Load Halfword (lh)
- Load Halfword Unsigned (lhu)
- lw x10 (rd), 12 (offset) (x15) (rs)

S-Format
- 2 sources reg, no destination, No sign extension
- Store Halfword (sh)
  - sh x14, 36 (x5) → store the lower 16 bits of x14 into memory at address [x5]+36, no sign extension

B-Format {comparison} {reg1} {reg2} {label}
- Range of instructions: 2's complement range: $[-2^{n-1}, 2^{n-1}-1]$
  - with 12 bits: $\pm 2^{11}$ bytes away from the PC
  - Instructions are 4-bytes, so can jump $\pm 2^{9}$ instructions away form the current instruction

Branch Instruction Addressing
- Encoded immediate = 0b0000 0000 0011
- Actual immediate offset = 0b00 0000 0000 1100
- 3 instructions (12 bytes) away
- Can jump $\pm 2^{11}$ instructions ($\pm 2^{13}$ bytes) away from current PC

nx16-bit Instructions
- Las bit is always 0b0 → discard this 1 bit instead of 2
- Range of bytes we can jump to: $\pm 2^{12}$
- Range of 32-bit instructions we can jump to: $\pm 2^{10}$

J-Format jal rd, label
- 20 bits for immediate
- Discard last bit
- Can jump by $\pm 2^{20}$ bytes, $\pm 2^{18}$ 32-bit instructions

JALR is an I-Type Instruction
- jalr rd, rs, imm → PC = [rs] + imm
- Usually used to return from a function (ret)
- Imm[11:0] holds range [-2048, +2047]
- Immediate always sign-extended to 32-bits before use
- Unlike JAL, must include the last 0 because we are using I-format

PC Relative Address
- PC → PC + offset
- Does the value in branch immediate field change if we change the location of the code in memory?
  - If moving all of code, then no (PC-relative offsets)

U-Format

Load Upper Immediate (LUI)
- Destination register = immediate << 12
- lui x5, 0xABCDE → x5 = 0xABCDE000
- LUI + ADDI to create long immediates
  - LUI x10, 0x87654 # x10 = 0x87654000
  - ADDI x10, x10, 0x321 # x10 = 0x87654321
- Load Immediate (LI) → pseudo-instruction
  - LI x10, 0x87654321
- Corner case How to set 0xABCDEEEE?

```
lui x10, ABCDE  # x10 = 0xABCDE000
addi x10, 0xEEE # x10 = 0xABCDDEEE
```

```
  0xABCDE000
+ 0xFFFFFEEE
------------
  0xABCDDEEE
```

```
  0xABCDE     0x000
+ 0xFFFFF   + 0xEEE
-----------   -----
  0xABCDD     0xEEE
```

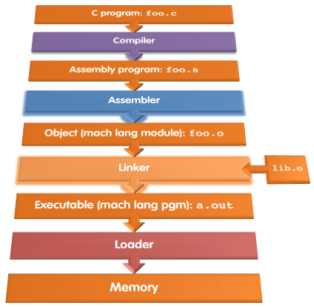
Add Upper Immediate PC (AUIPC)
- rd = PC + (immediate << 12)
  - auipc x5, 0xABCDE → x5 = PC + 0xABCDE000

```
# Call function at any 32-bit absolute address
lui  x5, <hi20bits>
jalr ra, x5, <lo12bits>


# Jump PC-relative with 32-bit offset
auipc x5, <hi20bits>
jalr ra, x5, <lo12bits>
```

Compiler, Assembler, Linker, Loader (CALL)

CALL Chain
- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudo-instructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). A `.s` file becomes a `.o` file.
  - Does 2 passes to resolve addresses, handling internal forward references
- Linker combines several `.o` files and resolves absolute addresses.
  - Enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
- Loader loads executable into memory and begins execution.



Interpretation vs. Compilation
- Interpreter directly executes a program in the source language
  - Easier to write (closer to high-level)
  - Slower
  - Provides instruction set independence
- Translator converts a program from the source language to an equivalent program in another language
  - Both INTERPRETER & COMPILER are translators
  - More efficient

Compiler
- output contain labels, pseudo-instructions, fully relative addressing
- 1) Lexer, 2) Parser, 3) Semantic Analysis and Optimization, 4) Code Generation

Assembler RELATIVE ADDRESSING
- A dumb compiler for assembly language
- Read and uses Directives

.text: Subsequent items put in user text segment (machine code)
.data: Subsequent items put in user data segment (binary rep of data in source file)
.globl sym: declares `sym` global and can be referenced from other files
.string str: Store the string `str` in memory and null-terminate it
.word w1…wn: Store the *n* 32-bit quantities in successive memory words

- Replace pseudo-instructions

la = auipc, addi
li = addi t1, x0, 4
mv rd, rs = addi rd, rs, 0
j loop = jal x0, loop

Pseudo	Real
`nop`	`addi x0, x0, 0`
`not rd, rs`	`xori rd, rs, -1`
`beqz rs, offset`	`beq rs, x0, offset`
`bgt rs, rt, offset`	`blt rt, rs, offset`
`j offset`	`jal x0, offset`
`ret`	`jalr x0, x1, offset`
`call offset` (if too big for just a `jal`)	`auipc x6, offset[31:12]` `jalr x1, x6, offset[11:0]`
`tail offset` (if too far for a `j`)	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`

Producing Machine Language

Simple Case
- Arithmetic, logical, shifts ... → convert into binary representations

Branches → PC-Relative
- "Forward Reference" problem → **2 passes**
- 1st pass: remember position of labels (as expanding pseudo-instructions)
- 2nd pass: uses label positions to generate code

Jumps (j and jal)
- Jumps within a file are PC relative (we can compute)
- Jumps to other files, we can't compute

Reference to Static Data
- la (load address) = lui (when absolute address) or auipc (when relocatable block) + addi
- Can't be determined immediately so create two tables

Symbol Table (non-static labels in present file)
- List of items in the file that may be used by other files.
  - Labels: function calling
  - Data: anything in the .data section; variables which may be accessed across files

Relocation Table (external, static reference)
- List of items that file needs to address of later
  - Any external label jumped to: jal (external, including lib files)
  - Any piece of data in static section (such as la instruction)

Object File Format
- object file header: size and position of the other pieces of the object file
- text segment: the machine code
- data segment: binary representation of the static data in the source file
- relocation information: identifies lines of code that need to be fixed up later
- symbol table: list of this file's labels and static data that can be referenced
- debugging information

Linker ABSOLUTE ADDRESSING

- sum(*.o) ⇒ *.exe
- Enable separate compilation of files
- Process:
  1. Take text segments from each .o file and put them together
  2. Take data segments from each .o file, put them together, and concatenate them onto the end of text segments
  3. Resolve references (go through relocation table, handle each entry → fill in all **absolute addresses** like **printf**)

Three Types of Addresses
- **PC-Relative Addressing** (beq, bne, jal) → never relocate
- **External Function Reference** (usually jal) → always relocate
- **Static Data Reference** (often auipc, addi) → always relocate

Absolute Addresses in RISC-V

- Which instructions need relocation editing?
  - Jump and link (jal) → ONLY for external
  - Loads and stores (lw, sw) to variables in static area, relative to the global pointer
  - Conditional branches → don't need to worry about since PC-relative addressing preserved even if code moves.

