CMPE110 Lecture 04
Instruction Set Architecture

Heiner Litz

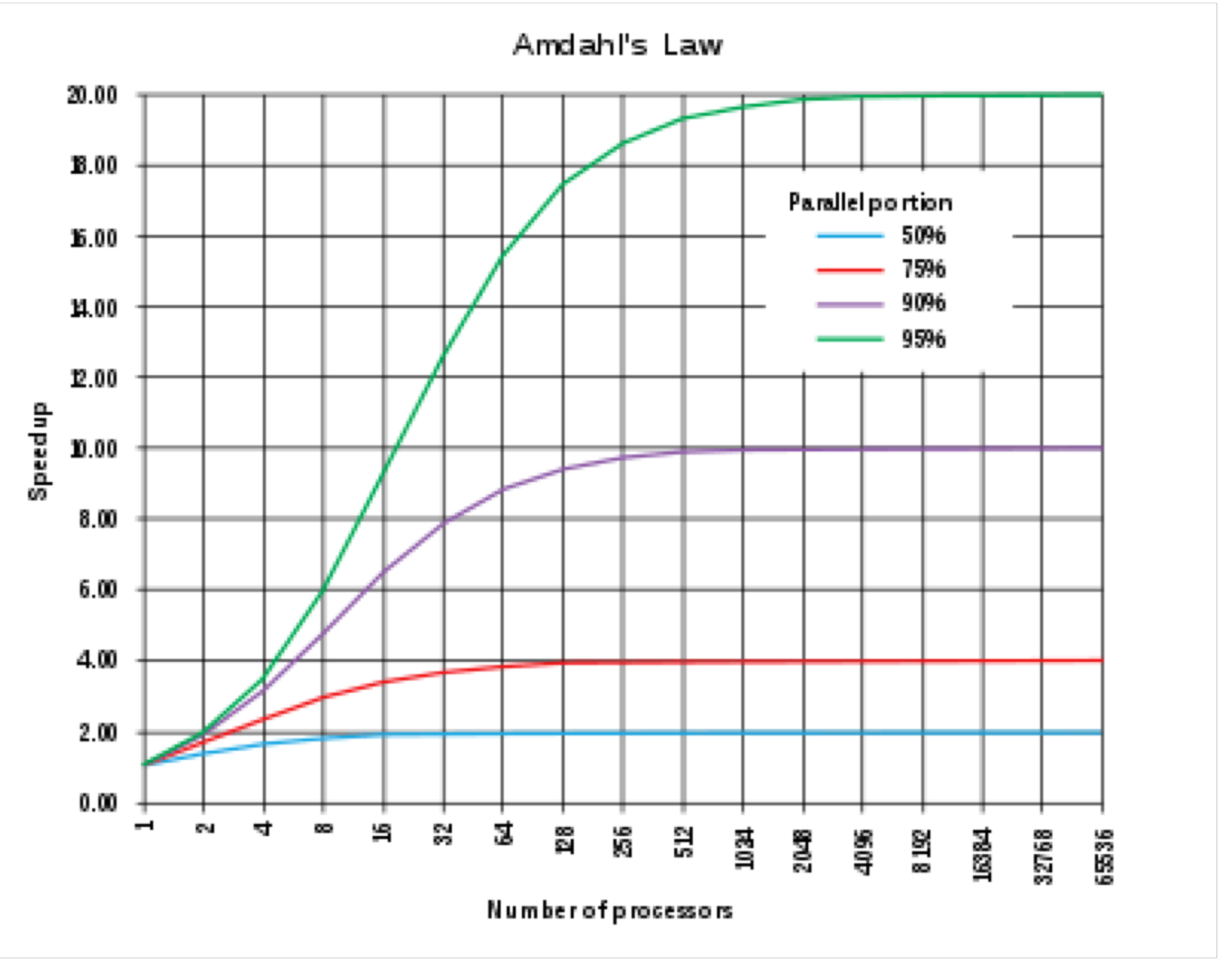
https://canvas.ucsc.edu/courses/19290

Announcements

- Homework groups have been created
- 1st HA Assignment will be released today
- Friday first Quiz

Review

Amdahl's Law



Amdahl's Law
Parallel portion
50%
75%
90%
95%
Speedup
Number of processors
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536

Amdahl's Law: Make Common Case Efficient

- Given an optimization x that accelerates fraction f_x of program by a factor of S_x, how much is the overall speedup?

$$Speedup = \frac{CPUTime_{old}}{CPUTime_{new}} = \frac{CPUTime_{old}}{CPUTime_{old}[(1-f_x)+\frac{f_x}{S_x}]} = \frac{1}{(1-f_x)+\frac{f_x}{S_x}}$$

- Lesson's from Amdahl's law
  - Make **<u>common cases</u>** fast: as $f_x \rightarrow 1$, speedup $\rightarrow S_x$
  - But don't overoptimize common case: as $S_x \rightarrow \infty$, speedup $\rightarrow 1 / (1-f_x)$
    - Speedup is limited by the fraction of the code accelerated
    - Uncommon case will eventually become the common one
- Amdahl's law applies to cost, power consumption, energy …

Other Benchmark Suites

- PARSEC
  - Focus on Multicore/Multithreaded
- STAMP
  - Transactional Memory
- Cloudsuite
  - Datacenter applications
- Microbenchmarks
  - FIO (block I/O)
  - Stream (memory bandwidth)
  - LINPACK (HPC, embarrassingly parallel)

Summarizing Benchmarks

Arithmetic mean $$\frac{1}{n}\sum_{i=1}^{n} T_i$$ Use with times, not with rates

Represents total execution time

Harmonic mean $$\frac{n}{\sum_{i=1}^{n} \frac{1}{R_i}}$$ Use with rates not with times

Geometric mean $$\left(\prod_{i=1}^{n} \frac{T_i}{T_{ri}}\right)^{\frac{1}{n}} = \exp\left(\frac{1}{n}\sum_{i=1}^{n} \log\left(\frac{T_i}{T_{ri}}\right)\right)$$ Good with normalized performance

Does not represent total execution time

- Can also use weighted version
- Be careful which one you use!

