Introduction, Perspective, and ISAs

Computer Architecture
ECE 6913

Brandon Reagen

NYU | TANDON SCHOOL OF ENGINEERING

Logistics and Syllabus

- Pre-reqs
  - Digital logic
  - C/Cpp
- Expectations
  - Self motivated, diligent students
  - You can't leave everything until the night before in this class!
- Grading
  - Cpp simulation projects: 55%
    - Single cycle MIPS datapath simulator: 10%
    - Pipelined MIPS datapath simulator: 20%
    - Cache simulator: 15%
    - Branch prediction simulator: 10%
  - Exams: 45%
    - Mid-term: 20%
    - Final: 25%
- Textbooks
  - Nothing required
  - Might find useful:
    - "Computer Organization and Design" – Hennessy and Patterson
    - "Computer Architecture: A Quantitative Approach" – Hennessy and Patterson

Late policy (in business days)
One free day
First day: 10%
After that 5% per day

Plagiarism

- Don’t do it.
- Please don’t do it!
  - You won’t learn anything
  - We will have to have many unpleasant conversations over Zoom
  - You will get a zero on the assignment/exam
    - **Cheat twice automatically fail the course**
- All submitted code is run through MOSS
  - Automatically analyzes code for similarity
    - Changing variable names will no mask cheating.
  - Https://theory.stanford.edu/~aiken/moss/
- Cheating:
  - Project you can work with 1 partner. Cannot share code with any other students
  - Exams are individual, **absolutely** no collaboration or discussion allowed

First time teaching

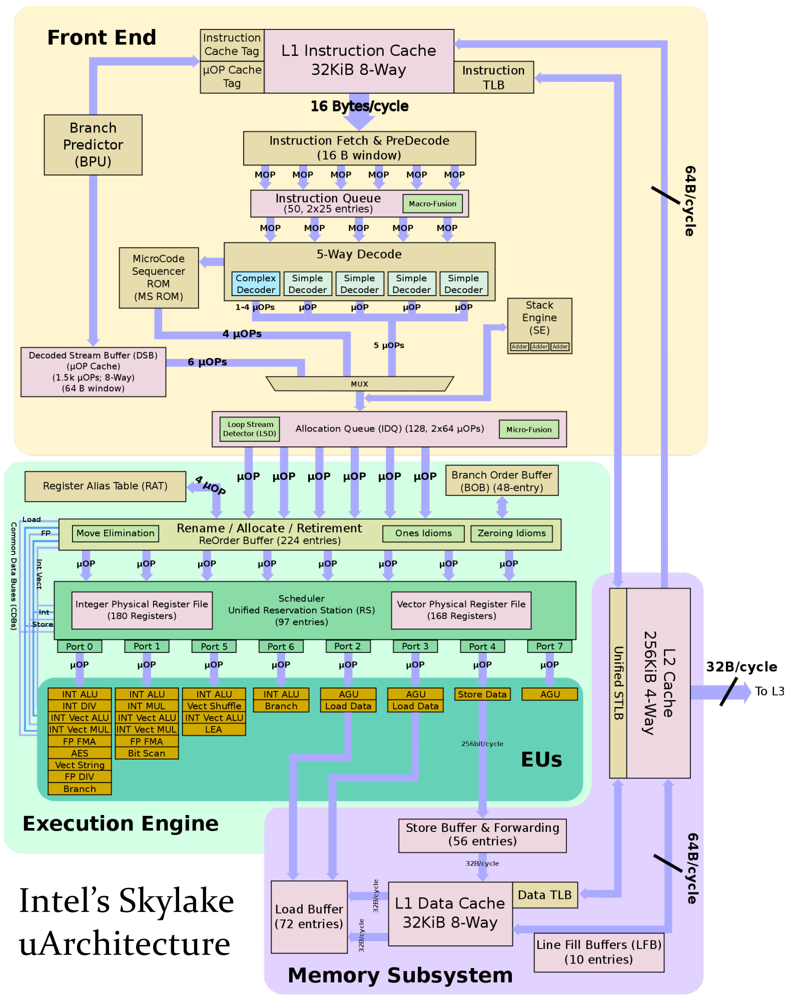
- Feedback welcome!
  - Let me know what works… and what doesn't
  - Please be constructive ☺

- Trying to focus on core material while preparing for practice
  - Hence all the coding projects
  - HW: Google what a Google/FB/MS/Amazon and Intel/AMD interview looks like
  - In a systems PhD program you need to write lots of code

This is not an easy class

- This class requires a lot of time
  - You'll learn how computers work
  - This class covers a lot of material

- Topics covered:
  - ISAs
  - Pipelining
  - OoO execution & Tomasulo's algorithm
  - Caches
  - Virtual memory
  - Branch prediction
  - Main memory
  - Prefectching
  - Bonus:
    - Specialized architectures & SoC design
    - Hardware support for Deep Learning

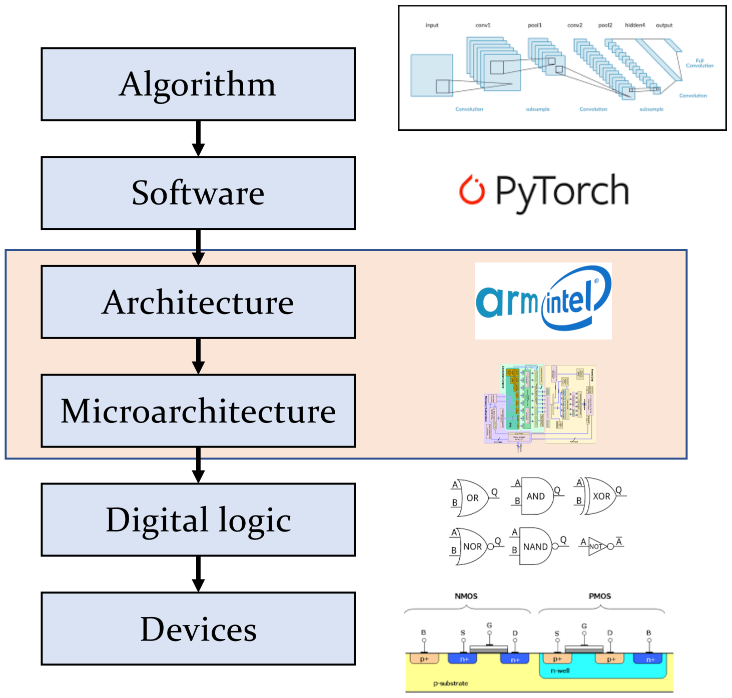


Front End
Instruction Cache Tag
µOP Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
MOP
Instruction Queue (50, 2x25 entries)
Macro-Fusion
MicroCode Sequencer ROM (MS ROM)
5-Way Decode
Complex Decoder
Simple Decoder
1-4 µOPs
µOP
4 µOPs
5 µOPs
Stack Engine (SE)
Adder
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs; 8-Way) (64 B window)
6 µOPs
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
Micro-Fusion
64B/cycle
Register Alias Table (RAT)
4 µOP
Branch Order Buffer (BOB) (48-entry)
Load
FP
Int
Vect
Store
Common Data Buses (CDBs)
Move Elimination
Rename / Allocate / Retirement ReOrder Buffer (224 entries)
Ones Idioms
Zeroing Idioms
Integer Physical Register File (180 Registers)
Scheduler Unified Reservation Station (RS) (97 entries)
Vector Physical Register File (168 Registers)
Port 0
Port 1
Port 5
Port 6
Port 2
Port 3
Port 4
Port 7
INT ALU
INT DIV
INT Vect ALU
INT Vect MUL
FP FMA
AES
Vect String
FP DIV
Branch
INT MUL
Bit Scan
Vect Shuffle
LEA
AGU
Load Data
Store Data
256bit/cycle
EUs
Execution Engine
Unified STLB
L2 Cache 256KiB 4-Way
32B/cycle
To L3
64B/cycle
Store Buffer & Forwarding (56 entries)
32B/cycle
Load Buffer (72 entries)
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB) (10 entries)
Memory Subsystem

Intel's Skylake uArchitecture

NYU TANDON SCHOOL OF ENGINEERING

What is this class/computer architecture?



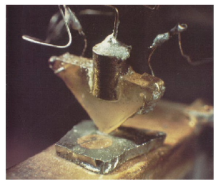
- Architecture (ISA)
  - Specification of what machine will do
- Microarchitecture
  - How specification is implemented
- Examples
  - ISA: x86 and IBM
    - Same code runs decades later
  - Microarchitecture: Pentium -> Skylake
    - Still x86, code will work!