Resolving References

- Linker assumes first word of first text segment is at address **0x04000000**
- Liner knows:
  - Length of each text and data segments
  - Ordering of text and data segments
- Liner calculates:
  - Absolute address of each label to be jumped to and each piece of data being referenced
- To resolve references:
  - Search for reference (data or label) in all "user" symbol tables → if not found, search library files → once absolute address is determined, fill in the machine code appropriately
- Output: executable file containing **text** and **data** and **header**

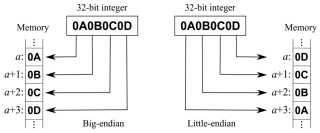
Loader

- Reads executable file's header to determine size of text and data segments
- Creates new address space for program large enough to hold text, data, and stack segments
- Copies instructions and data from executable file into new address space
- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Stack pointer assigned address of 1st free stack location
- Jumps to start-up routine that copies program's argument from stack to registers and sets the PC
- Executable files are stored on disk
- Linker is actually 2 parts: linker in the compiler, linker in the loader

Static vs. Dynamically Linked Libraries

Statically-linked Approach
- Library is part of the executable, so if library updates, we don't get the new update.
- Includes the entire library even if not all of it will be used.
- Executable is self-contained

Dynamically Linked Libraries (DLL)



SDS

Digital → discrete 1, 0
Analog → continuous

Logic Gate	Symbol	Description	Boolean
AND		Output is at logic 1 when, and only when all its inputs are at logic 1,otherwise the output is at logic 0.	X = A•B
OR		Output is at logic 1 when one or more are at logic 1.If all inputs are at logic 0,output is at logic 0.	X = A+B
NAND		Output is at logic 0 when,and only when all its inputs are at logic 1,otherwise the output is at logic 1.	$X = \overline{A \cdot B}$
NOR		Output is at logic 0 when one or more of its inputs are at logic 1.If all the inputs are at logic 0,the output is at logic 1.	$X = \overline{A+B}$
XOR		Output is at logic 1 when one and Only one of its inputs is at logic 1. Otherwise it is logic 0.	X = A⊕B
XNOR		Output is at logic 0 when one and only one of its inputs is at logic1.Otherwise it is logic 1. Similar to XOR but inverted.	$X = \overline{A \oplus B}$
NOT		Output is at logic 0 when its only input is at logic 1, and at logic 1 when its only input is at logic 0.That's why it is called and INVERTER	$X = \overline{A}$

Logic Gates

AND → out = A & B
OR → out = A | B; x | 1 = 1, x | 0 = x
XOR → out = A ^ B = $A\bar{B}+\bar{A}B$
NOT → ~A
NAND → ~(A&B) = $A\bar{B} + \bar{A}B + \bar{A}\bar{B}$
NOR → ~(A|B)
XNOR → ~(A^B) = $(A\bar{B})+AB$

Make XOR with AND and OR

A	B	out
0	0	0
0	1	1
1	0	1
1	1	0

out = $\bar{A}B + A\bar{B}$

Make XNOR

A	B	out
0	0	1
0	1	0
1	0	0
1	1	1

out = $\bar{A}\bar{B} + AB$

(A + B)(A + C) = A + BC
A + AB = A + B
(A+B)(A+B')C = AC

Laws of Boolean Algebra

Name	AND Form	OR form
Commutative	$AB = BA$	$A + B = B + A$
Associative	$AB(C) = A(BC)$	$A + (B + C) = (A + B) + C$
Identity	$1A = A$	$0 + A = A$
Null	$0A = 0$	$1 + A = 1$
Absorption	$A(A + B) = A$	$A + AB = A$
Distributive	$(A + B)(A + C) = A + BC$	$A(B + C) = AB + AC$
Idempotent	$A(A) = A$	$A + A = A$
Inverse	$A(\overline{A}) = 0$	$A + \overline{A} = 1$
De Morgan's	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}(\overline{B})$

How to Build an Adder

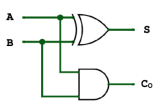
Half Adder

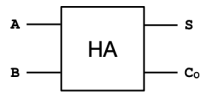
- How to add two bits together using logic gates?

A	B	C_o	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$

$C_o = AB$





Full Adder (single bit)

A	B	C_i	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S is 1 when 1 or 3 of the bits are 1

$S = A \oplus B \oplus C_i$

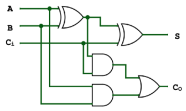
$C_o = \bar{A}BC_i + A\bar{B}C_i + AB\bar{C_i} + ABC_i$

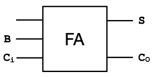
$C_o = C_i(\bar{A}B + A\bar{B}) + AB(\bar{C_i} + C_i)$

$C_o = C_i(A \oplus B) + AB$

$S = A \oplus B \oplus C_i$

$C_o = AB + C_i(A \oplus B)$

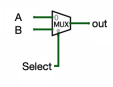




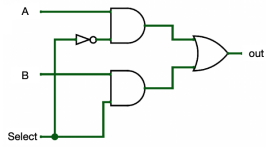
Arithmetic Logic Unit (ALU)

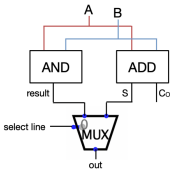
2:1 Multiplexors

- If select = 0, out = AND result; if select = 1, out = ADD result



$out = A\bar{S} + BS$





Combinational Logic

- As soon as the inputs are available, the output starts being computed.
- Output depends only on the current input

D Flip Flops (state element)

- Rising edge triggered → stores D to Q the instant the clock goes from 0 to 1
- Falling edge triggered → stores D to Q the instant the clock goes from 1 to 0

Write enable

- when WE = 0, contents of register stays the same on the clock trigger
- when WE = 1, contents of register updates on the clock trigger

Combinational Logic Delay

- The amount of time that it takes for a value to propagate through the combinational logic

Maximum allowable hold time

- Clk-to-q (reg1) + shortest CL time
- The amount of time that it takes for the input to B to change after the trigger

Minimum acceptable clock cycle (critical path)

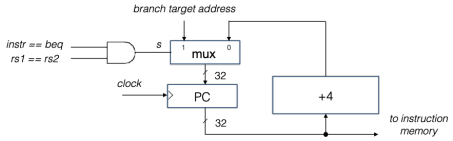
- clk-to-q (reg1) + longest CL time + setup time (reg2)
- Min clock cycle = The time it takes for the input of one state element to reach the input of the next state element.
- Minimum clock period (ns) = critical path
- Maximum clock frequency (MHz) = 1 / (min. clk perio)

*** Hold time < CLK-to-Q delay + min CL delay

*** CLK Period >= CLK-to-Q delay + critical path + setup time

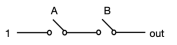
*** clk-to-q >= hold

PC Register

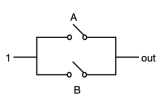


Transistors

Boolean Expressions from Switches



out = AB



out = A + B

DeMorgan's Law $\overline{(A + B)} = \bar{A}\bar{B}$

$(A + B) = \overline{\bar{A}\bar{B}}$

$AB + CD = \overline{(\bar{A}\bar{B})}\,\overline{(\bar{C}\bar{D})}$

NMOS → Gate = 1, switch closed, gate = 0, switch open

- Not good at passing 1s, so hook source to 0

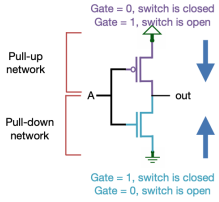
PMOS → Gate =0, switch closed, gate = 1, switch open

- Not good at passing 0, so hook source to 1

Vdd → supply voltage (Logic 1)

Ground → Logic 0

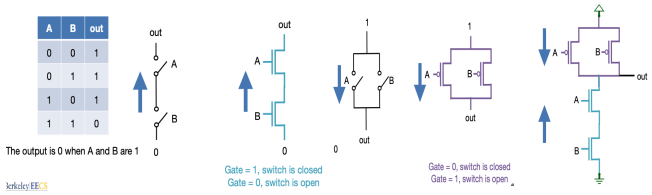
Building an Inverter



Building a NAND Gate

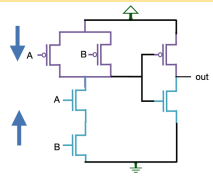
- Output is 0 when A and B are 1, output is 1 when A and B are 0