Example: SPECpower_ssj2008 for Xeon X5650

- Power/performance benchmark

$$\text{Overall ssj\_ops per Watt} = \left(\sum_{i=0}^{10} \text{ssj\_ops}_i \right) \Big/ \left(\sum_{i=0}^{10} \text{power}_i \right)$$

Target Load %	Performance (ssj_ops)	Average Power (Watts)
100%	865,618	258
90%	786,688	242
80%	698,051	224
70%	607,826	204
60%	521,391	185
50%	436,757	170
40%	345,919	157
30%	262,071	146
20%	176,061	135
10%	86,784	121
0%	0	80
Overall Sum	4,787,166	1,922
Σssj_ops/Σpower =		2,490

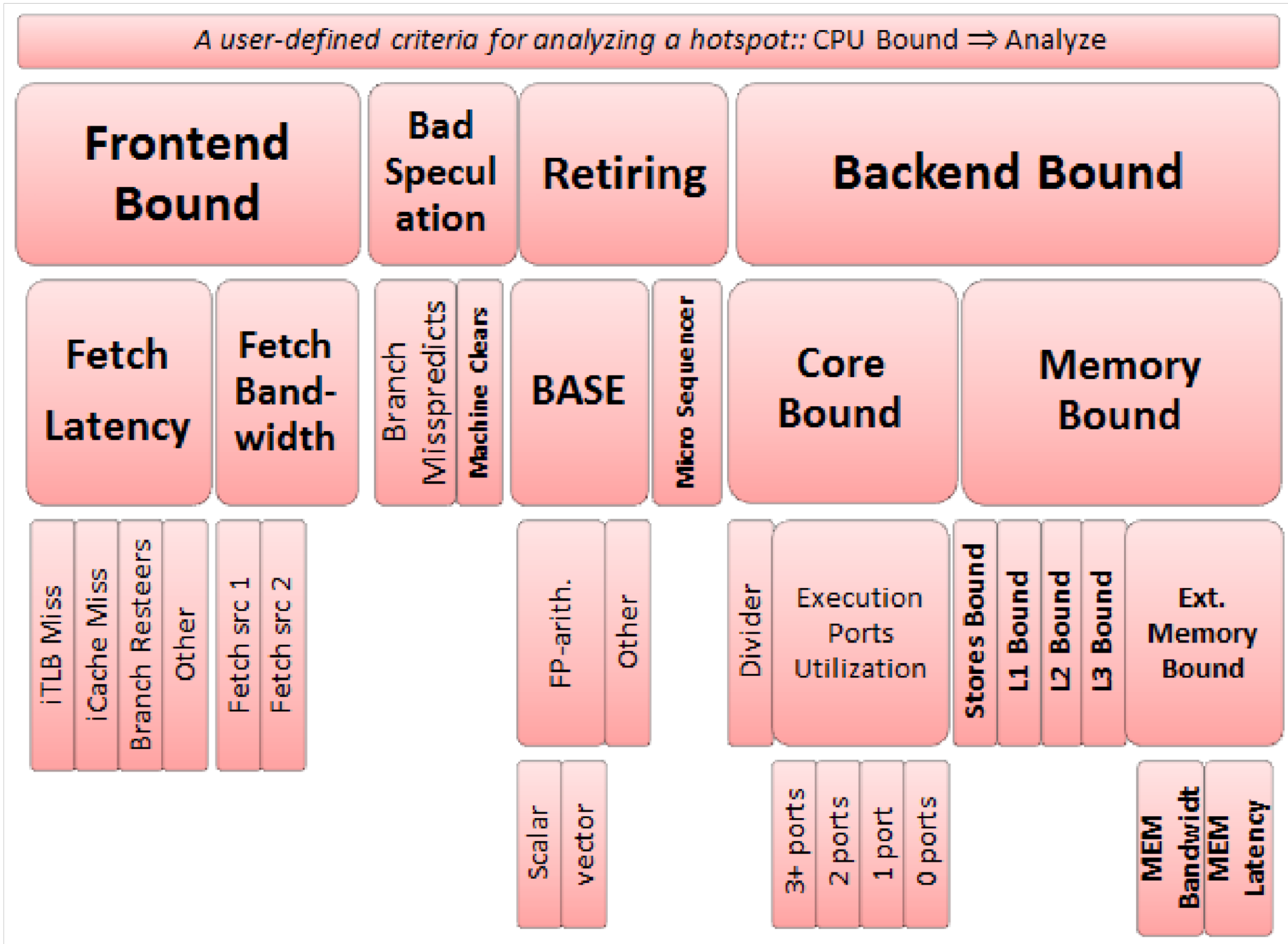
Measuring Performance (Linux)

- ***time** application*
  - Measures execution time of the application
  - Distinguishes between user and kernel
- ***perf record***
  - Low overhead sampling
  - Lists time spent per function
- ***perf topdown***
  - Uses hardware performance counters (PMU)
  - Enables microarchitectural studies

Perf record/report

```
94.72%  perf-test  libm-2.19.so       [.] __sin_sse2
        |
        --- __sin_sse2
           |
           |--44.20%-- foo
           |           |
           |            --100.00%-- main
           |                      __libc_start_main
           |                      _start
           |                      0x0
           |
           |--27.95%-- baz
           |           |
           |           |--51.78%-- bar
           |           |          foo
           |           |          main
           |           |          __libc_start_main
           |           |          _start
           |           |          0x0
           |           |
           |            --48.22%-- foo
           |                      main
           |                      __libc_start_main
           |                      _start
           |                      0x0
           |
```

Perf Topdown



Yasin, Ahmad. "A top-down method for performance analysis and counters architecture."

Microarchitectural Simulation

- Simulators: Gem5, Zsim, Mars, HDL
- Emulation + Timing Model
- Models processors hardware
- Can probe everything -> detailed analysis
- Enables modeling of new techniques

Instruction Set Architectures

Before We Build Hardware

What is the HW/SW interface?

At Their Core, Digital Systems Are Pretty Simple

Computers only work with binary signals (0 or 1 bits)

- More complex stuff expressed as sequences of bits
  - Numbers, characters, strings, pictures, ...
- Memory cells preserve bits over time
  - Flip-flops, registers, SRAM, DRAM
- Logic gates operate on bits (AND, OR, NOT, multiplexor)

To get the HW to compute something

- We express it as a sequence of simple instructions
- We encode these instructions as strings of bits