In this lecture

- Brief history lesson
  - Technology
  - Evolution of computer design
  - Scaling laws: Moore and Dennard
  - Current trends

- ISA and instructions
  - Instruction set architectures
  - Design philosophies
  - Instruction design
  - Example: MIPS ISA

First transistor (1947)



Transistors: 1

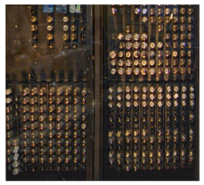
$\longrightarrow$

NVIDIA 2080ti (2018)

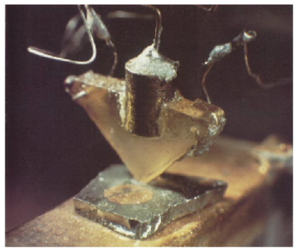
Transistors: 18,600,000,000

Technology/device advancement

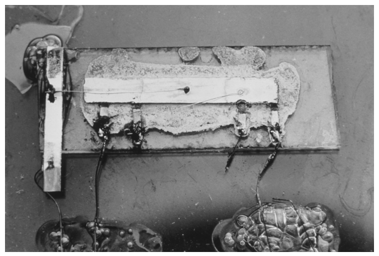
- Vacuum tubes (~1930s)
  - Impressive amounts of engineering and patience
  - Many initial successes (see ENIAC)



- Transistor (1947)
  - First solid-state switch
  - Win: no moving parts



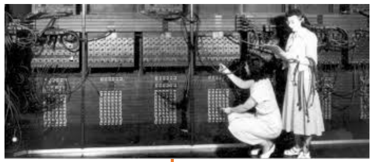
- Integrated circuits (1958)
  - Initial versions of what we use today
  - More and more trying to cram everything on chip



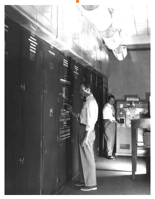
NYU TANDON SCHOOL OF ENGINEERING

Computers over time

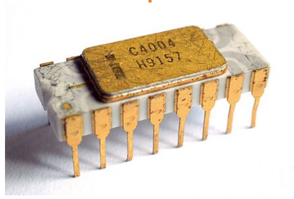
ENIAC
1945



EDVAC
1949

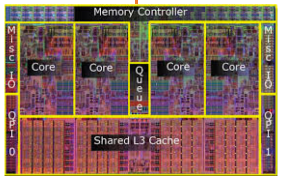


IBM 360
1964



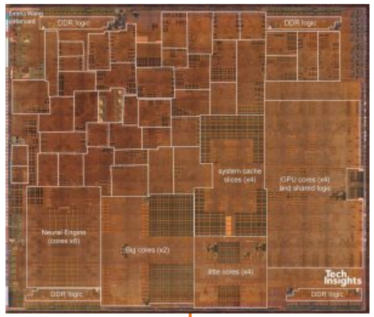
Intel 4004
1971

Pentium Pro
1995
P6 cores!



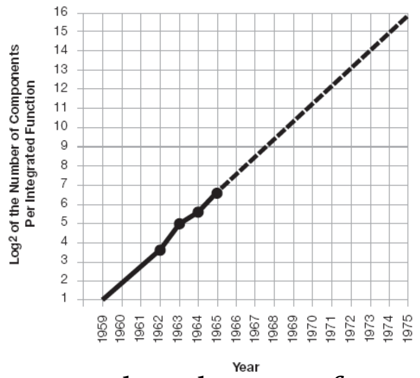
Intel Nahelem
2008

Apple A12 SoC



Performance scaling: Moore's law

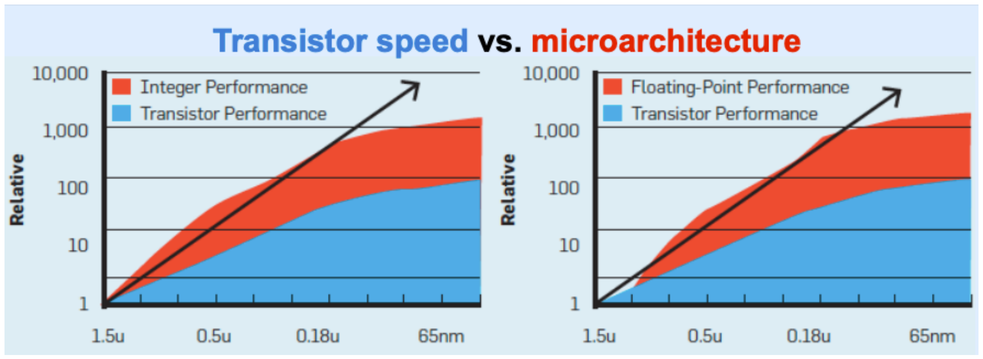
- One thing that made all this possible: device scaling
- In 1965 Moore estimated how many transistors chips would have



What were the takeaways from the paper?

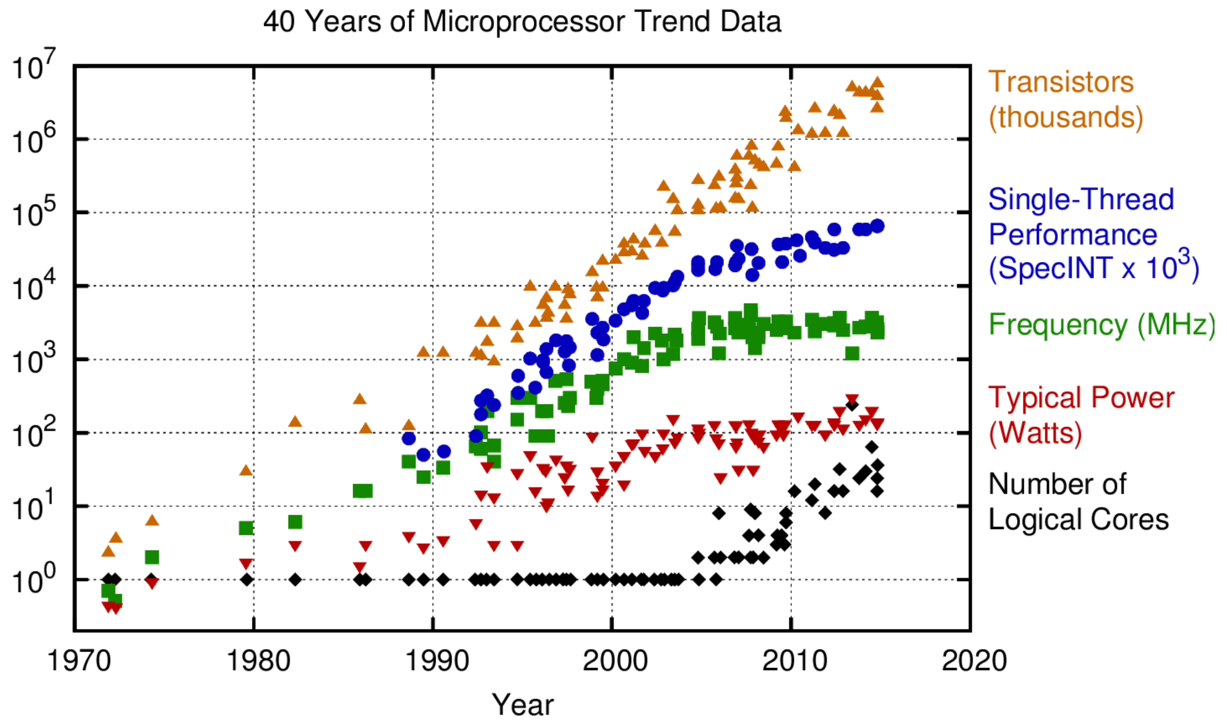
- Less cost per devices
- Right: Nearly followed scaling
- Wrong: Things broke down.

How important is this?



"The future of microprocessors" Shekhar Borkar, 2011.

What’s happening today?



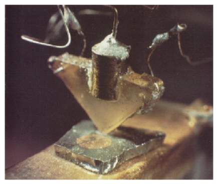
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

In this class

- Brief history lesson
  - Technology
  - Evolution of computer design
  - Scaling laws: Moore and Dennard
  - Current trends

- ISA and instructions
  - Instruction set architectures
  - Design philosophies
  - Instruction design
  - Example: MIPS ISA

First transistor (1947)



Transistors: 1

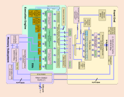
$\longrightarrow$

NVIDIA 2080ti (2018)