A	B	out
0	0	1
0	1	1
1	0	1
1	1	0



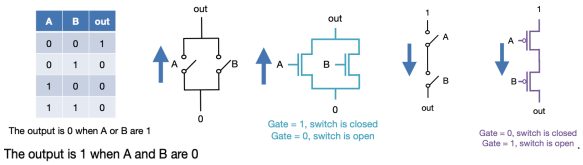
Building an AND Gate → Invert a NAND Gate

A	B	out
0	0	0
0	1	0
1	0	0
1	1	1



Building a NOR Gate

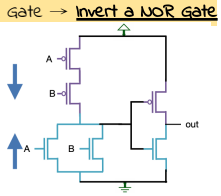
A	B	out
0	0	1
0	1	0
1	0	0
1	1	0

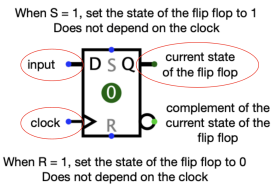


The output is 1 when A and B are 0

Building an OR Gate → Invert a NOR Gate

A	B	out
0	0	1
0	1	0
1	0	0
1	1	0





Transistors

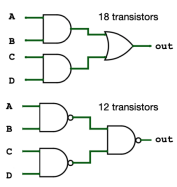
- Inverter
  - 2 transistors
- NAND gate
  - 4 transistors
- NOR gate
  - 4 transistors
- AND gate
  - 6 transistors
- OR gate
  - 6 transistors

Converting ANDs and ORs to NANDs and NORs

$out = AB + CD$

DeMorgan's Law

$out = \overline{(\bar{A}\bar{B})}\,\overline{(\bar{C}\bar{D})}$



Datapath

Phases:

- IF: send address to IMEM, read IMEM at addr
- ID: generate control signals from instruction, immediates, read registers from RegFile
- EX: ALU operations, branch comparisons
- MEM: read/write to DMEM
- WB: write back either PC+4, result of ALU, or data from memory to RegFile

Lw = tclk-to-q(PC) + tMEMRead + tImmGen + tMux + tALU + tMux + tRegFileSetup
Sw = clk-to-q + memread + max(regfileread, immgen) + mux + alu + memwrite
Branches = tclk−to−q(PC) + tMEMRead + max(tRegFileRead, tImmGen)+ max(tBranchComp, tMux + tALU) + tMux(PCSel) + tPCSetup
JAL = tclk−to−q(PC) + tMEMRead + tImmGen+ tMux + tALU + max((tMux(WBSel) + tRegFilSetup),(tMux(PCSel) + tPCSetup))

Control Logic "Truth Table" (incomplete)

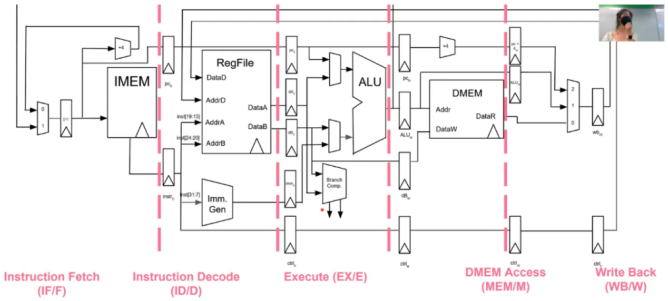
Inst[31:0]	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add	*	*	+4	-	-	Reg	Reg	Add	Read	1	ALU
sub	*	*	+4	-	-	Reg	Reg	Sub	Read	1	ALU
(R-R Op)	*	*	+4	-	-	Reg	Reg	(Op)	Read	1	ALU
addi	*	*	+4	I	-	Reg	Imm	Add	Read	1	ALU
lw	*	*	+4	I	-	Reg	Imm	Add	Read	1	Mem
sw	*	*	+4	S	-	Reg	Imm	Add	Write	0	-
beq	0	*	+4	B	-	PC	Imm	Add	Read	0	-
beq	1	*	ALU	B	-	PC	Imm	Add	Read	0	-
bne	0	*	ALU	B	-	PC	Imm	Add	Read	0	-
bne	1	*	+4	B	-	PC	Imm	Add	Read	0	-
blt	*	1	ALU	B	0	PC	Imm	Add	Read	0	-
bltu	*	1	ALU	B	1	PC	Imm	Add	Read	0	-
jalr	*	*	ALU	I	-	Reg	Imm	Add	Read	1	PC+4
jal	*	*	ALU	J	-	PC	Imm	Add	Read	1	PC+4
auipc	*	*	+4	U	-	PC	Imm	Add	Read	1	ALU

* means "for all values"
- means "don't care, use any value"

12

Instr	BrEq	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add	X	0	X	X	0	0	Add	0	1	1
lw	X	0	I	X	0	1	Add	0	1	0
bge	0/1	0/1	SB	0	1	1	Add	0	0	X
sw	X	0	S	X	0	1	Add	1	0	X
auipc	X	0	U	X	1	1	Add	0	1	1
jal	X	1	UJ	X	1	1	Add	0	1	2

	IF	ID	EX	MEM	WB
add	X	X	X		X
ori	X	X	X		X
lw	X	X	X	X	X
sw	X	X	X	X	
beq	X	X	X		
jal	X	X	X		X
bltu	X	X	X		



- Non-pipelined clock cycle: CP of datapath
- pipelined clock cycle: max(CP of each stage)
- pipelining increases latency (max delay *numcycle) and throughput (1/cycletime)

$$\frac{time}{program} = \frac{instructions}{program} \cdot \frac{cycles}{instruciton} \cdot \frac{time}{cycle}$$

- increase performance by:

	Instructions /program	Cycles/ instruction	Time/cycle
A	decrease	decrease	same
B	same	increase	decrease
C	same	same	decrease
D	increase	decrease	increase

Single cycle fastest clock time: tclk ≥ tPC clk-to-q + tIMEM read + tRF read + tmux + tALU + tDMEM read + tmux + tRF setup

Hazards * reordering code also helps

Structural Hazard - conflict of resource

- solutions: 1) stall, 2) more hardware; 3) separate memories
- RegFile: 2 independent write first, then read

Data Hazard (R-Type) - register access

- solution: separate ports
- Register Access Policy WB updates value, then ID reads new value
- ALU results -> 1) stalling, 2) forwarding (after EX)

```
if (rs1(n + 1) == rd(n) || rs2(n + 1) == rd(n) && RegWen(n) == 1) {
    forward ALU output of instruction n
}
```

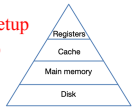
Data Hazard (Load) wait until MEM

- solution: load delay slots

Control Hazard (branch)

- branch prediction to reduce branch penalties -> keep a cache, if seen, backward

IF: PC clk-to-q + IMEM read + Reg setup
ID: Reg clk-to-q + max(RF read, ImmGen) + Reg setup
EX: Reg clk-to-q + max(ALU computation, Branch comparator) + mux
    Branch comparator: PC clk-to-q + branch comp.
    ALU computation: Reg clk-to-q + mux + ALU + Reg setup
MEM: Reg clk-to-q + max(DMEM read, DMEM write) + Reg setup
WB: Reg clk-to-q + mux + RF setup



Fully Associative Caches * only compulsory & capacity miss (no conflict miss)

- temporal locality vs. spatial locality
- Byte offset bits = log2(line size)
- Tag bits = address bits - offset bits

write through policy

- write to cache and memory at the same time

write back policy

- write data in cache and set dirty bit to 1, write to memory when this line gets evicted

Direct-mapped cache

- offset = log2(line size); index = log2(#lines); tag = address - 1 - 0

write-allocate vs. no write-allocate

- write allocate: on write miss, bring the line into cache and update the line
- No write-allocate: on a write miss, only update memory, not bring the line
- write through, no write-allocate
  - write hits: update cache and main memory
  - write miss: not bring to cache, update memory
- write back, write-allocate:
  - Read and write miss: bring line into memory
  - write: only update cache, set dirty bit to 1