Big Picture: Running a Program

High Level Language Program

Compiler

Assembly Language Program

Assembler

Machine Language Program

Machine Interpretation

Control Signal Specification

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
ld  $15,  0($2)
ld  $16,  4($2)
sd  $16,  0($2)
sd  $15,  4($2)
```

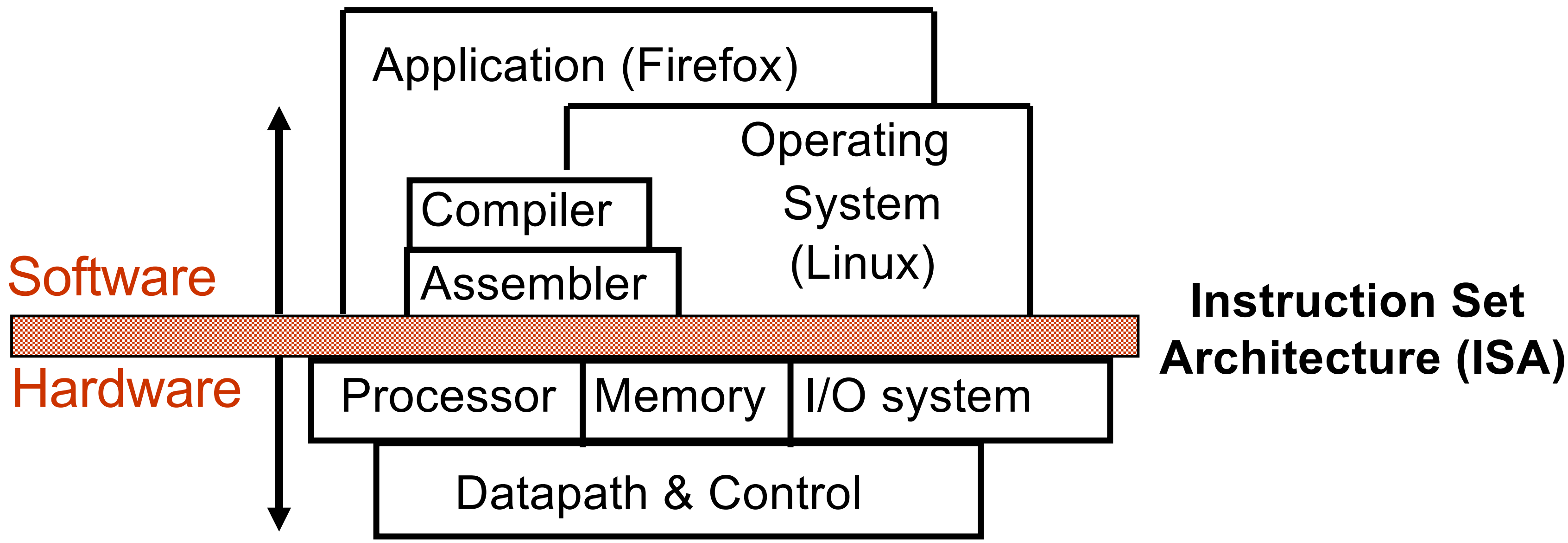
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

High/Low on control lines

Instruction Set Architecture (ISA)

- The HW/SW interface
  - The contract between HW and SW (compilers, assembler, etc)
- It defines
  - The state of the system (Registers, memory)
  - The functionality of each HW instruction
  - The encoding of each HW instruction
- It does ***not*** define
  - How instructions are implemented
  - How fast/slow instructions are
  - How much power instructions consume

Instruction-Set Architecture
Where does it fit?



Does the ISA Choice Matter

- If we write software in high-level languages, ISA choice is irrelevant, right?
  - x86, ARM, …

- Thoughts?

Instruction Set Architecture (ISA)

Many different ISAs

- Examples: Sun SPARC, PowerPC, IBM 390, MIP32, MIPS64, Intel x86 (IA32), Intel IA64 Intel x86-64, ARM A32, ARM A64

Many different chips can implement same ISA (family)

- 8086, 386, 486, Pentium, Pentium II, Pentium4 all implement IA32

ISAs last a long time, implementations are short lived

- x86 has been in use since the 70s
- IBM 390 started as IBM 360 in 60s
- Stable interface for software (binary compatibility)

Can you change an ISA?

RISC vs. CISC
Complex/Reduced Instruction Set

- Hot debate in the 80'ies
- CISC: X86, IBM 360, Motorola 68K
- RISC: MIPS, SUN Sparc, RISC-V
- CISC:
  - MUL Mem2 <= Mem0 * Mem1
- RISC:
  - Load REG0 <= Mem0