Transistors: 18,600,000,000

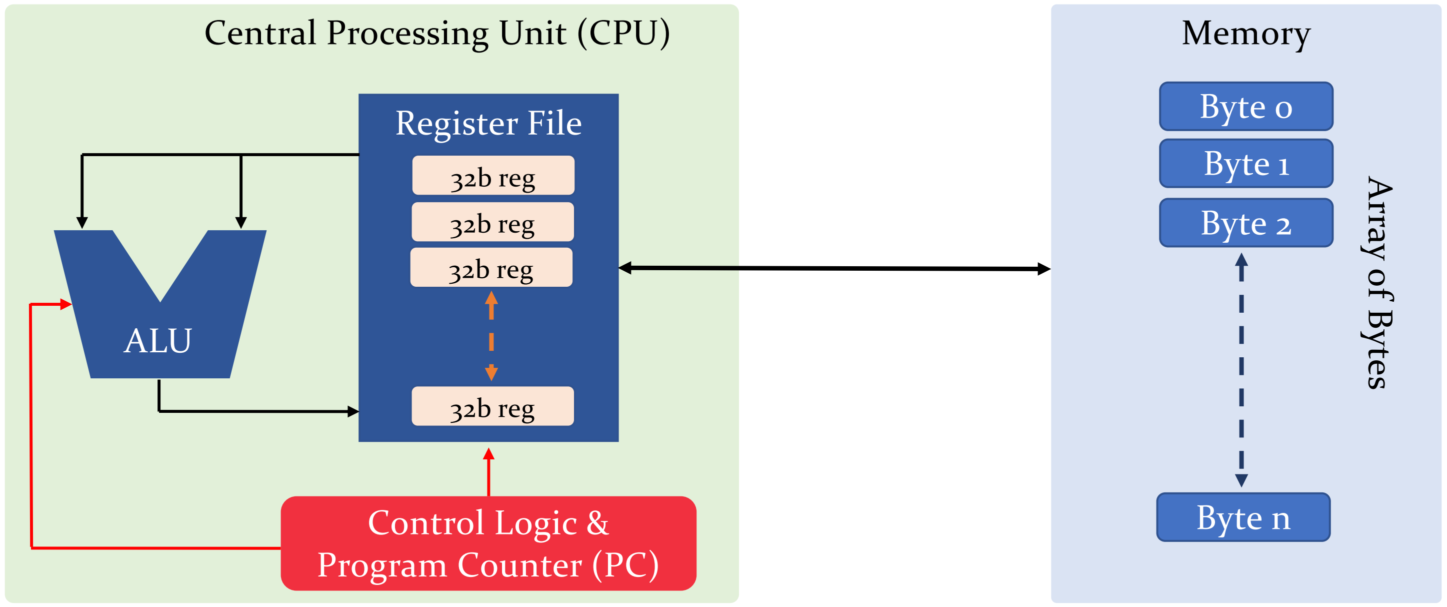
Architecture

Microarchitecture



TANDON SCHOOL OF ENGINEERING

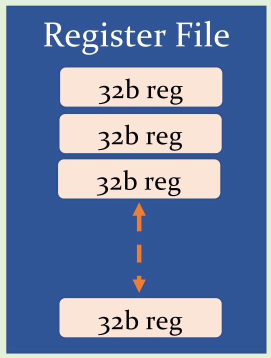
An abstract computer



Register file

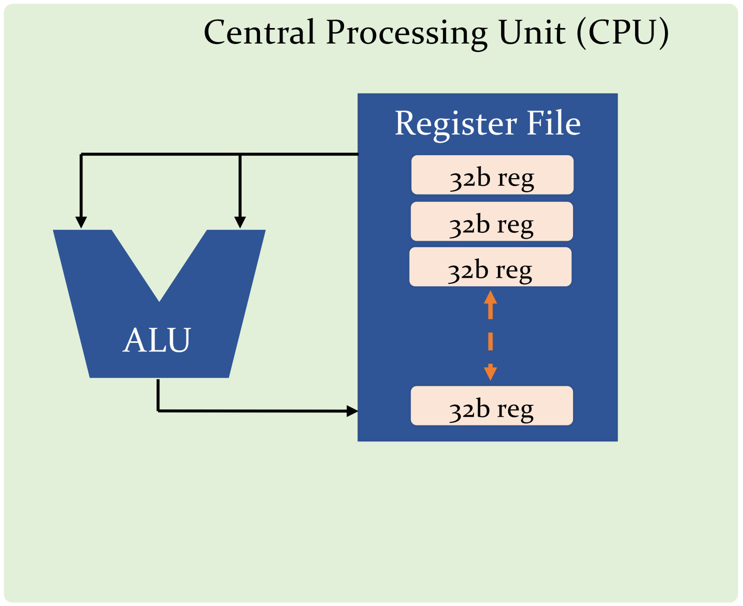
- Register file
  - Array of local registers
  - The base of the machine – where work starts and/or stops
  - Used to feed arithmetic operations and interface with memory
  - Limited number to keep access fast

  - We'll assume we have 32 registers
    - Q: how many address bits do we need?



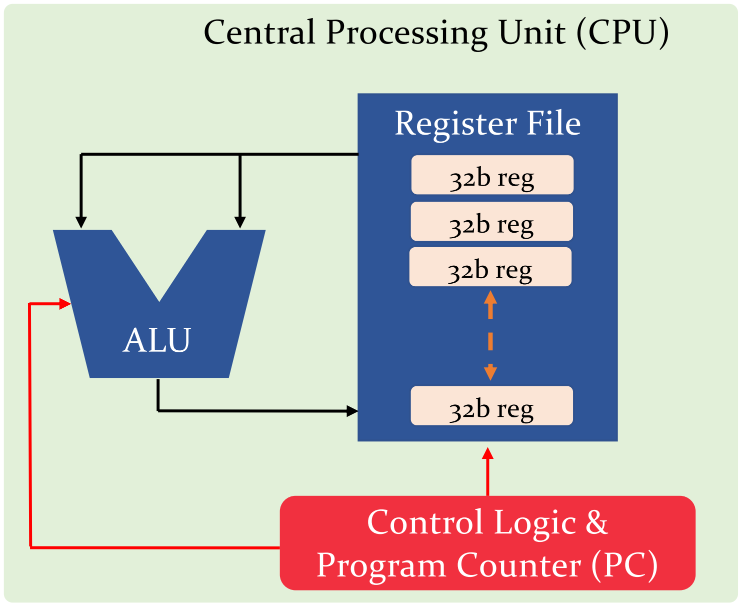
Arithmetic Logic Unit (ALU)

- ALU
  - Core computation block
  - Where operations get processed
    - E.g., ADD, MULT, OR, etc.
  - Interacts directly with the Register File
  - 1-2 inputs Read,
    1 output Written Back
  - Internals mostly what seen in Digital Logic course
    - Not part of this class



Control: Instructions and Program Counter

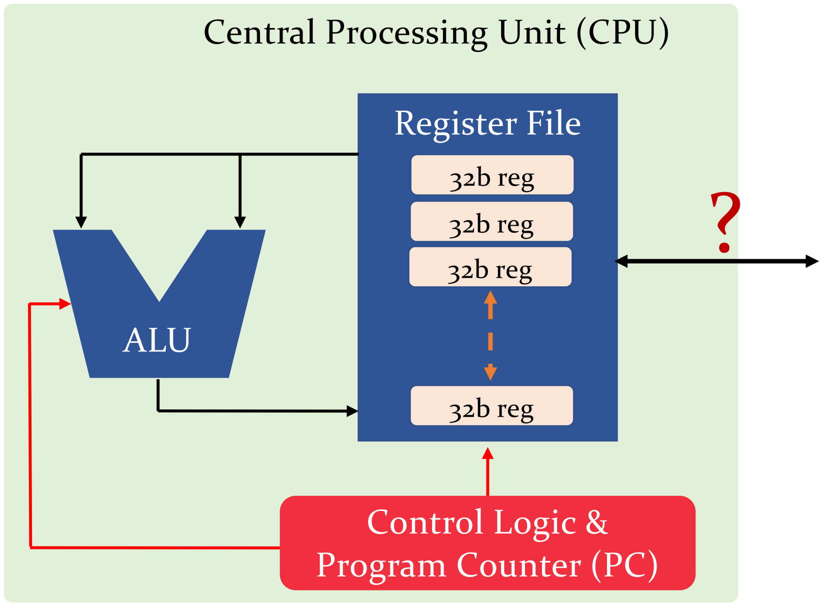
- "Control logic"
  - Programs are instructions
  - Instructions are interpreted by special logic and used to tell ALU and Register File what to do
    - RF: Read reg 1, reg 2
    - ALU: Add reg 1 + reg 2
    - RF: Write back sum to reg 3
- Program counter
  - Tells us where instructions "live"
  - Memory address for next instruction


Central Processing Unit (CPU)
Register File
32b reg
32b reg
32b reg
32b reg
ALU
Control Logic & Program Counter (PC)

NYU TANDON SCHOOL OF ENGINEERING

Control: Instructions and Program Counter

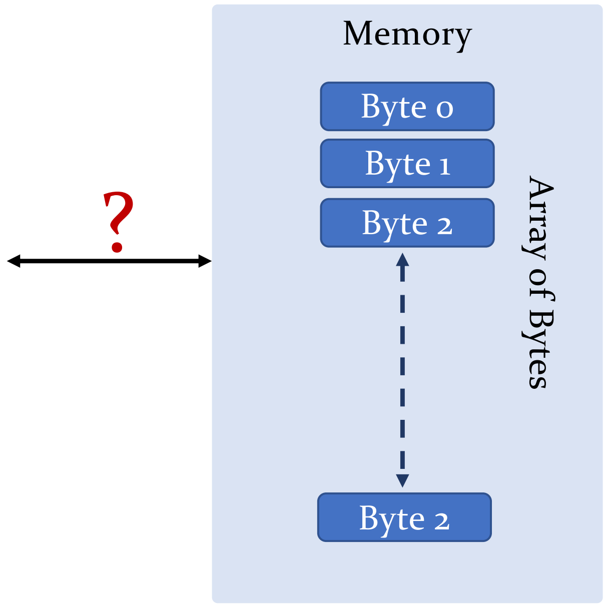
- "Control logic"
  - Programs are instructions
  - Instructions are interpreted by special logic and used to tell ALU and Register File what to do
    - RF: Read reg 1, reg 2
    - ALU: Add reg 1 + reg 2
    - RF: Write back sum to reg 3
- Program counter
  - Tells us where instructions "live"
  - Memory address for next instruction