Approximate LRU - Clock Algorithm

- Each line has 1 referenced bit, set to 1 each time the line is accessed
- when evicting, start at the first entry in the cache and check reference bit
  - If bit is 1, set it to 0 and move to the next line
  - If bit is 0, evict this line
- The next time need to evict a line, start searching at the point where we left off

When is other eviction policies better than LRU

- 4 cache lines, iterating through an array to access ABCDE, each element maps to a different line in cache
- LRU has 0% hit rate, MRU has 75% hit rate, random has non-zero hit rate

Set Associative Cache

- offset = log2(line size); index = log2(cachesize/(blocksize * associativity)); index = log2(num of sets); tag = address - 1 - 0

Compulsory Miss → first access

Capacity Miss → blocks replaced and later retrieved

Conflict Miss → set-associative or direct mapped, won't occur in fully associative

Average Memory Access Time (AMAT)

AMAT = hit time + miss rate * miss penalty

AMAT = L1 hit time + L1 miss rate * (L2 hit time + L2 miss rate * L2 miss penalty)

Local Hit Rate: # hits at this level / # accesses to this level

Global Hit Rate: # hits at this level / total number of accesses

L1 Miss Rate = global MR/ local MR

Associativity & AMAT

- Associativity increase → hit time increase, miss rate decrease (less conflict misses)
- Associativity cahnges, miss penalty mostly unchanged

Number of Entries & AMAT

- #entries increase → hit time increase (read data from larger memory)
- miss rate decrease (inc capacity dec conflic miss)
- Miss penalty unchanged
- increase in hit time overcome improvement in hit rate at some point, decrease performance

Block Size & AMAT

- Block size increase → hit time mostly unchanged (slightly reduced as #tags reduced);
- miss rate decrease at first due to spatial locality then increase due to increased conflict misses due to fewer blocks in cache
- Miss penalty increase

- increase cache size doesn't always improve hit rate (depends on nature of program).
- increase associativity decreases index bits, increase tag bits
- decrease associativity decrease hit time (check less spots), decrease hit rate (increase conflict misses)
- increase cache size: increase hit rate, increase hit time (longer to search more data)
- "program run indefinitely" = miss rate is 1
- miss penalty → determined by block size

Multilevel Cache

- L2 bigger than L1, so higher hit rate, longer to access (farther fr processor)
- If line in L1 dirty when evicted, update copy in L2
- L2 reduces L1 miss penalty

Parallelism

- Matrix multiplication: Cij = sumk(Aik * Bkj)
- Row major: aij = a[i*N+j], Column major: aij = a[i+j*N]
- C[i+j*N] += a[i+k*N] * b[k+j*N]

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Xeon e5345 (Clovertown)

Amdahl's Law

- Speedup w/ E = Exec time w/o E / Exec time w/ E
- Fraction F, factor S
- Execution time w/ E = Execution Time w/o E * [(1-F)+F/S]
- Speedup w/E = 1 / [(1-F)+F/S]

- Load balancing: every processor do the same amount of work

Two ways to use a multiprocessor

- Deliver high throughput for independent jobs via job-level parallelism
- Improve run time of a single parallel-processing program

Threads

- A sequential flow of instructions that performs some task.
- Each thread has a PC+processor registers and accesses the shared memory of the process
- Each core provides one or more hardware threads
- Common intel chip supports 2 threads/core
- OS multiplexes multiple software threads onto available hardware threads

Hardware multithreading

- Two copies of PC and registers inside processor hardware

- Multithreading → better utilization, share integer adders, floatingpoint units, all caches, memory controller

- Multicore → duplicate processors, share outer caches, memory controller

* OMP thread or OS (software) threads
  - Set num threads: omp_set_num_threads(x)
  - Get num threads: num_th = omp_get_num_threads()
  - Get thread id: th_ID = omp_get_thread_num()
  - Load reserved: lr, rd, rs (load word pointed to by rs into rd)
  - Store conditional: sc rd, rs1, rs2 (store values in rs2 into memory pointed to by rs1 only if reservation is valid and set the status in rd. return 0 if success, return nonzero if location has changed)

Test-and-Set

```
      li t2, 1
Try:  lr  t1, s1
      bne t1, x0, Try
      sc  t0, s1, t2
      bne t0, x0, Try
Locked:
      # critical sec
Unlock:
      sw x0,0(s1)
```

Atomic Memory Operations AMO

- Take the value ***pointed to*** by rs1
  - Load it into `rd`
  - Apply the operation to that value with the contents in `rs2`
    - if `rs2` == `rd`, use the *old* value in `rd`
  - store the result back to where `rs1` is pointed to

omp for reduction(+ : sum)

```
void
const long num_steps = 10;
double step = 1.0/((double)num_steps);
double sum[NUM_THREADS];
for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
    int id = omp_get_thread_num();
    for (int i=id; i<num_steps; i+=NUM_THREADS) {
        double x = (i+0.5) *step;
        sum[id] += 4.0*step/(1.0+x*x);
        printf("i =%3d,  id =%3d\n", i, id);
    }
}
double pi = 0;
for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
printf ("pi = %6.12f\n", pi);
}
```

```
        addi t0, x0, 0
        sw t0, 0(a1)
        ...
        # Assume we now spawn 4 threads in this code
        ...
Check:  jal ra, get_thread_num
        lr.w t0, (a1)
        bne a0, t0, Check
Done:   addi t0, t0, 1
        sc.w a0, t0, (a1)
        bne a0, x0, Check
        ...
        # Assume we now join the 4 threads in this code
        ...
        jr ra
```

for (int x = start; x < start + len/num_threads && x < len; x++)
if ((i / (n/omp_get_num_threads())) == omp_get_thread_num())

__m128i _mm_set1_epi32(int i)	sets the four signed 32-bit integers to i
__m128i _mm_loadu_si128(__m128i *p)	returns 128-bit vector stored at pointer p
__m128i _mm_add_epi32(__m128i a, __m128 b)	returns vector (a0+b0, a1+b1, a2+b2, a3+b3)
__m128i _mm_mullo_epi32 (__m128 a, __m128 b)	returns vector (a0*b0, a1*b1, a2*b2, a3*b3)
__m128i _mm_cmpgt_epi32(__m128 a, __m128 b)	returns: vector((a0>b0)?0xffffffff:0, (a1>b1)?0xffffffff:0, (a2>b2)?0xffffffff:0, (a3>b3)?0xffffffff:0)
void _mm_storeu_si128(__m128i *p,__m128i a)	stores 128-bit vector a at pointer p

Loop Unrolling

```
void multMat4( int n, float *A, float *B, float *C ) {
    int i,j,k;
    /* This is jki loop order. */
    for( j = 0; j < n; j++ )
        for( k = 0; k < n; k++ )
            for( i = 0; i < n; i++ )
                C[i+j*n] += A[i+k*n]*B[k+j*n];
}
```

```
void multMat6( int n, float *A, float *B, float *C ) {
    int i,j,k;
    /* This is kji loop order. */
    for( k = 0; k < n; k++ )
        for( j = 0; j < n; j++ )
            for( i = 0; i < n; i++ )
                C[i+j*n] += A[i+k*n]*B[k+j*n];
}
```

```
void transpose_blocking(int n, int blocksize, int *dst, int *src) {
    // YOUR CODE HERE
    for (int i = 0; i < n; i += blocksize){
        for (int j = 0; j < n; j += blocksize) {
            for (int x = i; x < blocksize + i; x++){
                for (int y = j; y < blocksize + j; y++) {
                    if (x<n && y<n){
                        dst[y+x*n] = src[x+y*n];
```