  - Load REG1 <= Mem1
  - MUL REG0 <= REG0 * REG1
  - Store Mem2 <= Reg0

RISC vs. CISC

CISC	RISC
Emphasis on Hardware	Emphasis on Software
Multi-cycle complex instructions	Simple (single clock) instructions
Memory-to-Memory load/store incorporated in instr.	Register-to-Register Separate load/store instructions
Small code size	Large code size
High CPI	Low CPI
Low clock frequency	High clock frequency
Variable length instructions	Same length instructions
Complex instruction decode	Simple instruction decode
HW difficult to implement	HW easy to implement

Intel/AMD's take on CISC

- X86 is a CISC ISA
- Internally Intel/AMD CPUs are RISC
- Transform CISC instructions into uOps
- Can use microcode engine
- Small code footprint of CISC, simple RISC instrs

- But: Decoding is still an issue (variable length)

Instruction Length and Format

- Fixed Length
  - Address of next instruction is easy to compute
  - ***Code density***: common instructions as long as rare
- Variable Length
  - Better code density
    - x86 averages 3 bytes (from 1 to 16 per instruction)
    - Common instructions are shorter
    - Less instruction memory to fetch
  - Fetch and decode are more complex
- Compromise: N fixed sizes (e.g. 32,64,128 bit)

Working Example: RISC-V ISA

RISC-V ISA

Developed by UC Berkeley

- Fully Open-source
- Clean slate design
- Typical of RISC ISAs (e.g., ARM, SPARC, MIPS)
- Combines the best of prior RISC ISAs
- Google, Nvidia, Qualcomm, Samsung, NXP, Micron, Marvell

Why RISC-V instead of Intel x86?

- RISC-V is simple, elegant and easy to understand
- Is becoming the 3rd most relevant ISA (after x86 & ARM)
- x86 is ugly and complicated to explain
- x86 is dominant on desktop
- ARM > x86 but still ugly and complicated