Central Processing Unit (CPU)
Register File
32b reg
32b reg
32b reg
32b reg
?
ALU
Control Logic & Program Counter (PC)

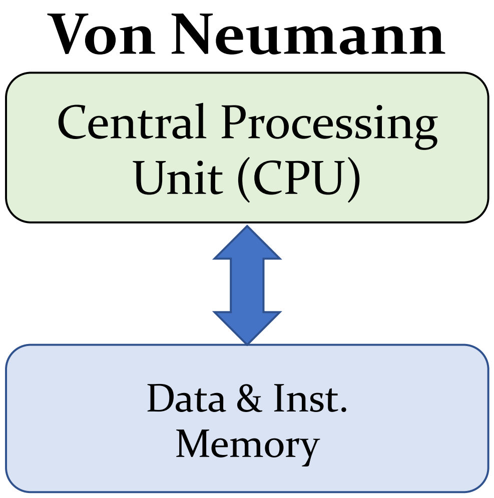
Memory: Big, 1D array of bytes

- Memory is where everything "lives"
  - Data is sent to and from memory via the Register File
  - Special instructions allow us to access memory (more later)

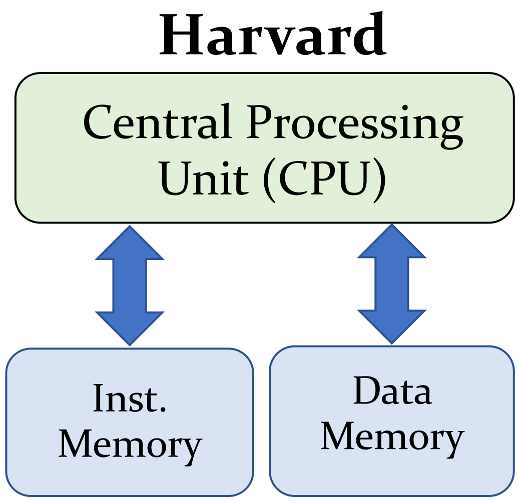
  - Q: Data lives in memory,
    but where do instructions live?



Memory: Where are instructions stored?



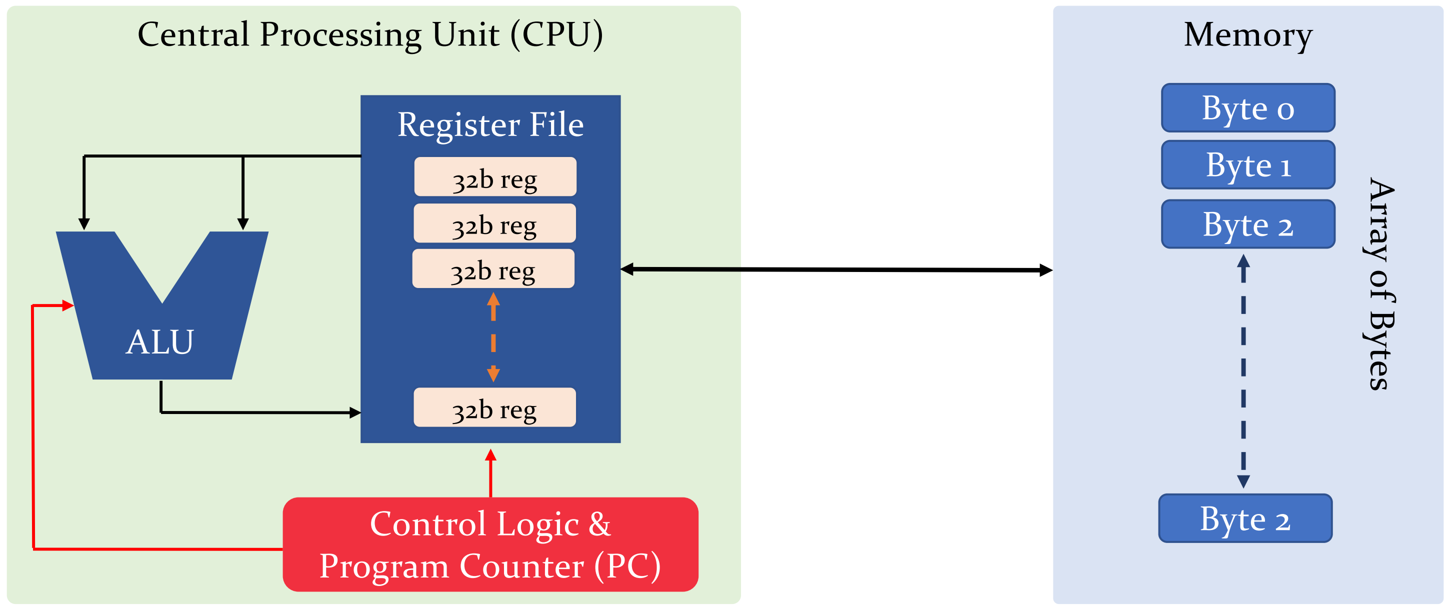
- Memory must be a RAM (random access memory) with read/write capability



- Data memory must be a RAM (random access memory) with read/write capability
- Instruction memory can be a ROM (read only memory)

NYU TANDON SCHOOL OF ENGINEERING

Core components of a processor



Instruction Set Architecture (ISA)

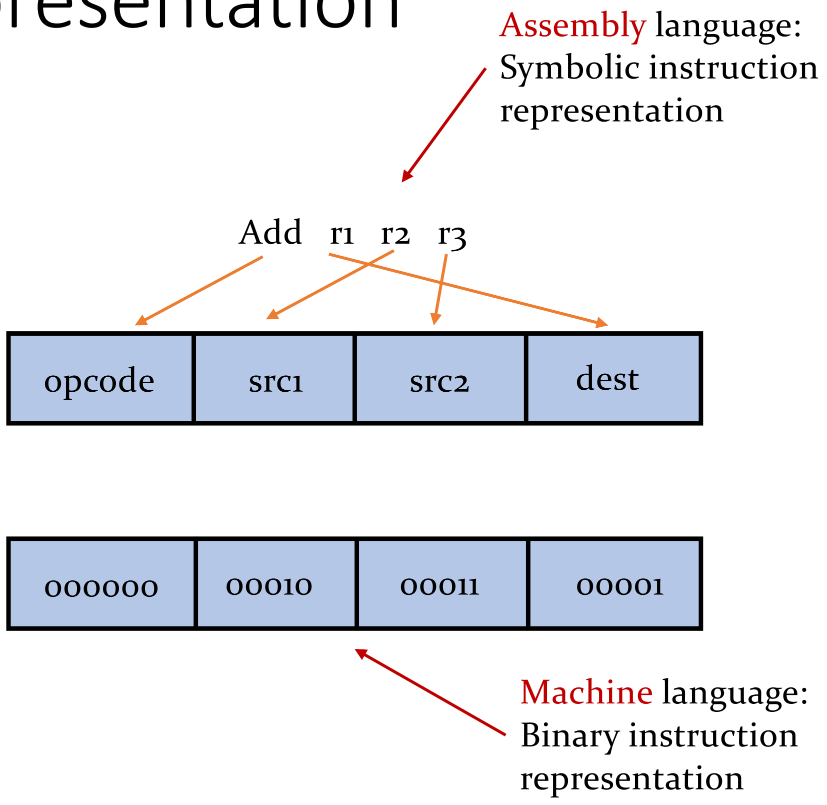
- Contract between the programmer (SW) and machine (HW)
  - ISA defines instructions and their functionality
    - Visible state of the machine, what the programmer/compiler can see
    - Changes to state as instructions are processed
  - Underlying machine implementation faithfully executes instructions
  - Programmers/compiler express functions as series of instructions

- Specification: Instructions and State
  - Exactly how each instruction changes the machine's state
  - Instruction and memory representation

- What does an ISA NOT do?
  - How instructions are implemented
  - Instruction performance or energy usage
  - These are left to the microarchitecture

Instructions: Terms & representation

- Operations
  - What the instruction does
  - Defined using "Op code"
- Operands
  - Input(s) and output identifiers
    - Register File addresses
  - Location and number
- Encoding
  - Everything represented as bits


Assembly language: Symbolic instruction representation
Add r1 r2 r3
opcode
src1
src2
dest
000000
00010
00011
00001
Machine language: Binary instruction representation

Note operand order in assembly verses machine code

Instructions: Operand counts

• None	NOP	Do nothing
• 1 Operand	NOT R1	R1 <= !R1
• 2 Operands	ADD R1, R2	R1 <= R1 + R2
• 3 Operands	ADD R1, R2, R3	R1 <= R2 + R3
• >3 Operands	MADD R3, R1, R2, R3	R4 <= R1 + (R2 * R3)

Instructions: Impact of operand number

Want to do:

A = (B + C) * (B - C)

And we have:

r1 = A   r2 = B   r3 = C

3 Operands

```
add   r1, r2, r3
sub   r4, r2, r3
mult  r1, r4, r1
```

2 Operands

```
mov   r1, r3
add   r1, r2
sub   r2, r3
mult  r1, r2
```

Questions:

Why do we want to limit operand count?

