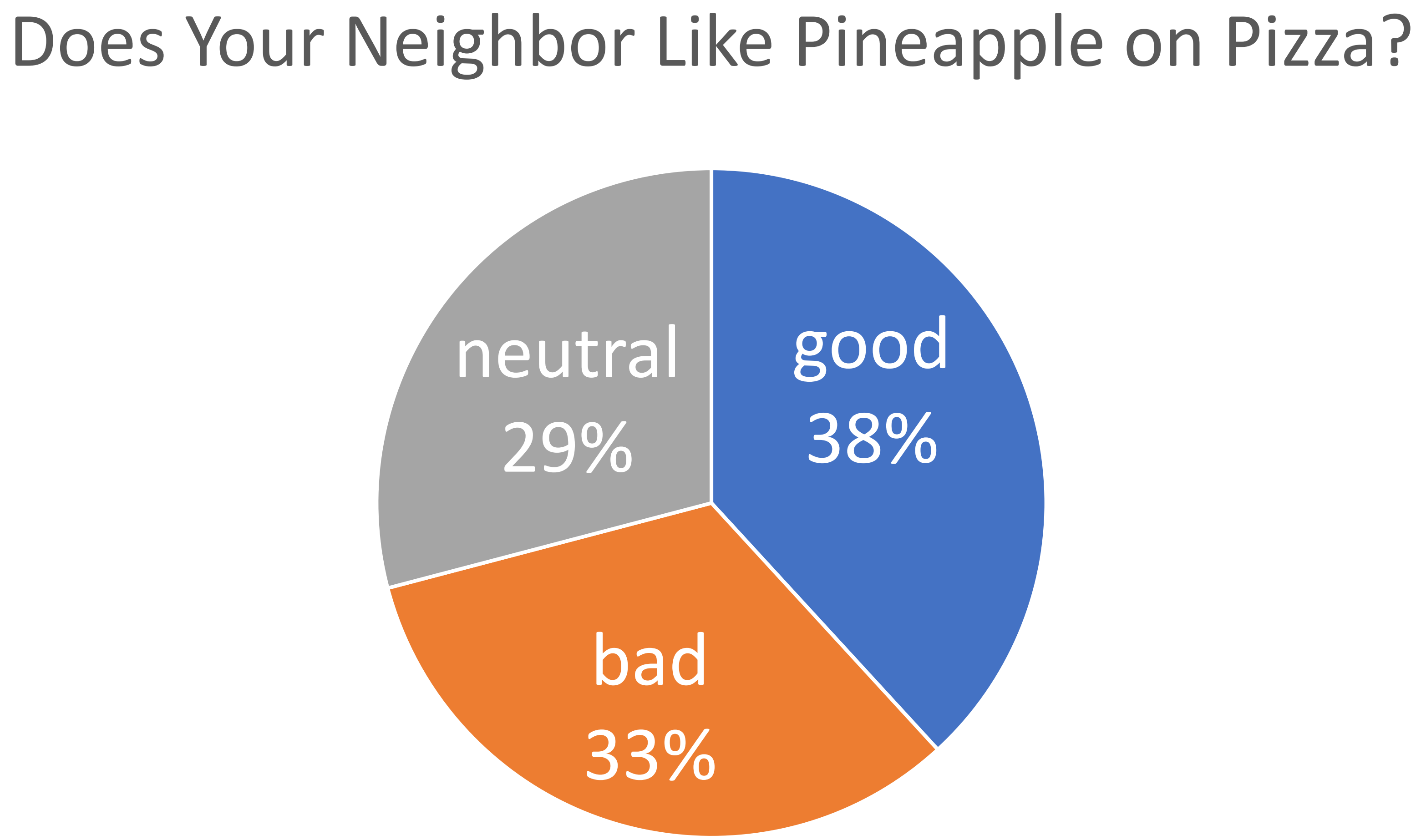
EECS 370 - Lecture 3

LC2K


All right, check out this bad boy.
Twelve megabytes of RAM, 500 megabyte hard drive.
Built-in spreadsheet capabilities and a modem that transmits at over 28,000 BPS.
PRODUCED BY
TODD STEVENS
Wow. What are you gonna use it for?
Games and stuff.