ISA Design Principles

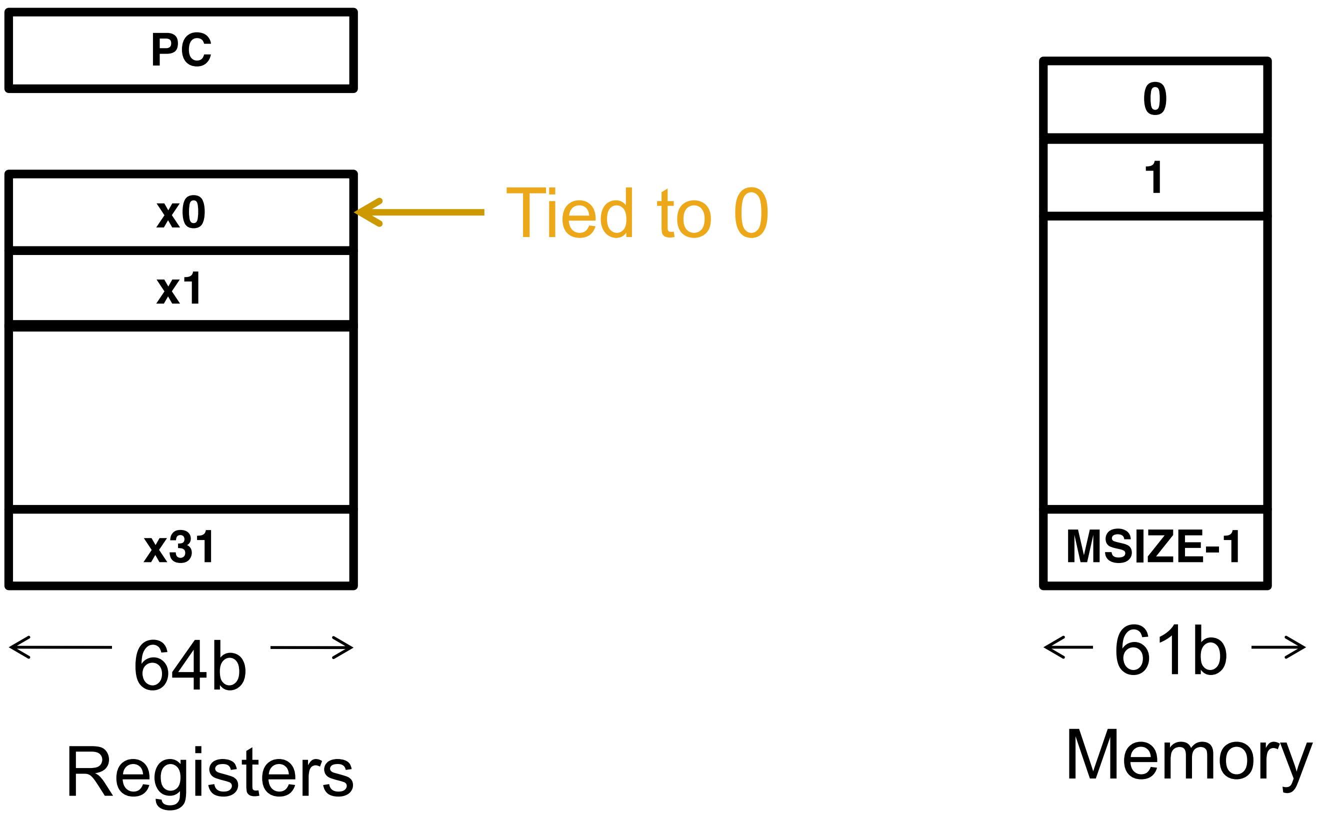
Simplicity favors regularity

Smaller is faster

Good design demands good compromises

Make the common case fast

RISC-V: System State



RISC-V Assembly

- Human readable form

- Example: add x3, x2, x1

opcode

destination operand

source 1 operand

Source 2 operand

Add value of register x2 and x1 and store into x3

RISC-V Instruction Format (machine format)

R-type

funct7	rs2	rs1	funct3	rd	opcode

Opcode: basic operation of instruction (7)

Rd: Register destination operand (5)

Funct3: additional opcode field (3)

Rs1: Register source 1 operand (5)

Rs2: register source 2 operand (5)

Funct7: additional opcode field (7)

Question: Why did RISC-V only define 32 registers?

I-type Question: What is an immediate?

immediate[11:0]	rs1	funct3	rd	opcode

S/B-type Question: Why is the immediate split?

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode

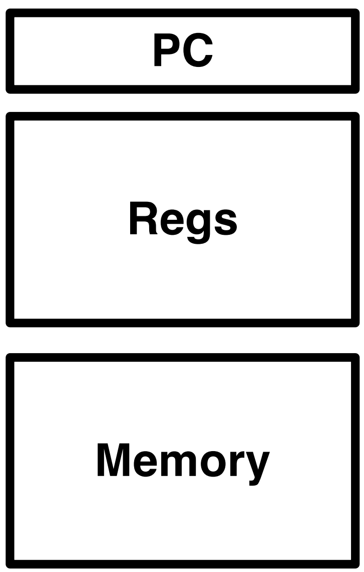
U/J-type

immediate[31:12]	rd	opcode

RISC-V Instruction Format

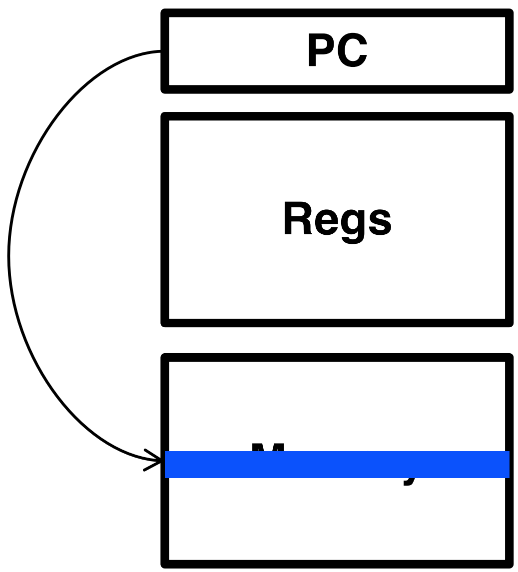
- All instructions are 32 bit to alleviate decoding
  - Smaller is faster
- Requires to interpret bit fields differently for different instructions (R vs. I vs. S/B vs. U/J)
  - Simplicity favors regularity
- Limits register file size to 32 (5 bits per operand)
  - Good designs demand good compromises

Instruction Execution



Before State

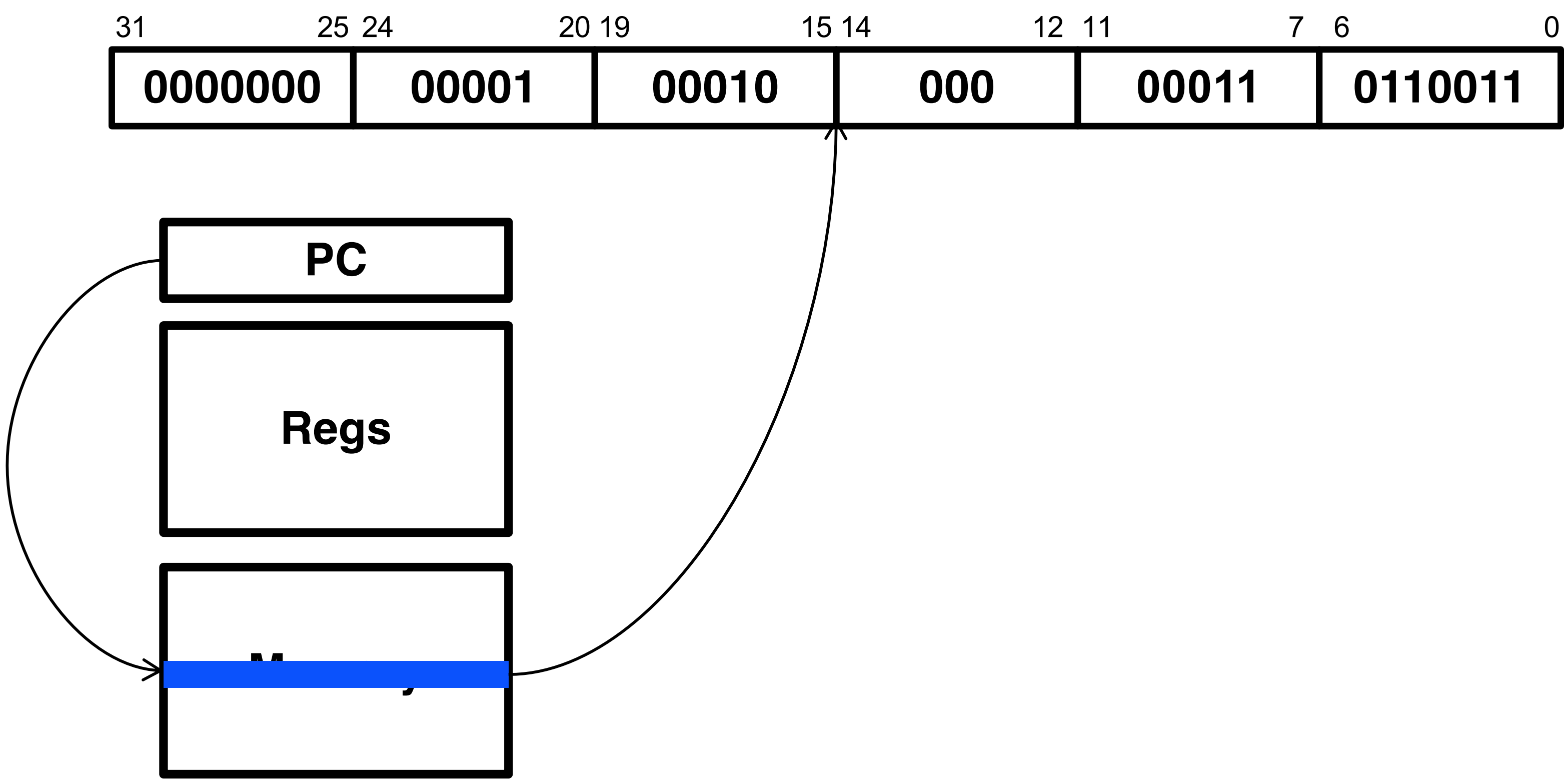
Instruction Execution



Before State

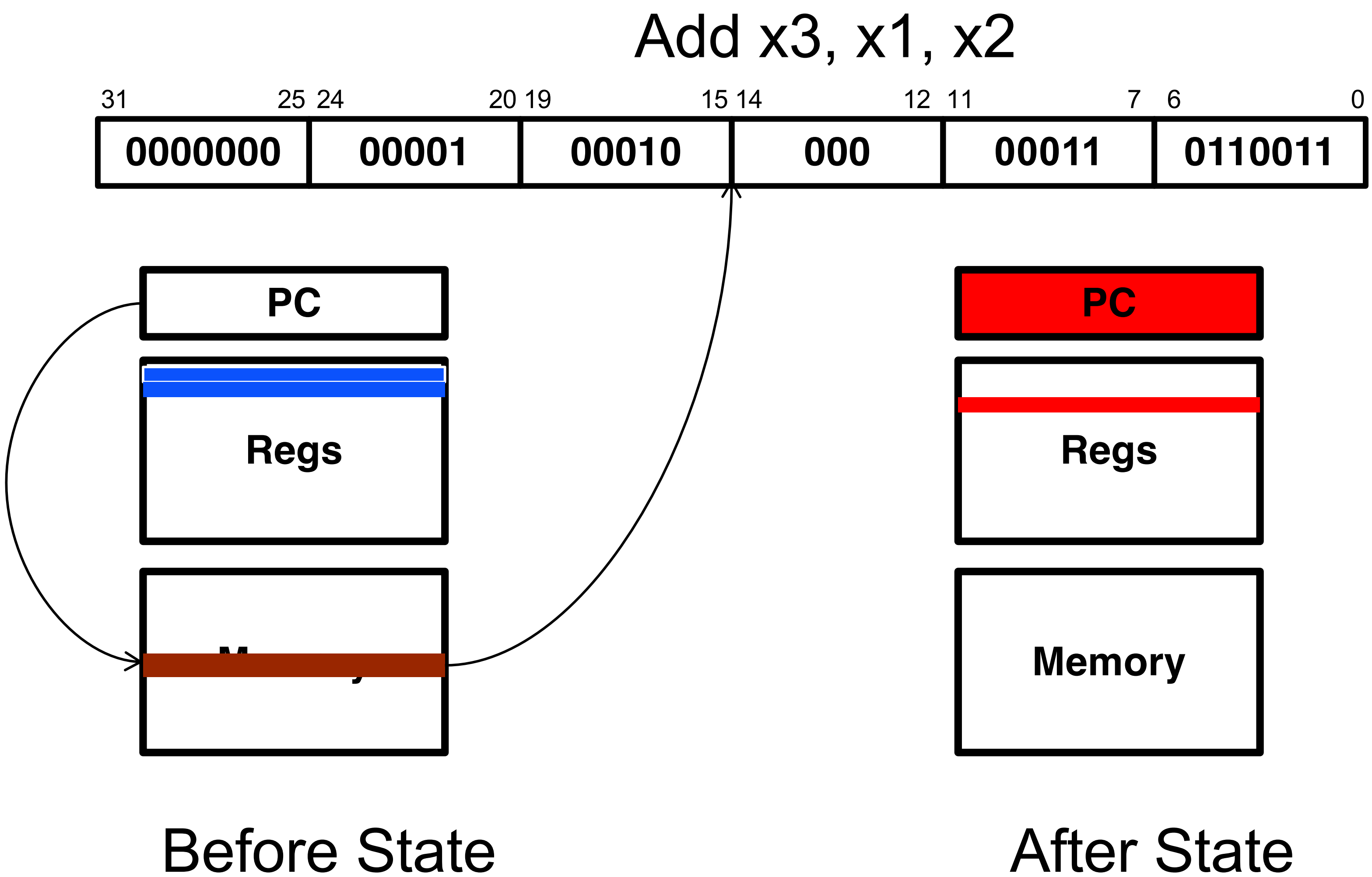
Instruction Execution

Add x3, x1, x2



Before State

Instruction Execution



Instruction Execution

Start with the *before state* of the machine

- PC, Regs, Memory

PC used to *fetch* an *instruction* from memory