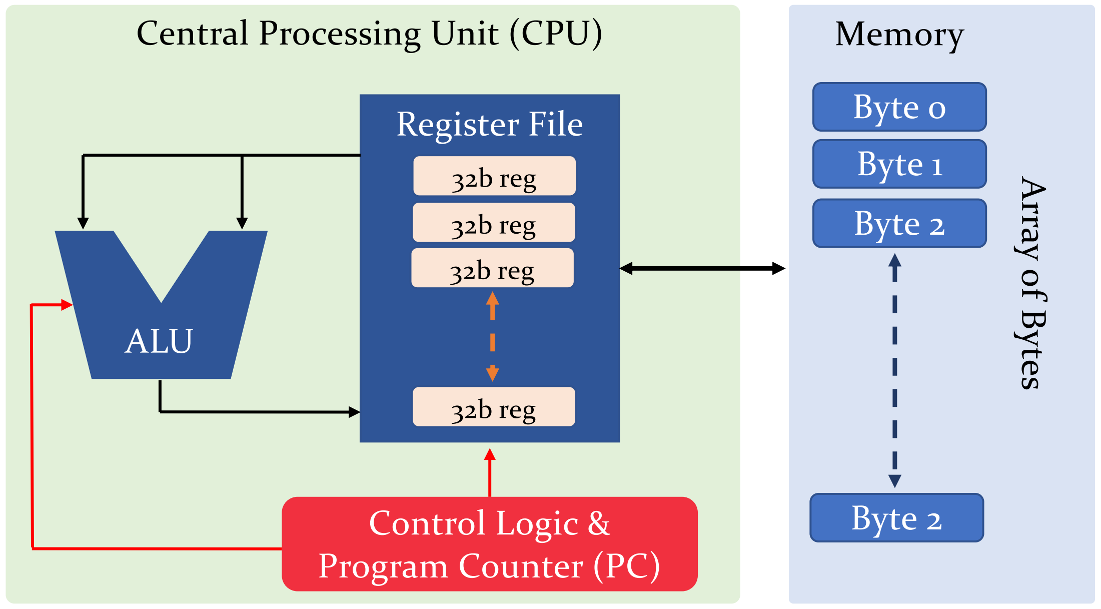
Why would we want to increase it?

NYU TANDON SCHOOL OF ENGINEERING

Machine state defined

Memory
- How addressed
- Data granularity
- Organization (where's the LSB?)
- Memory size
  - Example later
- Advanced features
  - E.g., protection

Registers
- Size & Type
  - PC – Program Counter
  - GPRs – General Program Registers
  - Special
    - E.g., hi/lo multiply
- GPR Count
  - Compiler needs to know how many
  - Physical devices! Not SW variables

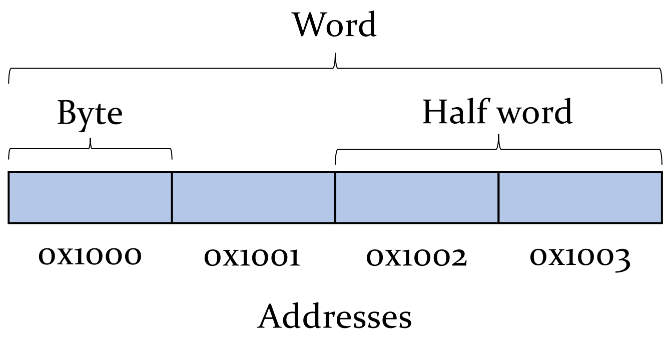


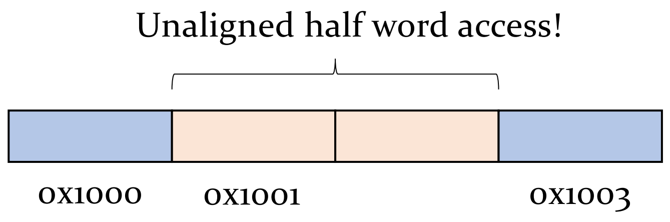
NYU TANDON SCHOOL OF ENGINEERING

Memory: Granularity and addresses

- What does an address mean?
  - It's just a number.. Need to be specific!
  - Smallest granularity of addressable data
    - Word: 32b per address
    - Byte: 8b per address

- Address alignment
  - Memory only addressed at fixed offset
  - If half word aligned, what's a legal address?
    - Divisible by 2
    - What's that mean for the LSB in binary?





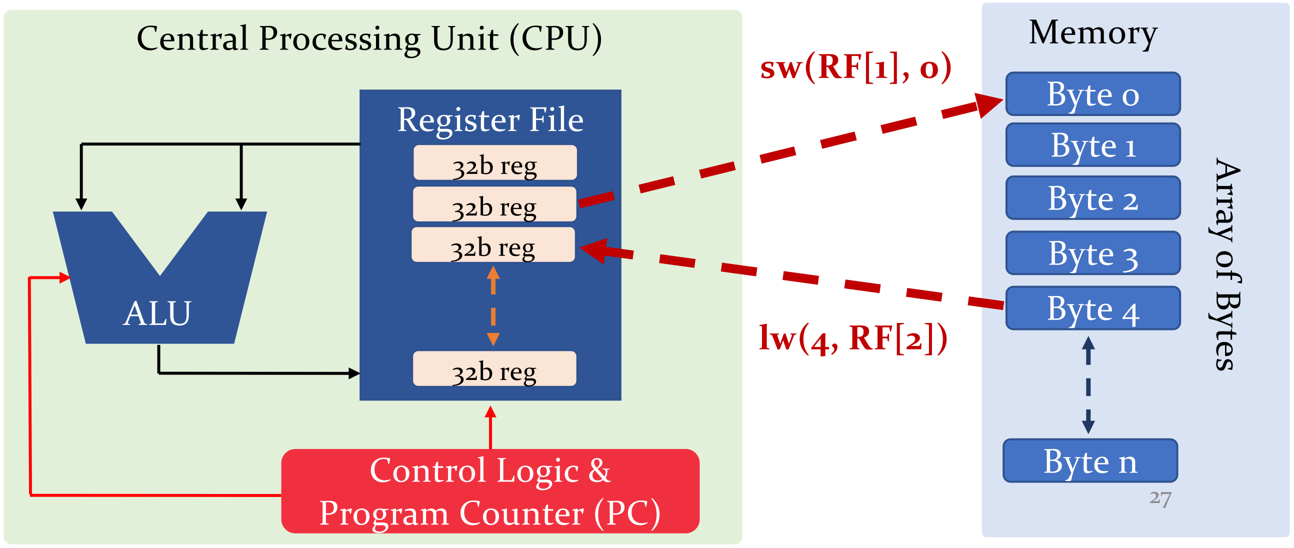
Memory: Accessing operands

Question:
Sw = "store word"
What can we say about legal addresses?
Answer:
Must be divisible by 4

- How do operands get their values?
  - Question of instruction agency:
    - **Load-store**: only explicit memory instructions can interact with memory and registers
    - **Memory-register/memory**: many instructions can directly access memory
      e.g., compute instructions could potentially load values directly from memory

This class will only look at Load-Store

Decouples complexity of hardware

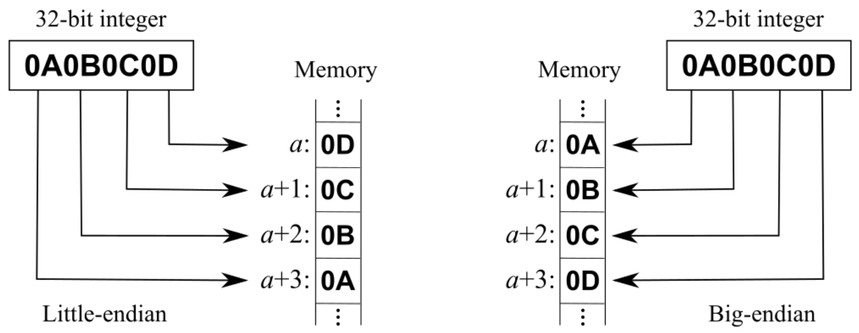


Memory instructions

- Load word (lw)
  - Bring a word at supplied address into core from memory
  - Saved in specified register
- Store word (sw)
  - Send a value at a register out to memory
  - Register and memory location specified in instruction
- Many ISAs have half word and “byte” versions
  - E.g., MIPS, what we’ll use, allows loads and stores of bytes and half words.
- Questions
  - Why do we store words if the program is still running?
  - Why not have more registers?