A. **(0.5 pt)** L1 Number of PTEs

64

L1 PTs * # of PTEs in L1 PT = 1 * 2^6 = 64

B. **(0.5 pt)** L2 Number of PTEs

4096

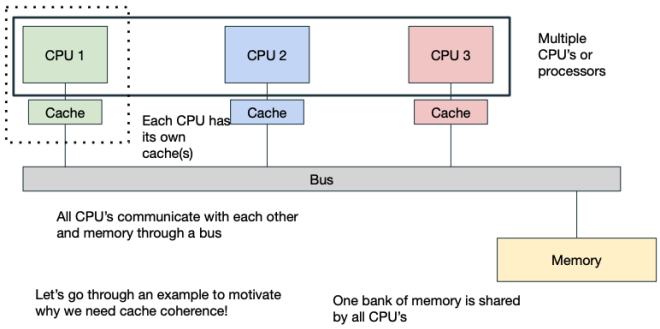
L2 PTs * size of L2 PT = 2^6 * 2^6 = 2^12 = 4096

```
long long int sum_simd(int vals[NUM_ELEMS]) {
    clock_t start = clock();
    __m128i _127 = _mm_set1_epi32(127); // This is a vector with 127s in
    long long int result = 0; // This is where you should put your final
    /* DO NOT MODIFY ANYTHING ABOVE THIS LINE (in this function) */

    for(unsigned int w = 0; w < OUTER_ITERATIONS; w++) {
        /* YOUR CODE GOES HERE */
        __m128i sum = _mm_setzero_si128();
        for (unsigned int i = 0; i < NUM_ELEMS / 4 * 4; i += 4) {
            __m128i curr = _mm_loadu_si128((__m128 *)(vals + i));
            __m128i mask = _mm_cmpgt_epi32(curr, _127);
            mask = _mm_and_si128(mask, curr);
            sum = _mm_add_epi32(sum, mask);
        }
        int temp[4];
        _mm_storeu_si128((__m128i *) temp, sum);
        result += temp[0] + temp[1] + temp[2] + temp[3];
        /* Hint: you'll need a tail case. */
        for (unsigned int i = NUM_ELEMS / 4 * 4; i < NUM_ELEMS; i++) {
            if (vals[i] >= 128) {
                result += vals[i];
            }
        }
    }
```

Shared Memory Multiprocessor (SMP)

- Two or more identical CPUs/Cores, single shared coherent memory



- **M: Modified**
  - I have the only copy, and can write, and its dirty
  - If this gets evicted, I need to flush the entry
- **O: Owned**
  - I have the ***official*** copy, and can write, and its dirty
  - When writing, I have to tell everyone else I'm writing (and it now turns Modified)
- **E: Exclusive**
  - I have the ***only*** copy
- **S: Shared**
  - I have a copy and can read away...
- **I: Invalid (duh)**

- Shared up-to-date data, other caches may have a copy (valid bit set, shared bit set)
- Modified: up-to-date data, changed (dirty), no other cache has a copy, OK to write, memory out-of-date (valid and dirty bit is set)
- Exclusive: up-to-date data, no other cache has a copy, OK to write, memory up-to-date
  - If other cache reads this line, the state becomes shared
  - If write to this line, state becomes modified, no need to broadcast
- Owner: up-to-date data, other caches may have a copy (must be in shared state), memory not up-to-date
  - This cache supplies data on read instead of going to memory
- Invalid

Valid	Dirty	Shared	State
0	0	0	Invalid
0	0	1	Invalid
0	1	0	Invalid
0	1	1	Invalid
1	0	0	Exclusive
1	0	1	Shared
1	1	0	Modified
1	1	1	Owned

Operating System

- Provides isolation between running processes
- Provides interaction with the outside world (external devices)

Illusionist - provide clean, easy-to-use abstractions of physical resources

Referee - manage protection, isolation, and sharing of resources

Context Switch

- Switching from executing one program to another
- Allow multiple processes to run on the same processor
- The OS determines when to context switch
  - The OS takes control of the CPU from the current process; saves the state of the current process; loads the state of the next process; hands over the CPU to the next process

Protection

- OS isolates processes from each other; OS isolates itself from other processes.

Dual Mode Operation

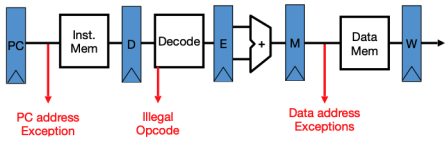
- 1) Kernel Mode ("supervisor"), 2) User Mode (OS mostly run in)
- Switching between user mode and kernel mode
  - System calls, interrupts, exceptions

System Calls (syscall)

- Allows the program to request a service from the operating system
  - Creating and deleting files; reading and writing files; accessing external devices like a scanner; ecalls in RISC-V
- Similar to function calls except it's executed by the kernel

Interrupts vs. Exceptions

- **Interrupts**
  - Caused by an event *external* to the current running program
  - Ex: Key press
  - Asynchronous to the current program
    - Does not need to be handled immediately, but should be handled soon
- **Exceptions**
  - Caused by an event *during* the execution of of the current program
  - Ex: illegal instruction, divide by zero
  - Synchronous
    - Must be handled immediately



Traps

- All instructions before the faulting instructions must complete
- All instructions after the faulting instruction must be flushed
- The faulting instruction must be flushed
- Execution of the trap handler begins

Trap Handler

- code that services the interrupt or exception
- Save the state of the current program (save all of the registers)
- Determin what caused the exception or interrupt
- Handle exception of interrupt
  - Either continue execution of the program
    - Restore the state of the progran
    - Return control to the program
  - or terminate the program
    - Terminate the program (free resources)
    - Schedule a new program

Which path to choose?

- Continue execution: interrupts, certain memory exceptions
- Terminate: illegal instruction, certian illegal memory accesses

Kernel

- Core of OS, manage recourses (scheduling, memory, IO)
- Not in kernel: UI, networking

Boot

1. The BIOS (Basic Input/Output System) runs
   - Power-on-self-test (POST)
   - The BIOS finds and executes the bootloader
2. The bootloader loads in part of the operating system
3. The operating system initializes services, drivers, etc
4. Launch a process that waits for an input in a loop

Bootstrapping: A chain of stages, in which at each stage, a smaller, simpler program loads and then executes the larger, more complicated program of the next stage (Wikipedia)

Berkeley EECS

Executing a program

1. Loader loads program into memory
2. Loader sets argc and argv
3. OS jumps to main and transfers control to the process

Virtual Memory

Page size = 2^offset
#Page table entry = 2^VPN
PTE = PPN + metadata (account word alignment)
PTE size = 2^PTE (in bytes) * #PTE = 2^PTE (in bytes) * 2^VPN
#page = PT size / page size
PPN bits = log2(num pages) = log2(RAM size)
Why VM: 1) RAM not big enough, 2) contiguous data, 3) protection b/w processes

- Q: How many bits would there be for the VPN, PPN, and page offset on a 32-bit machine with 8GB of RAM and 16KB pages?
  - Number of page offset bits = $\log_2(16\text{ KB}) = \log_2(2^4 * 2^{10}) = 14$
  - Number of VPN bits = 32 - 14 = 18
  - Number of PPN bits = $\log_2(8\text{GB}) - 14 = \log_2(2^3 * 2^{30}) - 14 = 33 - 14 = 19$

- How many virtual pages and physical pages would I have in a 32-bit system where I have a page size of 8KB and 8GB RAM?
  - Size of virtual address space: 2^{32}
  - Number of virtual pages: $2^{32} / 8K = 2^{32} / (2^3 * 2^{10}) = 2^{19}$
  - Size of physical address space: $8G = (2^3 * 2^{30}) = 2^{33}$
  - Number of physical pages: $2^{33} / 8K = 2^{33} / (2^3 * 2^{10}) = 2^{20}$

- Number of entries in the page table depend on **Virtual Address Space**
- The **number of entries decreases** as the size of the pages **increases**.
- The **size of each entry decreases** as the size of the pages **increases**.
- With VM, **2 accesses** to make each time to access a piece of data (read page table, use translated address to read data)

Page Fault Exception * page is not in RAM

1. Hardware (CPU) generates a **page fault exception**
2. The hardware jumps to the OS page fault handler to fix it
3. The OS chooses a page to evict from RAM (if no more free space)
4. If the page is **dirty**, it needs to be written back to disk first
5. The OS updates the corresponding PTE to point to disk
6. The OS brings in the page we wanted disk and puts it in RAM
7. The OS updates the PTE of the new page
8. The OS jumps back to the instruction that caused the page fault (This time it won't cause a page fault since the page has been loaded.)