Instruction directs how to compute the *after state* of the machine

- For add: PC = PC+4 and rd = rs+rt

This happens ***atomically***

C vs. RISC-V Programmers Interface

	C	RISC-V ISA
Registers		32 64b integer, R0 = 0 32 32b single FP 16 64b double FP PC and special registers
Memory	local variables global variables	2^{61} linear array of bytes
Data types	int, short, char, unsigned, float, double, aggregate data types, pointers	doubleword(64b), word(32b), byte(8b), half-word(16b)
Arithmetic operators	+, -, *, %,++, <, etc.	add, sub, and, sll, etc.
Memory access	a, *a, a[i], a[I][j]	ld, sd, lh, sh, lb, sb
Control	If-else, while, do-while, for, switch, procedure call, return	branches, jumps, jump and link

Why Have Registers?

Alternative: memory-memory ISA?

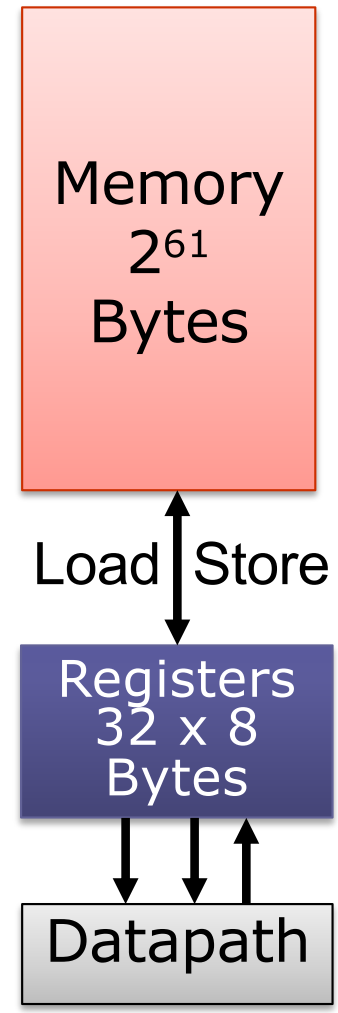
- All HLL(C/Java/..) variables declared in memory
- Instructions operate directly on memory operands?
  - E.g. Digital Equipment Corp (DEC) VAX ISA

Benefits of registers

- Smaller is faster (100-1000x)
- Multiple concurrent accesses
- Shorter names (fewer bits to encode)

Load-Store or RISC ISAs

- Arithmetic instructions only use register operands
- Data loaded into registers, operated on, and stored back to memory



Using Registers

- Registers are a finite resource that needs to be managed
    - By the assembling programmer
    - Or the compiler (register allocation)
- Goal
    - Keep data in registers as much as possible
- Issues
    - Finite number of registers available
        - Spill registers to memory when all registers in use
    - Arrays
        - Data is too large to store in registers
- What’s the impact of fewer or more registers?

Working Code Example (C)

Simple C procedure: sum_pow2 = 2^{b+c}