Memory: Word organization

- Endianness: How Bytes ordered in a word
  - Little Endian (x86)
    - "The most significant Byte is stored at the highest address"
    - Backwards… to me
  - Big Endian (MIPS, ARM)
    - "The most significant Byte is stored at the lowest address"
    - Intuitive… to me
    - Think about reading a word one byte at a time, at address a:



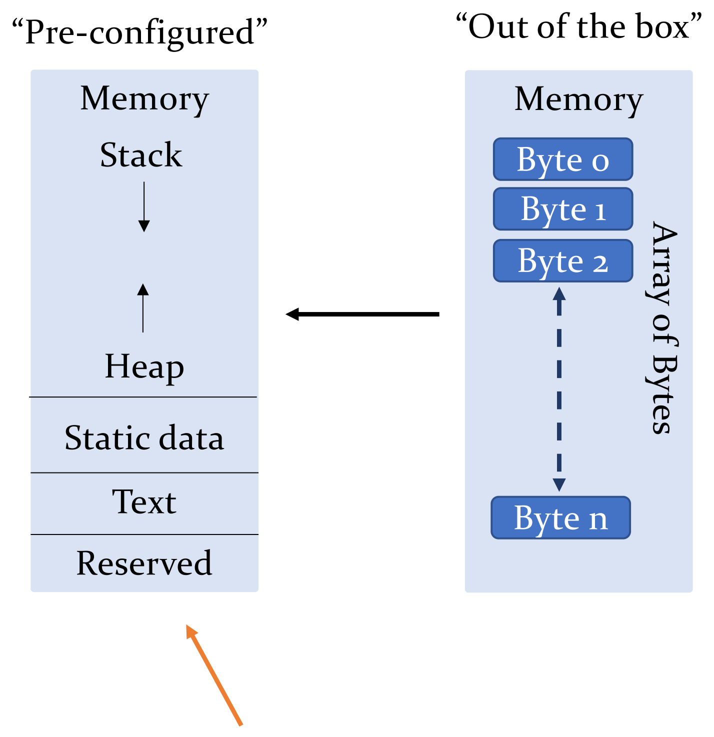
Huge semantics headache..

Memory: How big is memory?

- Word size and memory size
  - Let's assume the ISA uses words with 32b and memory is Byte addressable
  - What's the maximum size memory the computer can have?
    - 4 GB = 2^32 = 2^2 * 2^30 = 4 * (1024<K> * 1024<M> * 1024<G>) = 4 GB
- This is what is meant by 32b architecture
  - Memory addresses, and operands, use 32 bits!
- Today machines have 64b, how much memory can they have?
  - 16 EB!
  - 2^64 = 2^4 * (1024<K> * 1024<M> * 1024<G> * 1024<T> * 1024<P> * 1024<E>) = 2^4 EB = 16 EB

More detailed view of memory

- Divide memory into regions
  - Stack grows "down"
    - Saves registers, function inputs/outputs
  - Heap grows "up"
    - Used for dynamic data structures
  - Static data
    - Vars declared statically (outside functions)
  - Text
    - Stores instructions
    - We're discussing "stored program machines"
  - Reserved
- Special registers tell us where these are
  - Covered later



Harvard or Von Neumann?

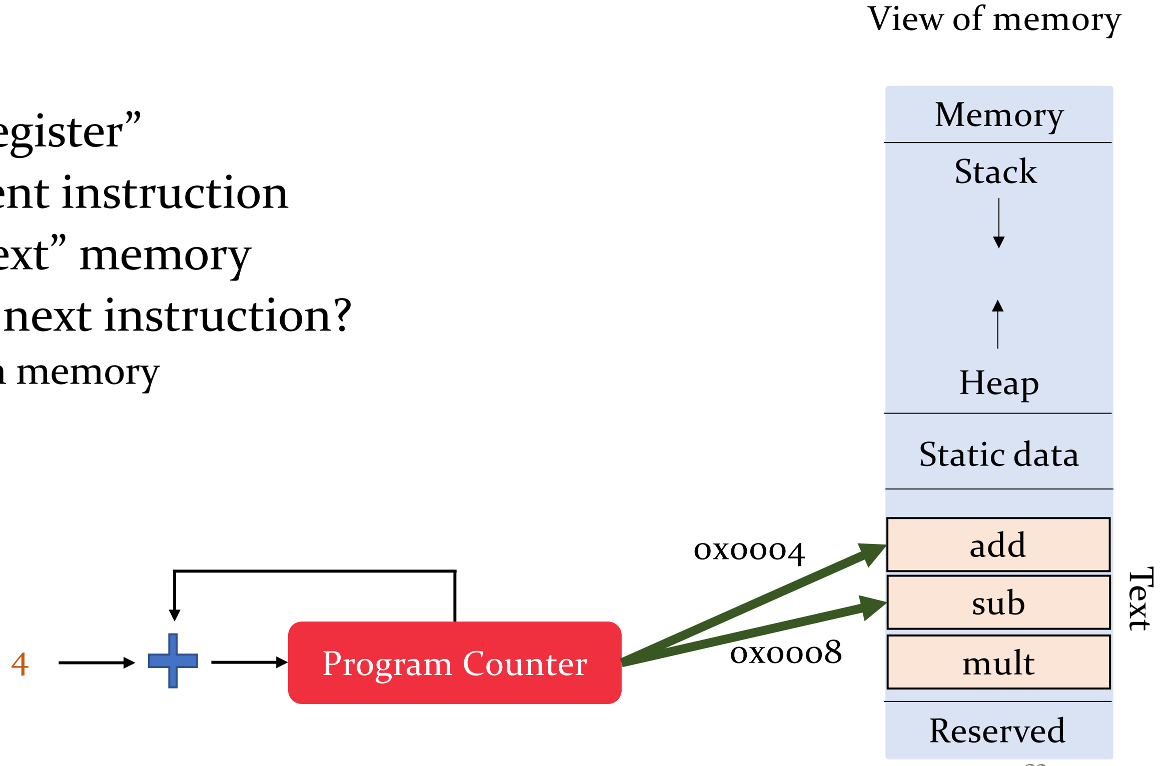
Control: Finding the next instruction

- Program counter
  - "Instruction address register"
  - Stores address of current instruction
  - Programs stored in "Text" memory
  - What happens to find next instruction?
    - Add 4 to PC, load from memory

3 Operands:

A = (B + C) * (B - C)

```
add   r1, r2, r3
sub   r4, r2, r3
mult  r1, r4, r1
```



MIPS ISA Overview

- 32-bit ISA
  - Instruction length: 32 bits
  - Data word length: 32 bits
  - Main memory address: 32 bits

- Load-store architecture
  - 32 GPRs

- *Byte* addressable main memory
  - 32-bit addresses => 4GB of memory

- RISC architecture
  - Reduced instruction set computing
  - Simple hardware, easy to implement
  - Push many complexities to software

Entire ISA in two pages:

https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf

Instruction types: Overview

- Types of MIPS instructions
  - Compute:
    - Arithmetic
    - Logical
  - Memory/Data transfer
    - Load
    - Store
  - Control:
    - Conditional (branch)
    - Unconditional jump

- MIPS has 3 instruction representations
  - Register format
  - Immediate format
  - Jump format

MIPS Instructions: R-Type

opcode	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

opcode:
denotes operation and function

rs & rt:
Source operand registers

rd:
Destination operand register

Shamt:
Shift amount (sll, srl)

funct:
Sub-opcode identifier

Add: 10 0000 = 20_{hex}
Sub: 10 0010 = 22_{hex}
Or: 10 0101 = 25_{hex}
Sll: 00 0000 = 00_{hex}

Note: 000000 Opcode for all R-type
Use “funct” field to specify add/sub…
Slight simplification earlier

MIPS Instructions: R-Type

opcode	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

add	rd, rs, rt	// R[rd] ← R[rs] + R[rt]; signed addition
addu	rd, rs, rt	// R[rd] ← R[rs] + R[rt]; “unsigned” addition, only difference is doesn’t overflow
sub	rd, rs, rs	// R[rd] ← R[rs] - R[rt]; signed subtraction
or	rd, rs, rt	// R[rd] ← R[rs] | R[rt]; bit-wise Boolean OR operation
sll	rd, rt, shamt	// R[rd] ← R[rt] << shamt; logical shift left
mult	rs, rt	// {hi, lo} ← R[rt] * R[rs]; multiply rt, rs; access result with mfli $r1, mflo $r2

Note: *Machine language* register order doesn’t match assembly language!