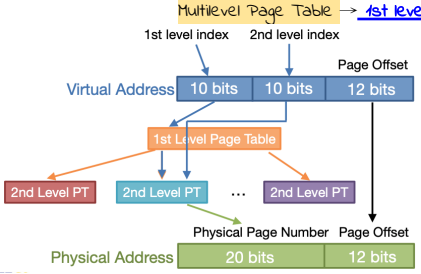
Page Table Metadata

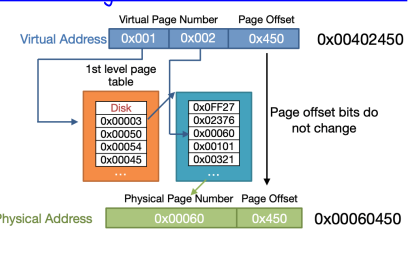
- Valid, dirty
- Page protection: read, write, execute
  - Memory protection fault or segmentation fault

Translation Lookaside Buffer (TLB) → cache for page table

- Must be small to be fast
- Usually fully associative cache, random or FIFO replacement policy
- On a context switch, TLB is flushed.

Multilevel Page Table → 1st level PT must always be in RAM, at leaste 1 2nd level RAM





- Q: If I'm running 10 applications on my 32-bit computer with 4GB RAM, 4KB pages and 1-level page tables, how much of my RAM is consumed by page tables? (size of PTE is 4 bytes)
  - VA size = 32 bits
  - PA size = $\log_2(4\text{ GB}) = \log_2(2^2 * 2^{30}) = 32$ bits
  - # bits in page offset = $\log_2(4\text{ KB}) = \log_2(2^2 * 2^{10}) = 12$
  - # bits in VPN = 32 - 12 = 20
  - # entries in page table = 2^{20}
  - size of each entry is ~4 bytes
  - size of one page table = $2^{20} * 2^2 = 2^{22}$
  - total RAM consumed by pages tables = $10 * 2^{22}$ bytes = 40 MB

- Q: If I'm running 10 applications on my 32-bit computer with 4GB RAM, 4KB pages and a 2-level page table, how much of my RAM is consumed by 1st level page tables? (size of PTE is 4 bytes)
  - # bits in VPN = 32 - 12 = 20 = 10 bits for level 1 + 10 bits for level 2
  - Size of 1st level page table = page size = 4KB
  - total RAM consumed by 1st level pages tables = $10 * 2^{12}$ bytes = 40KB

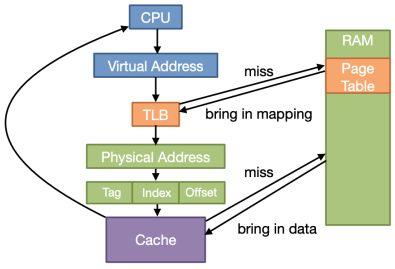
- TLB contains mappings from VPN to the final PPN

Synonyms -> Different virtual addresses can map to the same physical address

Homonyms -> The same virtual address can map to different physical addresses

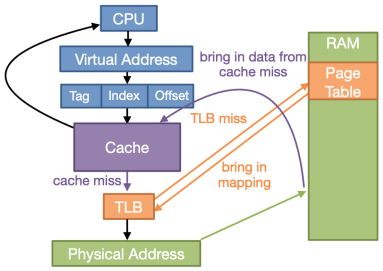
- For **PIPT**, no need to flush the cache since flushing the TLB will force cache replacements (since the new process will point to a different physical address, which is not in the cache).
- For **VIVT**, you need to flush the cache since a different process might use the same virtual addresses.
- For **VIPT**, no need to flush the cache since the tag comparison will fail anyway due to TLB flush (new set of physical addresses = new tags = forced cache replacements like in PIPT).
- In all cases, **TLB always gets flushed when context switching.**

Physically Indexed, Physically Tagged Cache



- Pros: Can use the same cache formultiple processes
- Cons: must translate address before accessing the cache
- Switch process: 1) update PT address register; 2) clear TLB valid bits
- AMAT: translation (TBL, PT) + data access (cache)

Virtually Indexed, Virtually Tagged Cache

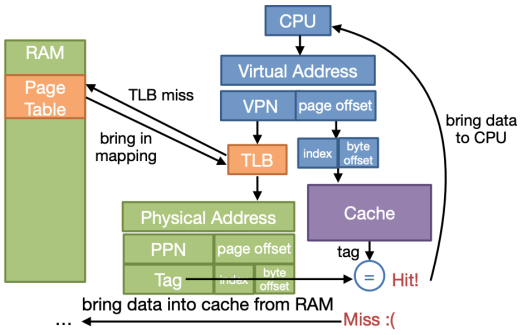


- Cannot use the same virtual cache for different processes (synonyms and homonyms)
- Solve this issue by flushing the cache on context switch

Virtually Indexed, Physically Tagged Caches (VIPT)

- Use physical address to access cache → avoid synonym and homonym
- Size of cache is limited by number of page offset bits
- How to make cache larger with the same page size? → increase associativity

- Q: With 4kB pages, how many bytes can a direct-mapped (1-way) VIPT cache store?
  - 4kB
  - We can only use the page offset bits (12 bits for 4kB pages) to index into the cache. So the index can only address 12 bits of address, or 4kB of data



Memory Mapped IO

- writes to device, hardware intercepts, no actual memory access.

Polling

- CPU periodically checks the status register to see if there is any data to receive
- occur while performing a context switch

- Let's say it takes 400 cycles to complete a polling operation
- We have to poll a mouse 30 times per second. Our clock runs at 1 GHz (1 billion cycle/second).
- How much of our processor time is spent polling?

Number of cycles per second spent polling

30 polls/sec * 400 cycles/poll = 12K cycles/sec

12K cycles/sec / 10^9 cycles/sec = 0.0012%

- I want to read from a disk that can transfer data at a rate of 16 MB/sec
- I can only accept 16B per poll
- The clock runs at 1 GHz (1 billion cycle/second).
- What percent of my processor time is spent polling?