Important Data



Lingering Questions

- "How does the compiler know what is what? For example how does the computer know that 100011 is supposed to be a char 'C' instead of an unsigned int of 67, or a signed int of -61."
  - The compiler knows because the user will label a variable as "char" or "int" or "unsigned int"
  - The hardware doesn't know or care: it's just 100011 and trusts that the program will execute the appropriate instructions

- Lingering questions / feedback? I'll include an anonymous form at the end of every lecture: https://bit.ly/3oXr4Ah

Announcements

- HW 1
  - Posted on website, due next Friday
- P1
  - 3 parts, first part due in two weeks
- Setup clinic
  - Need help getting your debugger setup?
  - Dedicated help on Friday!
  - Homework 1 has a question requiring you to show debugger is setup
- OH
  - Going on now: see website->Google Calendar->Office Hours

Instruction Set Architecture (ISA) Design Lectures

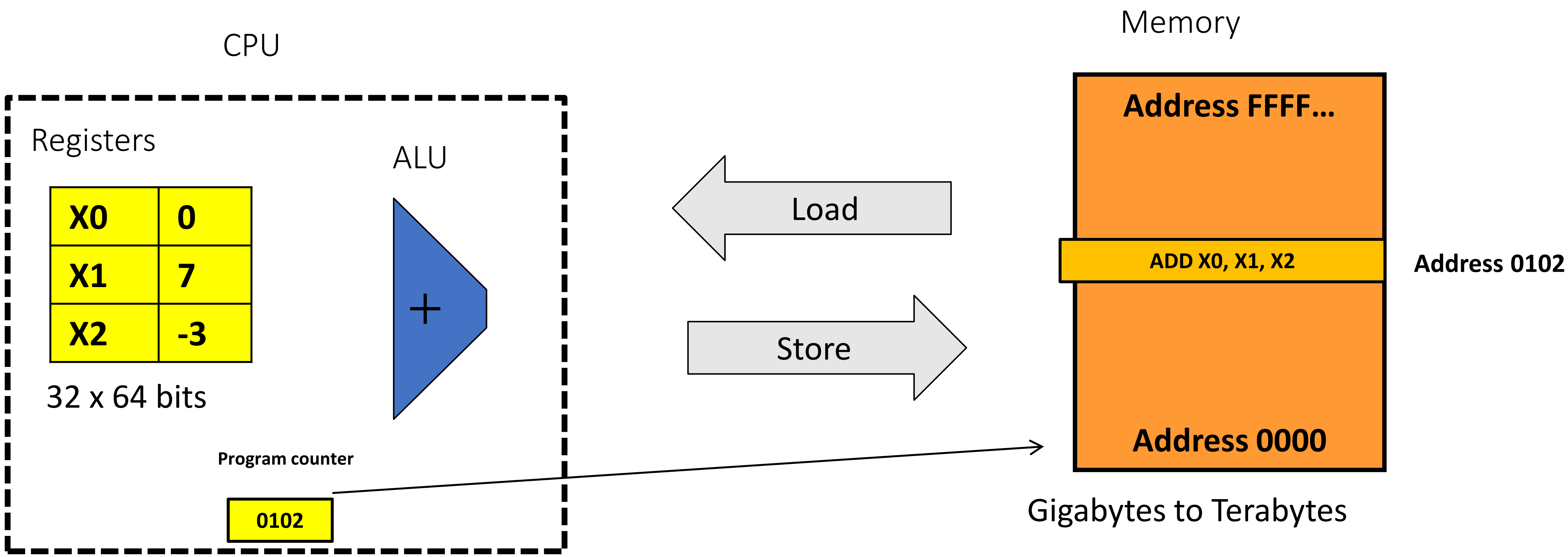
"People who are really serious about software should make their own hardware." — Alan Kay

- Lecture 2: ISA - storage types, binary and addressing modes
- **Lecture 3 : LC2K**
- Lecture 4 : ARM
- Lecture 5 : Converting C to assembly – basic blocks
- Lecture 6 : Converting C to assembly – functions
- Lecture 7 : Translation software; libraries, memory layout

Reminder- System Organization

Let's execute this short program:

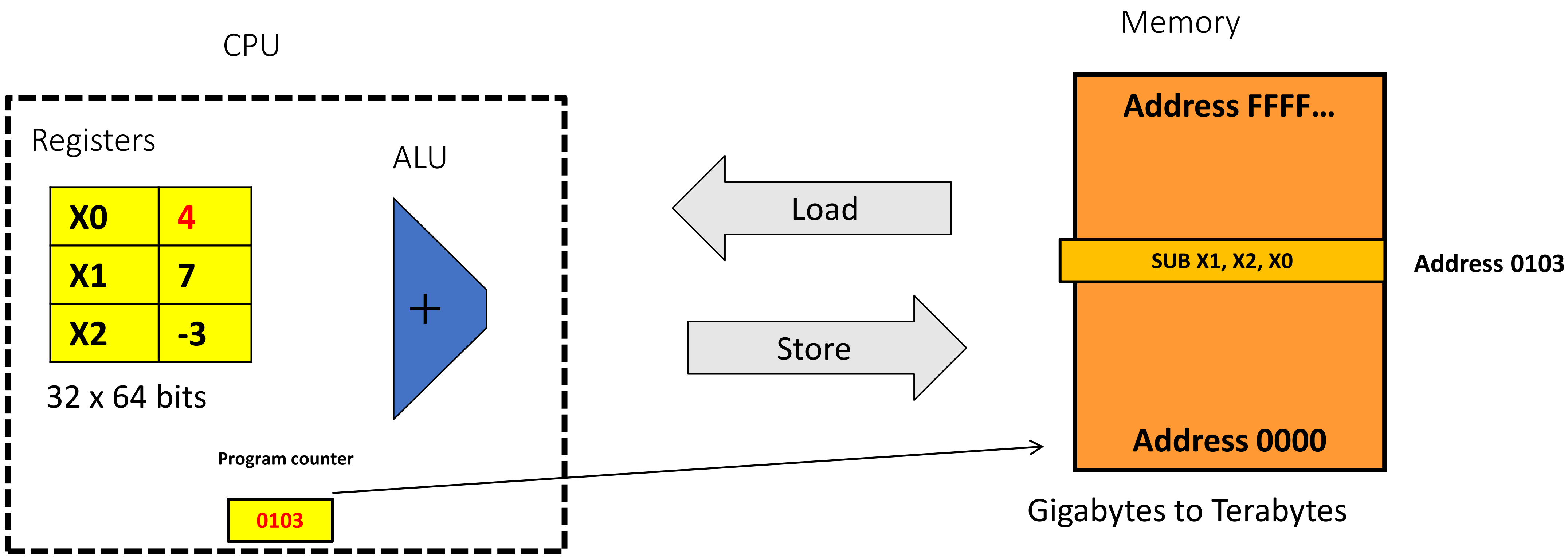
ADD X0, X1, X2
SUB X1, X2, X0



Reminder- System Organization

Let's execute this short program:

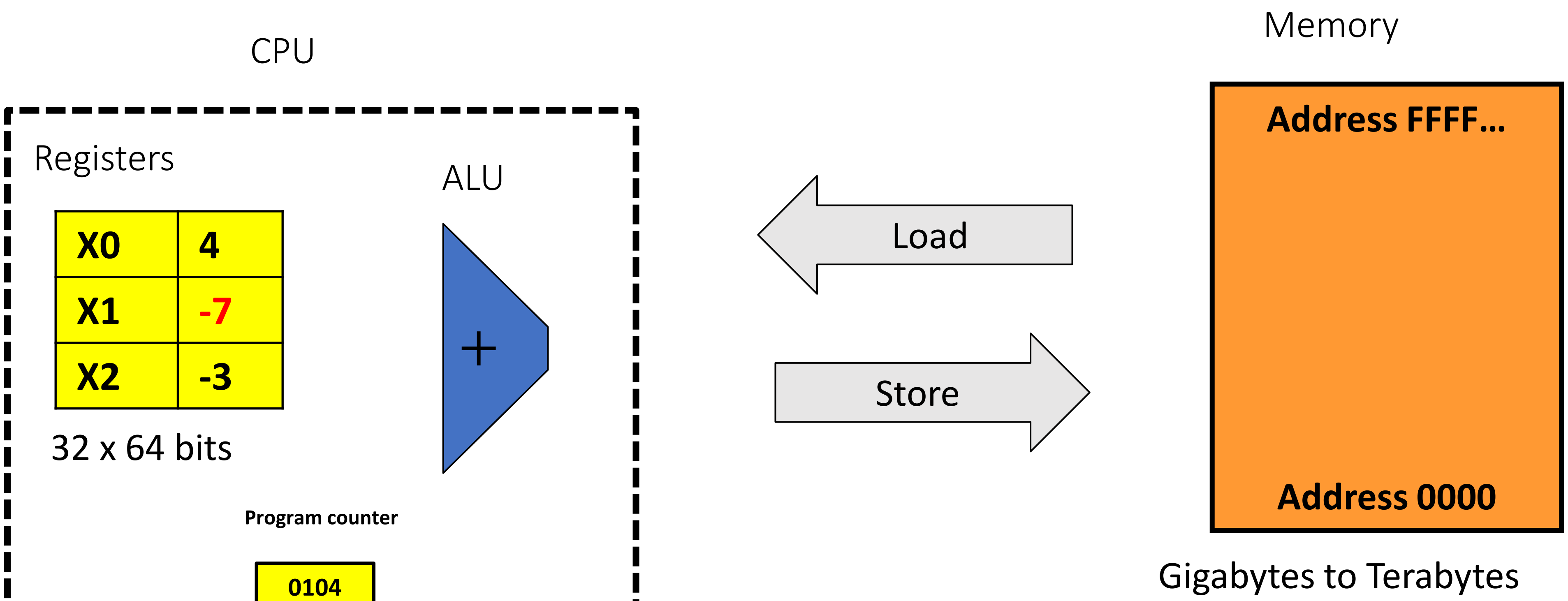
ADD X0, X1, X2
SUB X1, X2, X0



Reminder- System Organization

Let's execute this short program:

```
ADD X0, X1, X2
SUB X1, X2, X0
```



Encoding Instructions

- We saw last time that binary numbers can represent signed and unsigned numbers, chars, and fractional numbers
- But they must also represent instructions themselves!
  - After all, memory is just a collection of 1s and 0s
- We need a way of ***encoding*** instructions in order to store them in memory

Software program to machine code

Example ISA (simplified)

Compile → **Assemble**

C program

```
main()
{
  int a, *b, c;
  c = a + *b;
}
```

Assembly code

```
.text
.global main
r4 = load(r2);
r3 = add(r1,r4);
```

Machine code

```
0001 1101 0010 0000
1010 0001 0110 1011
```

Assembly Instruction Encoding

- Since the EDSAC (1949) almost all computers stored program instructions the same way they store data.
- Each instruction is encoded as a number

Example ISA (simplified)

add	R2	R3	R1
011011	010	011	001

$$011011010011001 = 2^0 + 2^3 + 2^4 + 2^7 + 2^9 + 2^{10} + 2^{12} + 2^{13}$$

$$= \mathit{13977}$$

- This is the number stored in memory!

Poll: How many different "operation codes" could be supported by this ISA? How many registers?

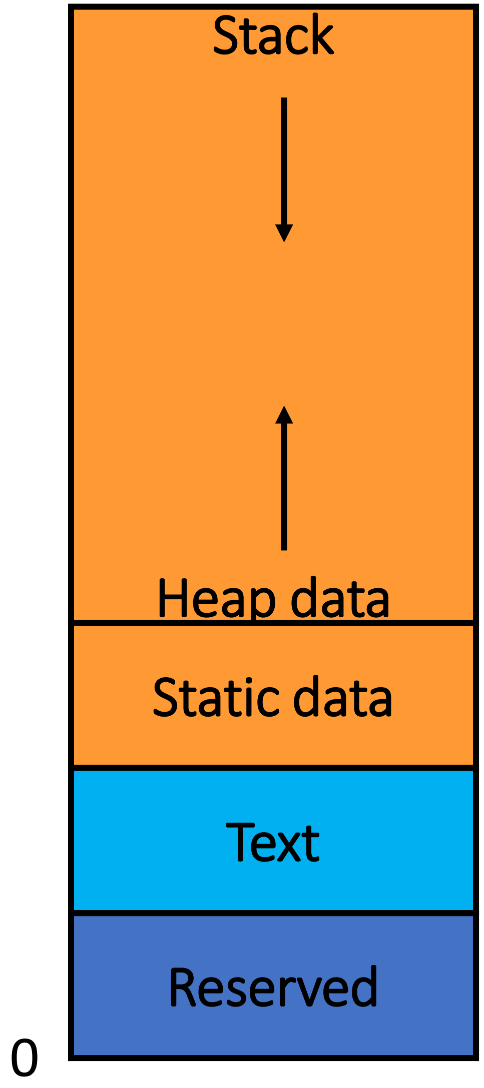
What if we run out of registers?