MIPS Instructions: I-Type

opcode	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

opcode: denotes operation and function

rs: Source operand register

rt: “Target: operand register

immediate: Constant or address

MIPS Instructions: I-Type

opcode	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

addi	rs, rt, imm	// R[rt] ← R[rs] + {SignExtend, imm}; MSB of imm is extended to 32 bits
ori	rs, rt, imm	// R[rt] ← R[rs] | {ZeroExtend, imm}; bit-wise Boolean OR operation
beq	rs, rt, imm	// if{R[rs] == R[rt]} branch to PC + 4 + BranchAddress; ("PC relative")
		// Else go to PC+4
		// BranchAddress = {SignExtend, imm, 00}
lw	rs, rt, imm	// R[rt] ← Mem[{SignExtend, imm} + R[rs]] ("Displaced/based")

NYU TANDON SCHOOL OF ENGINEERING

MIPS Instructions: J-Type

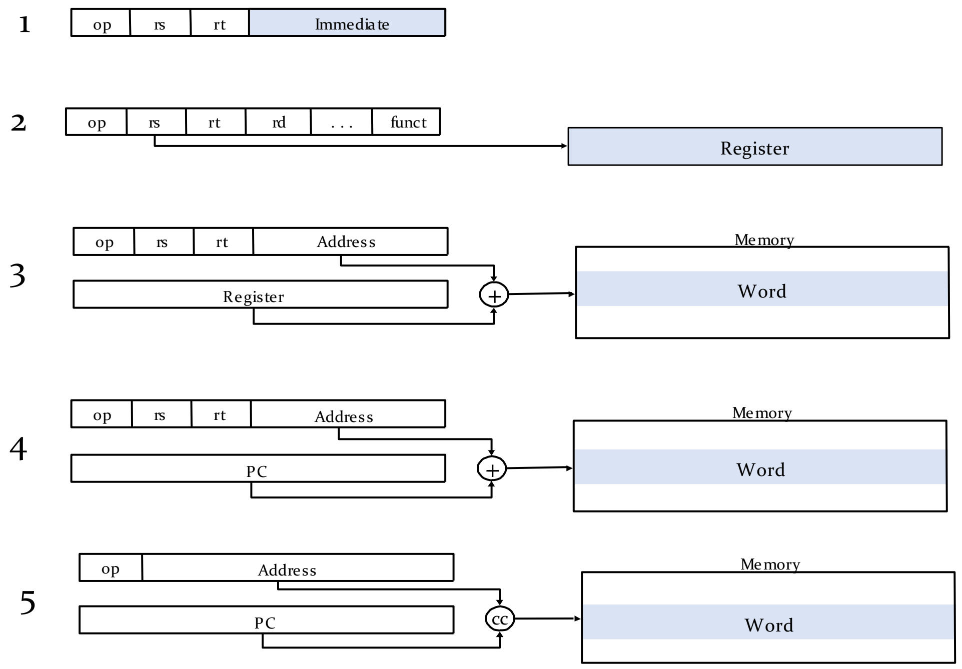
opcode	address
6 bits	26 bits

```
                // JumpAddress = { (PC+4)[31:28], address, 2'b0 }
                // (Pseudodirect)
j      address  // PC  ← JumpAddress
jal    address  // R[31] ← PC+4; PC  ← JumpAddress;
```

Branch instructions verses jump?
Why store PC+4?

Supported addressing modes

1. Immediate
   Addr = Constant
2. Register
   Addr = Register
3. Base/displacement
   Addr = Const + Register
4. PC-relative
   Addr = Const + PC
5. Pseudodirect
   Addr = Const concat PC



NYU TANDON SCHOOL OF ENGINEERING

MIPS Instructions: J-Type

opcode	address
6 bits	26 bits

j   address   // PC ← JumpAddress
jal   address   // R[31] ← PC+4; PC ← Jumpaddress;
                  //JumpAddress = { PC+4[31:28], address, 2'b0 }

Branch instructions verses jump?

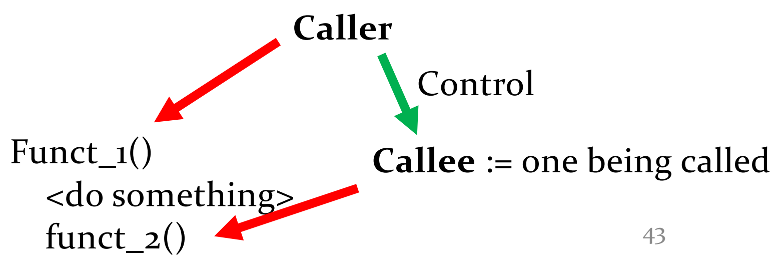
Why 31?
Completely arbitrary?
Who makes these rules..

Why do we need conventions?

- Let’s think about functions
  - What happens when you transfer control?
  - View of registers verses memory

- MIPS calling conventions
  - We will be using a simplified convention
  - The full convention is excessively detailed
    - All core concepts provided here
  - A **great** description can be found here
    - https://sites.cs.ucsb.edu/~kyledewey/cs64w16/documentation/calling_convention/calling_convention.html

- Caller: function calling another function
- Callee: function being called



The MIPS register convention

$zero:	hardwired to zero	(register 0)	
$at:	assembler temporary	(1)	
$v0, $v1:	function return values	(2, 3)	
$a0 - $a3:	function arguments	(4-7)	
$t0 - $t9:	temporaries	(8-15, 24, 25)	[Can be overwritten by callee]
$s0-$s7:	saved	(16-23)	[Callee must save/restore before call]
$gp:	global pointer for static data	(28)	
$sp:	stack pointer	(29)	
$fp:	frame pointer	(30)	
$ra	return address	(31)	

*26-27 are kernel regs, we won't use them.

Supporting functions: Jumps and $ra

- $ra: return address
  - Special register to store next instruction of caller
    - What does that mean? Tells us how to get home!

- Jump instructions help us call functions
  - jal: "jump and link"
    - Will branch to specified location AND store PC+4 in $ra (i.e., R[31])
  - jr: "jump register"
    - Will jump to location specified at register, like $ra!

- We can use jal & jr to get in and out of functions

Supporting functions: arguments

- How do we get arguments to functions?
  - Use reserved registers \$a0 - \$a3
  - To use:
    - Caller loads values into \$a0 - \$a3
    - Caller immediately jumps to function
    - Callee can read \$a0 - \$a3, will have correct values!
    - < do something >
    - Callee jumps back to Caller using: jr \$ra

- Note: when control returns to caller, \$a0-\$a3 may have changed
  - This is the whole point of the convention

Supporting functions: return values

- How do we get values back from function calls?
  - $v0 - $v1 are reserved for returning values from functions
  - In callee, can load $v0-$v1 with values
  - When control returns to caller, can trust return values are there

Supporting functions: temporary registers

- What happens if we need more than \$a* and \$v*?
  - \$t0 - \$t9 can be used
  - "All bets are off" between function calls
    - Callee assumes nothing about initial \$t* values
    - Caller assumes nothing about values of \$t* when control returns

NYU | TANDON SCHOOL OF ENGINEERING

Supporting functions: saved registers

- What if you're in the middle of a computation and need a function?
  - Values in $s* are preserved when control returns
    - Caller uses $s1 then calls a function
    - Callee first saves value of $s* in *memory*
    - Callee does its compute
    - Callee restores $s* values
    - Callee returns control
    - Caller assumes control, value of $s* is same as before function call

Supporting functions: saving registers

- Register values are preserved in memory using the “stack”
- Each function get a unique *stack frame*
  - Recall: $sp register tracks the “top” of the stack (grows?)
  - Functions can write register values to stack, then increment $sp
  - When one calls another, it won’t look “backwards”
    - Can.. But that’s why conventions are important!
  - When functions return control, they reset $sp to where caller left off

- This is super important, why?
  - Think about $ra

Contrived example

Initialize	li $s0, 7 li $s1, 3
Some compute	sub $s2, $s0, $s1
Jump prep	move $a0, $s0 move $a1, $s1
Jump!	jal compute_f1
Get return val	move $s3, $v0
Compute..	slt $s4, $s2, $s3

Q: what happens if a function calls another function..?

compute_f1:

Procedure prep: save all used s* regs	addi $sp, $sp, -12 sw $s0, 8($sp) sw $s1, 4($sp) sw $s2, 0($sp)
Free to compute!	add $s0, $a0, $a1 addi $s1, $a1, 1 sub $s2, $a0, $s1 mult $s2, $s1
Return prep: save return value restore s* regs restore sp!	mflo $v0, $lo lw $s0, 8($sp) lw $s1, 4($sp) lw $s2, 0($sp) addi $sp, $sp, 12
Return control! (Jump)	jr $ra