Number of polls needed per second to ensure that no data is missed → (16 MB/sec) / (16 B/poll) = 1M polls/sec

Number of cycles per second spent polling → 1M polls/sec * 400 cycles/poll = 400M cycles/sec

Unacceptable

400M cycles/sec / 10^9 cycles/sec = 40%

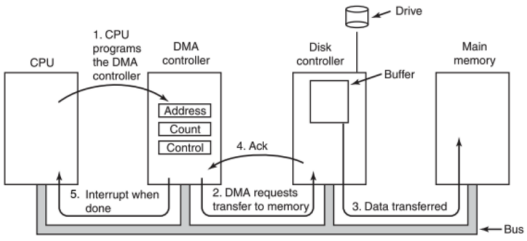
Interrupts

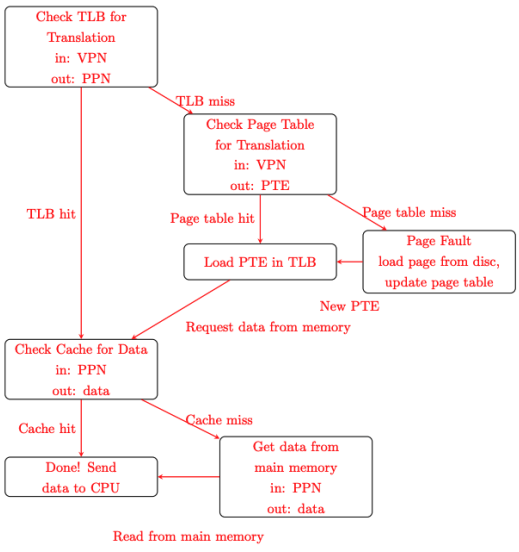
- Polling wastes resources when no data is available from IO device
- Interrupts: stop current program, transfer control to trap handler in OS
- Save program counter and user registers

	Interrupts	Polling
If no I/O activity	No wasted cycles	Lots of wasted cycles
If lots of I/O activity	Expensive – saving/restoring state	Less expensive - only poll when we are already context switching
Better suited for events that are	• asynchronous (unsure when event will occur) • urgent • infrequent	• synchronous (occurs at fixed intervals) • not urgent • frequent (majority of polls are a hit)
Examples	Disk	Keyboard, mouse

Direct Memory Access (DMA)

- DMA engine (hardware), controls movement of data b/t memory and IO devices, allow CPU to do other work
- Initiate transfer, CPU setup: memory address to place data, # of bytes, IO device #, direction of transfer
- Incoming Data
  - Receive interrupt form device
  - CPU takes interrupt, initiates transfer
  - DMA engine handle the transfer (CPU free)
  - Upon completion, DMA engine interrup the CPU again
- Outgoing Data
  - CPU initiate transfer, confirms external device ready
  - CPU initiates transfer
  - DMA engine handle transfer
  - DMA engine interrup CPU to signal completion





Network

Software send steps:

- Application copies data to OS buffer
- OS calculates checksum
- OS sends DMA request to network interface hardware, start

Software receive steps

- Network interface copies data from network interface hardware to OS buffer, triggers interrupt
- OS calculates checksum, if OK send ACK, if not, delete message (sender resend)
- If ok, OS copies data to user address space, signal application to continue

- Hierarchy of layers - network "stack":
  - Application (chat client, game, etc.)
  - Transport (TCP, UDP)
  - Network (IP)
  - Data Link Layer (ethernet)
  - Physical Link (copper, wireless, etc.)

Application
Presentation
Session
Transport
Network
Data Link
Physical

- Protocol packet structure and control commands to manage communication
- Protocol families (suites) a set of cooperating protocols that implement the network stack
- Encapsulation carry higher level info within lower level envelope

Layer 4 Protocol: Transmission Control Protocol (TCP)

- Connection based, ports, reliable, in order, bytestream, congestion control

Universal Datagram Protocol (UDP)

- No connections, datagram, unreliable, out of order
- Use UDP than TCP when:
  - Can't tolerate latency in initiating a connection
  - Old data is useless data (voice and video games), TCP pause data stream in case of error

Types of Faults in Digital Designs

- Design bugs, manufacturing defects, runtime failures
- *** Dependability via Redundancy, redundancy → error detection

Spatial Redundancy replicated data/info/hardware to handle hard and soft failures

Temporal Redundancy retry to handle soft failures

Dependability Measures

- Reliability: Mean Time To Failure (MTTF)
- Service interruption: Mean Time to Repair (MTTR)
- Mean time between failures MTBF = MTTF + MTTR
- Availability = MTTF / (MTTF + MTTR)
- Improving availability
  - Increase MTTF (more reliable hw/sf + fault tolerance)
  - Decrease MTTR (improve tools for diagnosis and repair)
- Annualized Failure rate (AFR) = (#Hw * 8760 hr/yr) / MTTF = 1 / MTTF

- 1 nine: 90% => 36 days of repair/year
  - Eduroam Reliability?
- 2 nines: 99% => 3.6 days of repair/year
- 3 nines: 99.9% => 526 minutes of repair/year
- 4 nines: 99.99% => 53 minutes of repair/year
- 5 nines: 99.999% => 5 minutes of repair/year

Error Detection/Correction Codes (EDC/ECC)

Parity Bit → error detection coding

- Non-zero parity check indicates error occurred
- Even # of errors on different bits not detected
- Odd # of errors are detected

- X = 0101 0101
- 4 ones, *even* parity now
- Write to memory: 0101 0101 0 to keep parity even
- Y = 0110 0111
- 5 ones, *odd* parity now
- Write to memory: 0110 0111 1 to make parity even

- Read X from memory 0101 0101 0
- 4 ones => even parity, so no error
- Read X from memory 1101 0101 0
- 5 ones => odd parity, so error

Hamming ECC

Single Error Detection (SEC) - min hamming dist = 3

- Use more parity bits to pinpoint bit(s) in error, so they can be corrected.
- Example: Single error correction (SEC) on 4-bit data
  - use 3 parity bits, with 4-data bits results in 7-bit code word
  - 3 parity bits sufficient to identify any one of 7 code word bits
  - overlap the assignment of parity bits so that a single error in the 7-bit word can be corrected
- **Procedure**: group parity bits so they correspond to subsets of the 7 bits:
  - p_1 protects bits 1,3,5,7
  - p_2 protects bits 2,3,6,7
  - p_3 protects bits 4,5,6,7

erkeley EECS

1	2	3	4	5	6	7
p_1	p_2	d_1	p_3	d_2	d_3	d_4

Note: number bits from left to right.

Bit position number	
$001 = 1_{10}$	p_1
$011 = 3_{10}$	
$101 = 5_{10}$	
$111 = 7_{10}$	
$010 = 2_{10}$	p_2
$011 = 3_{10}$	
$110 = 6_{10}$	
$111 = 7_{10}$	
$100 = 4_{10}$	p_3
$101 = 5_{10}$	
$110 = 6_{10}$	
$111 = 7_{10}$	

1	2	3	4	5	6	7
p_1	p_2	d_1	p_3	d_2	d_3	d_4

- Note: parity bits occupy power-of-two bit positions in code-word.
- On writing to memory:
  - parity bits are assigned to force even parity over their respective groups.
- On reading from memory:
  - check bits (c_3,c_2,c_1) are generated by finding the parity of the group and its parity bit. If an error occurred in a group, the corresponding check bit will be 1, if no error the check bit will be 0.
  - check bits (c_3,c_2,c_1) form the position of the bit in error.

- Example: $c = c_3c_2c_1 = 101$
  - error in 4,5,6, or 7 (by c_3=1)
  - error in 1,3,5, or 7 (by c_1=1)
  - no error in 2, 3, 6, or 7 (by c_2=0)
- Therefore error must be in bit 5.
- *Note the check bits point to 5*

- By our clever positioning and assignment of parity bits, the check bits always address the position of the error!

- c=000 indicates no error

Double Error Detection (DEC)

- Adding on extra parity bit covering the entire word can provide double error detection

1	2	3	4	5	6	7	8
p_1	p_2	d_1	p_3	d_2	d_3	d_4	p_4

- On reading, the C bits are computed (as usual) plus the parity over the entire word, P:

C=0 P=0, no error
C!=0 P=1, correctable single error
C!=0 P=0, a double error occurred
C=0 P=1, an error occurred in p_4 bit

- Overhead involved in single error correction code:
  - let p be the total number of parity bits and d the number of data bits in a $p + d$ bit word.
  - If p error correction bits are to point to the error bit ($p + d$ cases) plus indicate that no error exists (1 case), we need:

$2^p >= p + d + 1$,
thus $p >= \log(p + d + 1)$
for large d, p approaches $\log(d)$

1.1 How many bits do we need to add to 0011_2 to allow single error correction?

m parity bits can cover bits 1 through $2^m - 1$, of which $2^m - m - 1$ are data bits. Thus, to cover 4 data bits, we need 3 parity bits.