```c
int sum_pow2(int b, int c)
{
    int pow2 [8] = {1, 2, 4, 8, 16, 32, 64, 128};
    int  a, ret;
    a = b + c;
    if (a < 8)
        ret = pow2[a];
    else
        ret = 0;
    return(ret);
}
```

Arithmetic Instructions

Consider C statement: a = b + c; (line 5)

Assume the variables are in registers `x1-x3` respectively

```
Add x1, x2, x3        # a = b + c
```

Similar instructions

- Arithmetic: sub
- Logical: and, or, nor

Complex Operations

What about more complex C statements?

```
a = b + c + d - e;
```

Break into multiple instructions

```
Add xt0, x1, x2         # x5 = b + c
Add xt0, xt0, x3             # x5 = x5 + d
Sub xt0, xt0, x4             # a = x5 - e
```

xt0 is a temporary register

Numbers representation: Signed & Unsigned

If given b[n-1:0] in a register or in memory

Unsigned value

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

Signed value (2's complement)

$$value = -(b_{n-1} 2^{n-1}) + \sum_{i=0}^{n-2} b_i 2^i$$

Unsigned & Signed Numbers

X	unsigned	signed
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

Example values

4 bits

Unsigned: $[0,\ 2^4 - 1]$

Signed: $[-\ 2^3,\ 2^3 - 1]$

Equivalence

Same encodings for non-negative values

Uniqueness

Every pattern represents unique integer

Not true with sign magnitude

RISC-V Constants

Often want to specify a constant operand in the instruction

Constant == immediate == literal == offset

Use the `addi` instruction

`addi dst, src1, immediate`

The immediate is a 12 bit signed value between -2^{11} and $2^{11}-1$

To enable 32 bit immediate use **`lui dst, immediate`**

Example:

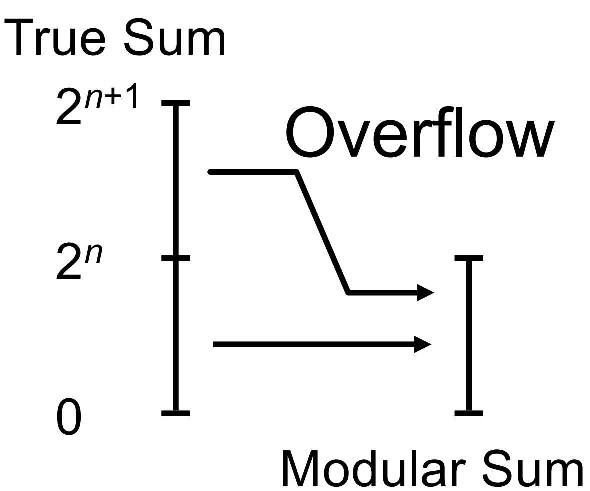
```
C: a++;
RISC-V: addi x1, x1, 1    # a = a + 1
```

Keep in Mind: Arithmetic Overflow

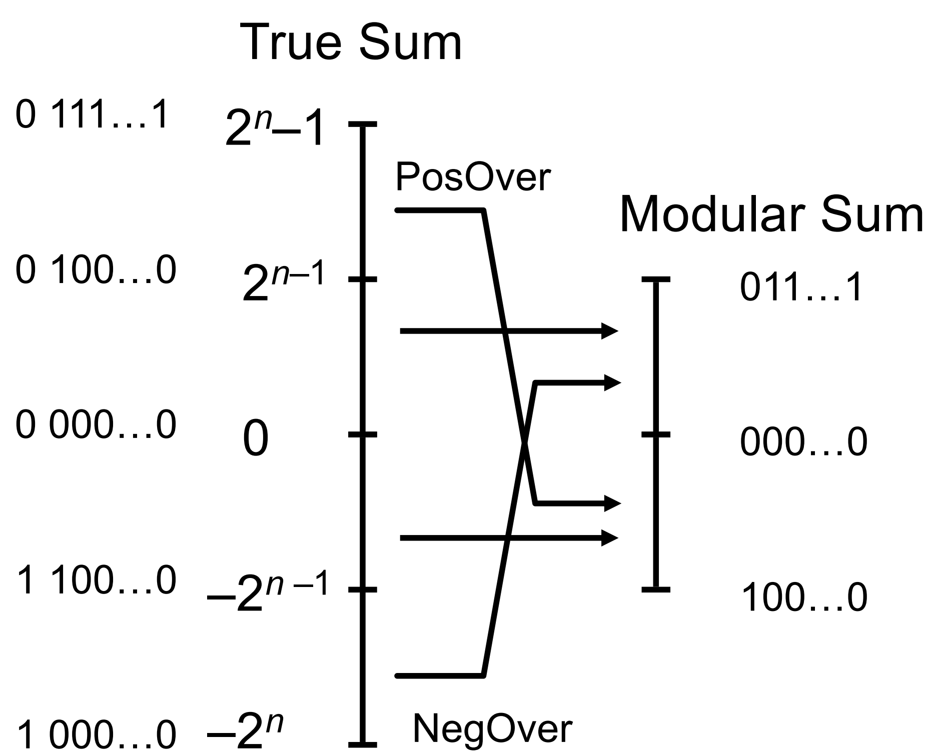
When the sum of two n-bit numbers can not be represented in n bits

Unsigned

- Wraps Around
  - If true sum $\geq 2^n$
  - At most once

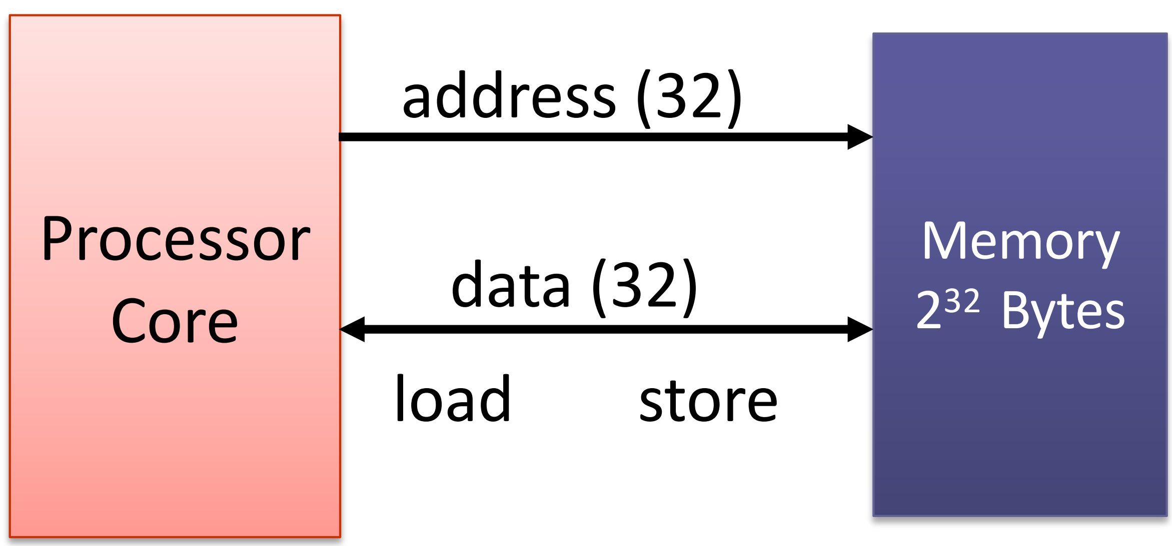


Signed



Memory Data Transfer

- Data transfer instructions move data to and from memory
  - Load moves data from a memory location to a register
  - Store moves data from a register to a memory location
- All memory access happens through loads and stores
  - Floating-point loads and stores for accessing FP registers



Data Transfer Instructions: Loads

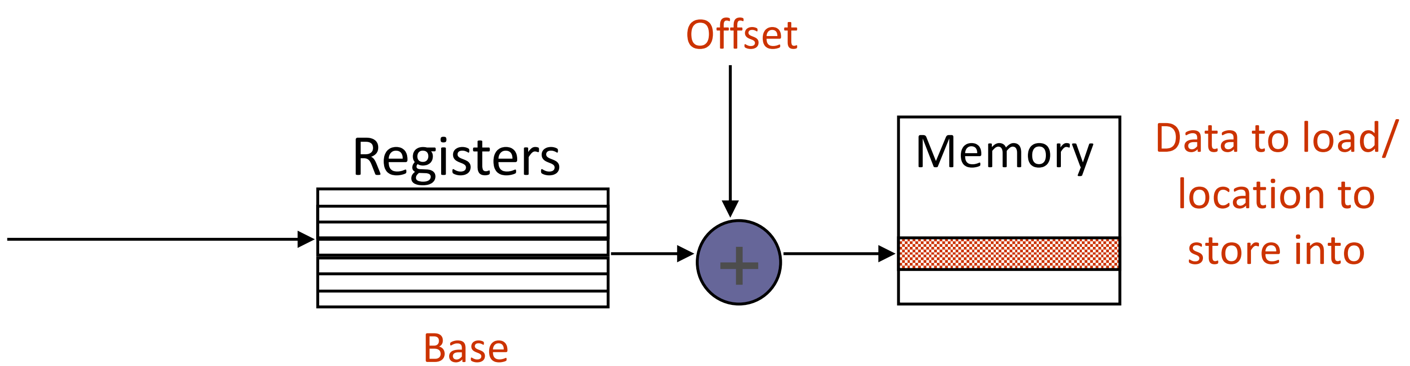
Data transfer instructions have three parts

- Operator name (defines the transfer size as well)
- Destination register
- Base register address and constant offset

```
ld dst, offset(base)
```

- Offset value is a 12-bit signed constant (immediate)

Displacement Addressing Mode



Effective address is a byte addresses

- Must be aligned to words, half-words, & bytes
- More on this later

Loading Data Example

Consider the C example: `a = b + *c;`

Assume a in x1, b in x2, c in x3

`ld` instruction:

```
Ld xt0, 0(x3)          # xt0 = Memory[c]
                       # xt0 is temp reg
Add x1, x2, xt0# a = b + *c
```

Accessing Arrays

Arrays are really pointers to the base address in memory

- Address of element A[0]

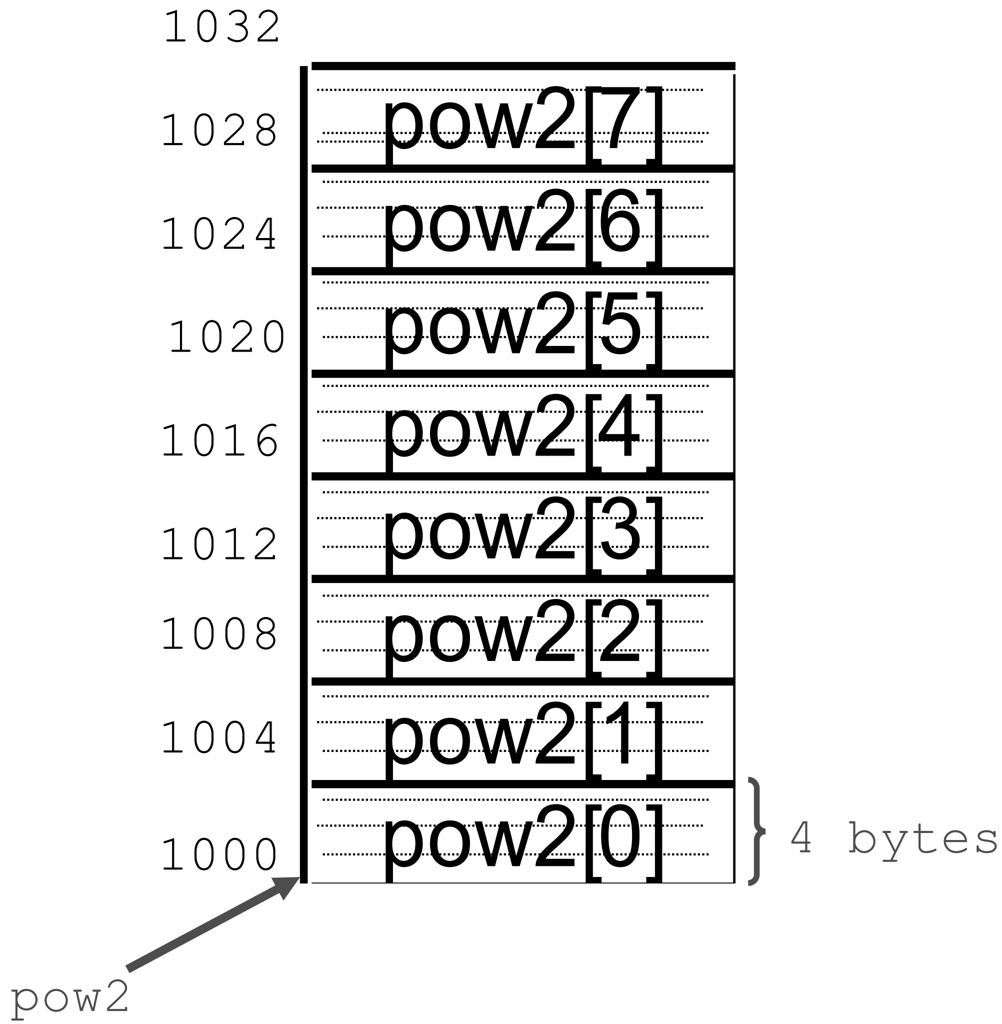
Use offset value to indicate which index

- Note: addresses are in bytes, so multiply by the size of the element
- Unlike C, assembly does not do pointer arithmetic for you!

Consider the integer array where `pow2` is the base address

- With this compiler on this architecture, each int requires 4 bytes
- The data to be accessed is at index 5: `pow2[5]`
- Then the address from memory is `pow2 + 5 * 4`

Array Memory Diagram



Array Example

Example: **`a = b + pow2[7];`**

```
x3 = 1000
```

`lw` instruction with offset:

```
Ld xt0, 28(x3)    # xt0 = Memory[pow2[7]]
Add x2, x1, xt0   # a = b + pow2[7]
```

Storing Data

Store is the reverse of load

- And identical in address generation

Copy data from the source register to an address in memory

```
sd src, offset(base)
```

Offset value is a 12-bit signed constant

Storing Data Example

Example: `*a = b + c;`

Assume a in x3, b in x1, c in x2

`sd` instruction:

```
add xt0, x1, x2         # $t0 = b + c
sd xt0, 0(x3)           # Memory[s0] = b + c
```

Storing to an Array

Example: `a[3] = b + c;`

Assume b in x2, c in x3, a in x4

sd instruction with offset

```
add xt0, x2, x3      # $t0 = b + c
sd xt0, 12(x4)       # Memory[a[3]] = b + c
```

Question: Why 12 and not 3?

Indexed Array Storage

Example: `a[i] = b + c;`

Assume i in x3, b in x1, c in x2, a in x4

Address generation + store:

```
Add xt0, x1, x2# $t0 = b + c
Sll xt1, x3, 2       # $t1 = 4 * i
Add xt2, x4, xt1     # $t2 = a + 4*i
sd  xt0, 0(xt2)# Memory[a[i]] = b+c
```

Interface to I/O

Load/stores provide interface to memories

How about interface to I/O devices?

Huge variety in I/O devices

- Printer, USB camera, network interface, hard disk….
- Huge variety in functionality and performance requirements

Special I/O instructions for each type of device

- What would be the problem with this?

Interface to I/O

Break the problem in two

- Communicating bits to the I/O device
- Executing operations based on these bits

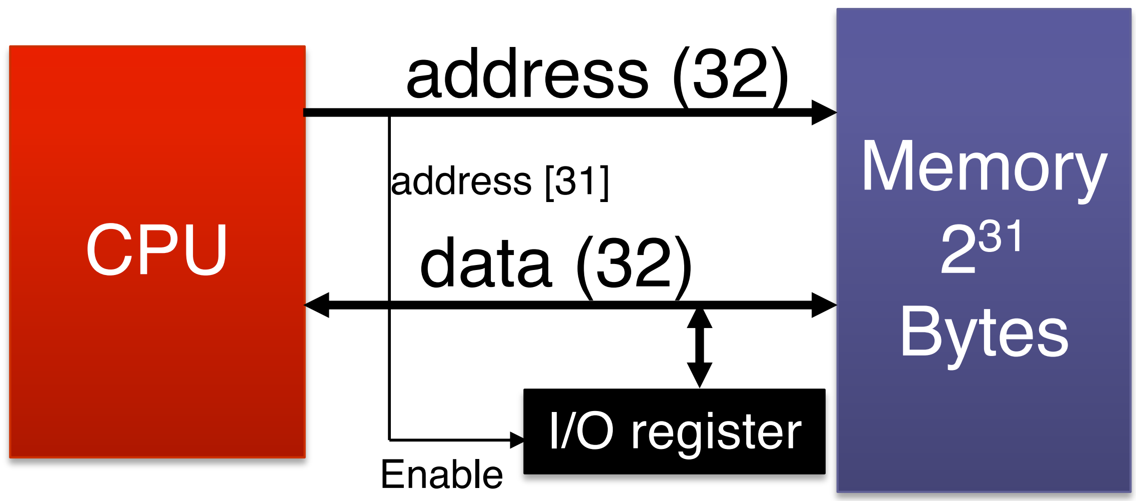
Register/memory based interface to I/O

- Every device includes some registers and memory
- Reading/writing these registers communicates bits
  - Similar across all I/O devices
- Every device has it own protocol of what these bits mean
  - Check status, read command, send command, …
  - Software must know this protocol (device driver)

Memory Mapped I/O: Preview

Loads/stores can be used to move bits to/from I/O devices

- A load moves data from a an I/O device register to a CPU register
- A store moves data from a CPU register to a I/O device register



I/O register at address 0x80000000