Visualizing the stack

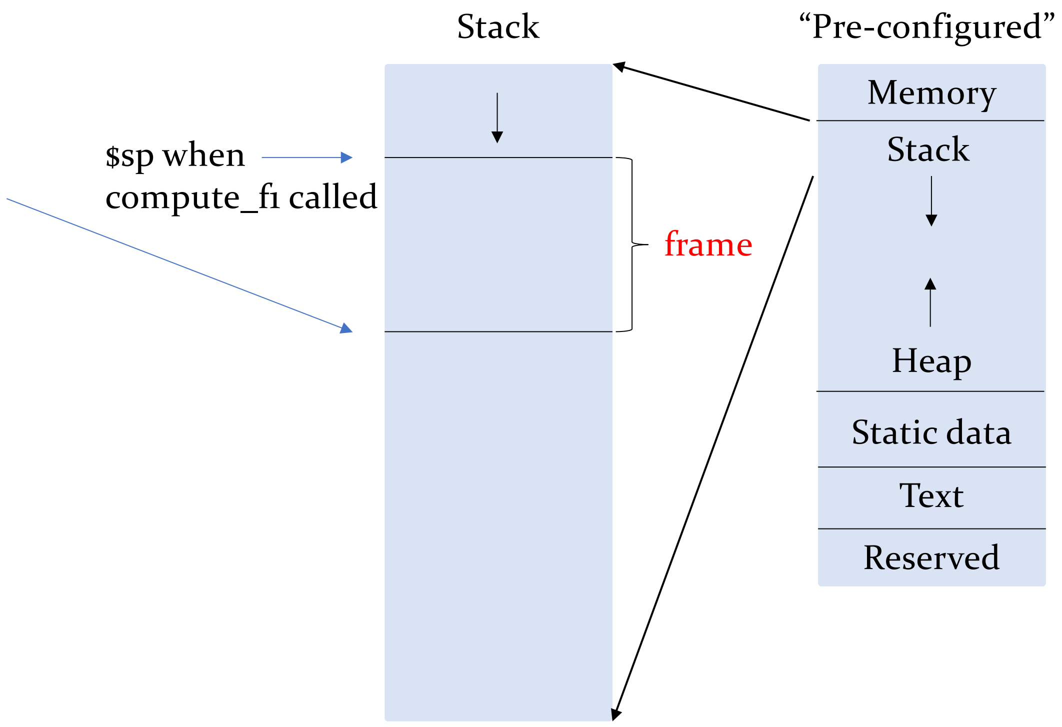
compute_f1:

Setup

```
addi $sp, $sp, -12
sw $s0, 8($sp)
sw $s1, 4($sp)
sw $s2, 0($sp)
.....
```

Teardown

```
lw $s0, 8($sp)
lw $s1, 4($sp)
lw $s2, 0($sp)
addi $sp, $sp, 12
```



Visualizing the stack

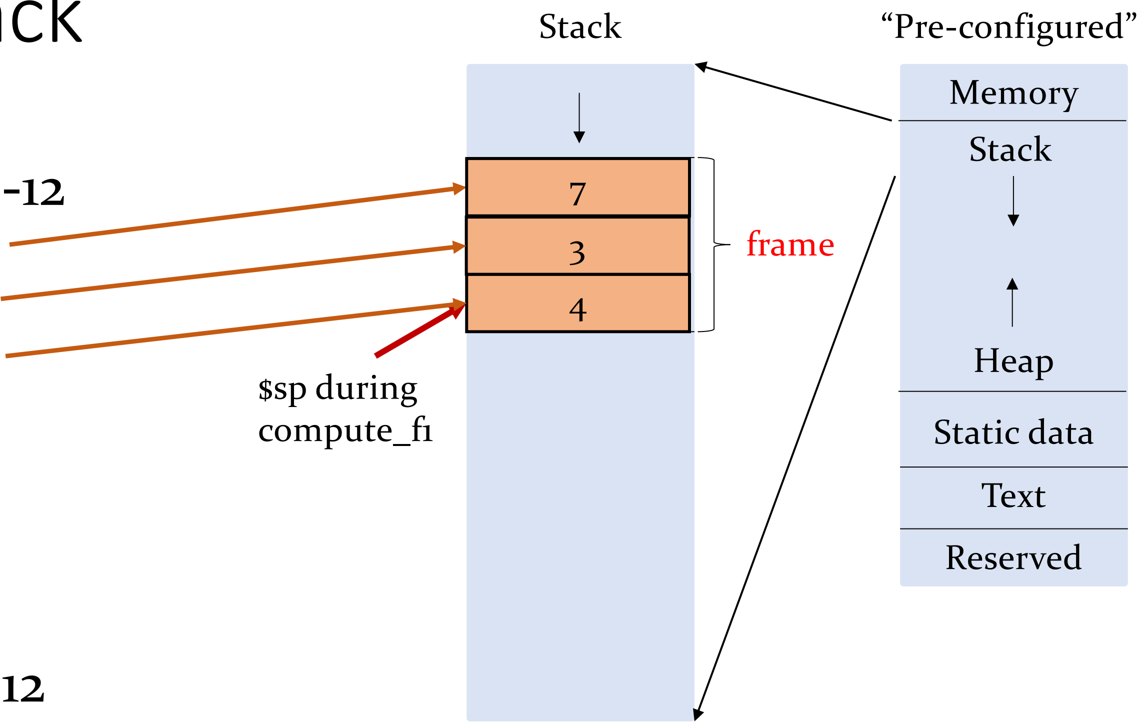
compute_f1:

Setup

```
addi $sp, $sp, -12
sw $s0, 8($sp)
sw $s1, 4($sp)
sw $s2, 0($sp)
.....
```

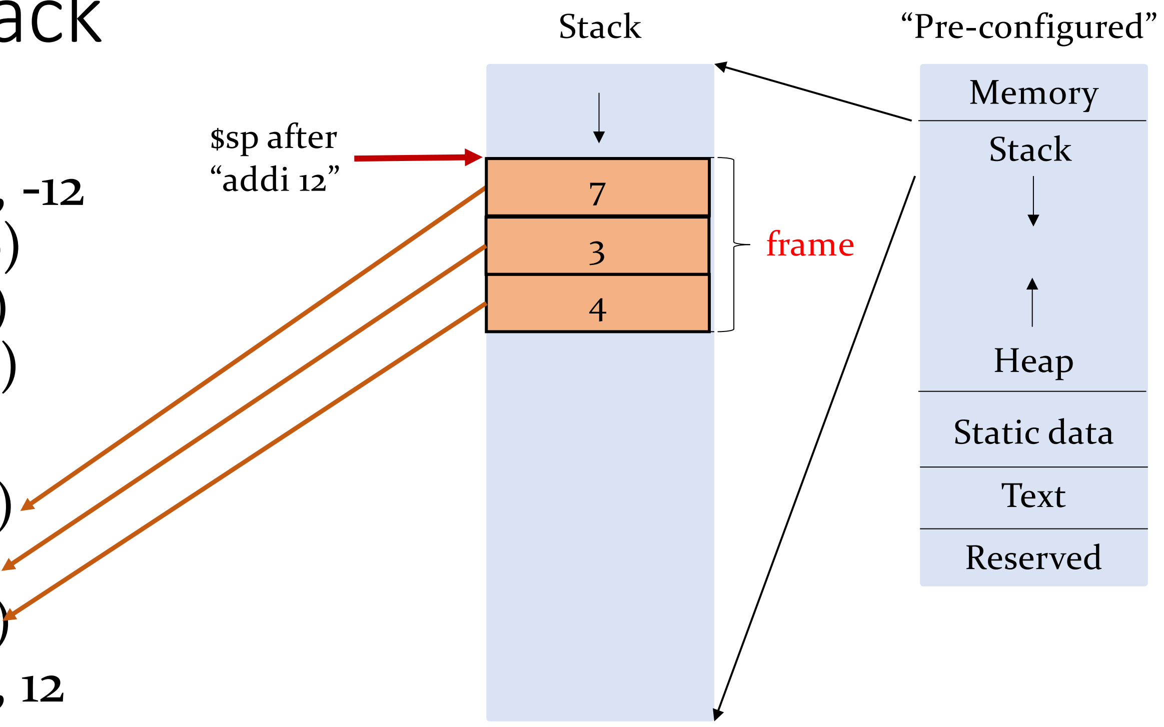
Teardown

```
lw $s0, 8($sp)
lw $s1, 4($sp)
lw $s2, 0($sp)
addi $sp, $sp, 12
```



Visualizing the stack

```
compute_f1:
Setup     addi $sp, $sp, -12
          sw $s0, 8($sp)
          sw $s1, 4($sp)
          sw $s2, 0($sp)
          .....
Teardown  lw $s0, 8($sp)
          lw $s1, 4($sp)
          lw $s2, 0($sp)
          addi $sp, $sp, 12
```



Q: what happens if a function calls another function..?

MIPS Summary

- 32b architecture
  - 32 32-bit GPRs
  - Byte addressable memory
- Load-store architecture
  - All compute use values from registers
  - Only special instructions access memory
- 32 general purpose registers
  - We will assume the simplified calling convention

What makes a good ISA?

- Simplicity favors regularity
  - Instruction size, instruction format, data format
  - Makes hardware implementation cleaner
- Smaller is faster
  - Fewer bit to move, write, and read per instruction
  - Register file is faster than memory
- Make the common case fast
  - Constants tend to be small, immediate field optimized for this
- Good design demands compromise
  - Special formats for important exception
  - E.g., jumping far away (as we saw)

Summary

- That's pretty much it for architecture in this class!

- For the remainder we'll get into microarchitecture
  - How the MIPS ISA can be implemented in hardware
  - Many state-of-the-practice performance optimizations
  - Most of architecture is microarchitectural improvements
  - ISAs rarely change (x86, ARM, MIPS)

- Next time: MIPS single cycle machine and performance metrics