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11
Parity bit coverage	p1	X		X		X		X		X		X		X		X
	p2		X	X			X	X			X	X			X	X
	p4				X	X	X	X					X	X	X	X
	p8								X	X	X	X	X	X	X	X

RAID: Redundant Arrays of Disks

Raid 0: Striping (actually no redundancy)

- Spread data across multiple disks
- Improves bandwidth linearly, doesn't help latency

RAID 1: Disk Mirroring/Shadowing (online sparing)

- Each disk is fully duplicated onto its mirror (high availability)
- Writes go to disk and mirror, read from original disk unless failure
- Most expensive solution, 2x capacity overhead

RAID 3: Single parity Disk

- Disk drives themselves code data and detect failures
- Reconstruction of data can be done with P disk if know which failed

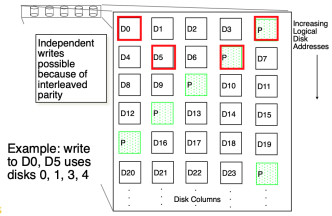
- Writes change data disk and P disk
- P contains parity of other disks per stripe. If disk fails, use P and other disks to find missing info.

RAID 4: High I/O Rate Parity

- Interleave data in data block rather than bit level → more parallelism
- Reconstruction of data can be done with single parity disk
- Reading (w/o) fault involve only data disk
- Writing involves data disk + parity disk (bottleneck for write)

RAID 5: High I/O Rate Interleaved Parity

- Independent writes possible because of interleaved parity

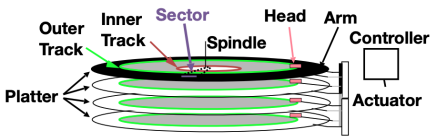


RAID 6: Add another parity block per stripe

- 2 blocks per stripe rather than 1
- Sacrifice capacity for increased redundancy
- Array can tolerate 2 disk failures and continue operating

	Configuration	Pro/Good for	Con/Bad for
RAID 0	Split data across multiple disks	No overhead, fast read / write	Reliability
RAID 1	Mirrored Disks: Extra copy of data	Fast read / write, Fast recovery	High overhead → expensive
RAID 2	Hamming ECC: Bit-level striping, one disk per parity group	Smaller overhead	Redundant check disks
RAID 3	Byte-level striping with single parity disk.	Smallest overhead to check parity	Need to read all disks, even for small reads, to detect errors
RAID 4	Block-level striping with single parity disk.	Higher throughput for small reads	Still slow small writes (A single check disk is a bottleneck)
RAID 5	Block-level striping, parity distributed across disks.	Higher throughput of small writes	The time to repair a disk is so long that another disk might fail in the meantime.

Disk



Disk Access Time = Seek time + rotation time + transfer time + controller overhead
Transfer time = data size / transfer rate
Rotation time = 1/rpm * 6000 * 0.5
Transfer time = transfer rate convert to ms * 0.5

- Seek Time = time to position the head assembly at the proper cylinder
- Rotation Time = time for the disk to rotate to the point where the first sectors of the block to access reach the head
- Transfer Time = time taken by the sectors of the block and any gaps between them to rotate past the head

Flash Memory: Solid State Drive (SSD)

- Bandwidth similar to spinning disk, spinning has better storage density and cost density
- No seek time, no additional latency for random access vs. sequential access of a block
- HDD access time measured in milliseconds, SSD in microseconds

Power Usage Effectiveness (PUE)

PUE = Total power / IT power

Sources speculate Google has over 1 million servers. Assume each of the 1 million servers draw an average of 200W, the PUE is 1.5, and that Google pays an average of 6 cents per kilowatt-hour for datacenter electricity.Estimate Google's annual power bill for its datacenters.

$1.5 \cdot 10^6$ servers $\cdot$ 0.2kW/server $\cdot$ \$0.06/kW-hr $\cdot$ 8760 hrs/yr $\approx$ \$157.68 M/year

Google reduced the PUE of a 50,000-machine datacenter from 1.5 to 1.25 without decreasing the power supplied to the servers. What's the cost savings per year?

$\text{PUE} = \frac{\text{Total building power}}{\text{IT equipment power}} \implies Savings \propto (PUE_{old} - PUE_{new}) * \text{IT equipment power}$

$(1.5 - 1.25) \cdot 50000$ servers $\cdot$ 0.2kW/server $\cdot$ \$0.06/kW-hr $\cdot$ 8760hrs/yr $\approx$ \$1.314 M/year

Cloud

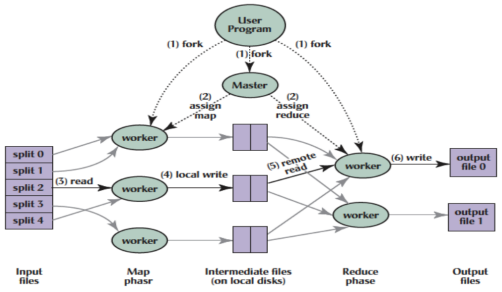
- Shared platform with illusion of isolation
- Attraction of low-cost cycles
- Elastic service

Request-Level Parallelism (RLP)

MapReduce

- Apache Hadoop, Apache Spark

```
file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a + b)
```



4.1 (Summer 2013 Q1) Solve for the maximum controller overhead to meet the following specifications: We need disk latency under 18 ms while reading 800 B of data. The hard drive spins at 6000 rev/min with a seek time of 2.5 ms and transfer rate of 80 KB/s (SI prefix). Include units!

controller overhead $\leq$ disk latency – seek time – rotation time – transfer time
rotation time = 0.5/rpm * (60 sec/1 min) = 0.005 s = 5 ms
transfer time = data size/transfer rate = 0.01 s = 10 ms
controller overhead $\leq$ 18 – 2.5 – 5 – 10 ms = 0.5 ms
max controller overhead → 0.5 ms

4.2 (Fall 2015 QF3) We have a hard drive with a controller overhead of 5 ms. The disk has 12000 cylinders, and it takes 2 ms to cross 1000 cylinders. The drive rotates at 2400 RPM, and we want to copy half a MB of data. Our hard drive has a transfer rate of 500 MB/s. What is the access time of a read from disk?

We want to add up the times that it takes to access this disk. We add them up in the following order: Seek + Rotation + Transfer + Controller. The number of tracks in the seek time is always number of tracks/3, and rotation time is averaged by taking half of the time to rotate around a disk.
$12000/3 * 2/1000 + 1/2400 * 60 * 1000 * 1/2 + 1/2 * 1/500 * 1000 + 5\text{ms} = 15.25$ ms

g) You have an SSD which can transfer data in 32-byte chunks at a rate of 64 MB/second. No transfer can be missed. If we have a 4GHz processor, which takes 200 cycles for a polling operation, what fraction of time does the processor spend polling the SSD drive for data? Leave your answer in the box provided as a percentage.

10%

SHOW YOUR WORK
64 MB/sec / 32 B/poll = 2M polls/sec
2M polls/sec * 200 cycles/poll = 400M cycles/sec
4 GHz clock ⇒ 4000M cycles/sec; 400M c/s / 4000M c/s = 10%

"redundancy through parity." 2, 3, 4, 5
"parity is bit-striped." 2
"a pro is having a small overhead." 0, 2, 3
"parity is byte-striped." 3
"fast small reads are a pro." 0, 1, 4
"parity is block-level striped." 4, 5
"fast small writes are a pro." 0, 1, 5
"disks play a part in increasing reliability." 1, 2, 3, 4, 5
"higher throughput is a pro." 4, 5