- Modern architectures give us between 4 and 64 registers
- What if I want to sort a list of 1000 integers?
- Need to make use of **memory**
- Large array of storage accessed using memory addresses
  - A machine with a 32 bit address can reference memory locations 0 to 2^{32}-1 (or 4,294,967,295).
  - A machine with a 64 bit address can reference memory locations 0 to 2^{64}-1
    (or 18,446,744,073,709,551,615—18 exa-locations)

- **load** instructions ready from memory, places in a register
- **store** instructions write from register into memory

"We will never make a 32-bit operating system" - Bill Gates, 1983

Memory architecture: The ARM (Linux) Memory Image



Stack frames: local variables, parameters, etc.

Dynamically allocated data—new or malloc()

Global data and static local data

Machine code instructions (and some constants)

Reserved for operating system

Addressing Modes

- Direct addressing
- Register indirect
- Base + displacement
- PC-relative

Direct Addressing

Example ISA (simplified)

- Consider this code:

```
const double PI = 3.14;

double two_pi() {
  return 2*PI;
}
```

Not practical in modern ISAs… if we have 32 bit instructions and 32 bit addresses, the entire instruction is the address!

- When we load PI, it's ALWAYS the same address
  - If the ISA supports it, we can just hardcode that address in the instruction
- Like register addressing
  - Specify address as immediate constant

```
load r1,  mem[1500]  ; r1 ← contents of location 1500
jump      mem[3000]  ; jump to address 3000
```

- Useful for addressing locations that don’t change during execution
  - Branch target addresses
  - Global/static variable locations

Example ISA (simplified)

Register indirect

- Consider this code:

```
int my_arr[2] = {6666, 7777};
int* ptr = &my_arr[0];
for(int i=0; i<2; i++) {
  int x = *ptr;
  ptr++;
}
```

- Everytime we load into x, it's a different address
- But the address is always stored in another variable
- If ISA supports it, we could use a load like this
  - **`load r1, mem[r2]`**

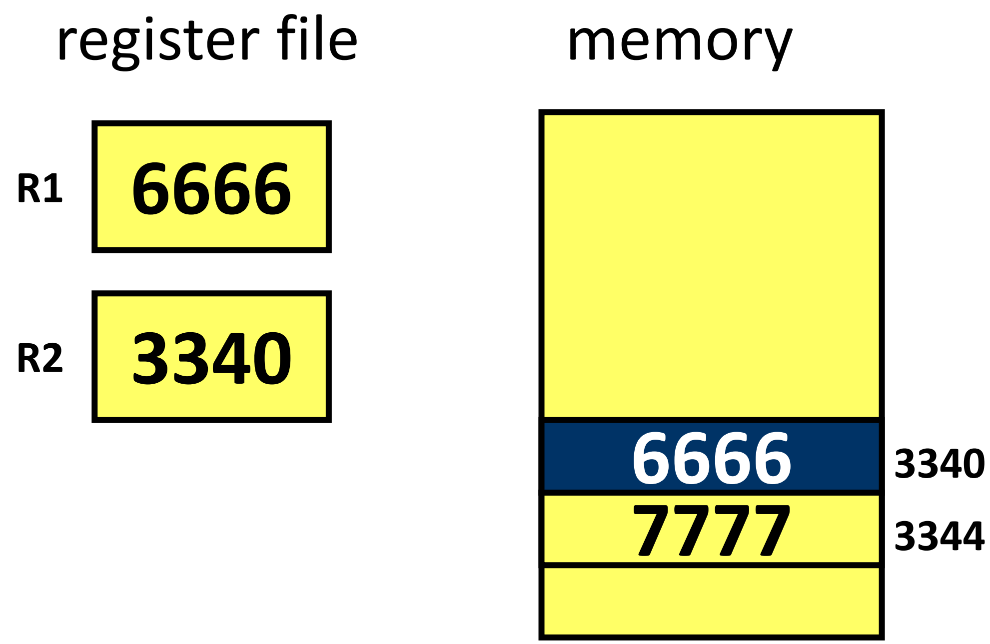
Example ISA (simplified)

Register indirect

- Consider this code:

```
int my_arr[2] = {6666, 7777};
int* ptr = &my_arr[0];
for(int i=0; i<2; i++) {
  int x = *ptr;
  ptr++;
}
```

```
load r1, mem[ r2 ]
add  r2, r2, #4
load r1, mem[ r2 ]
```



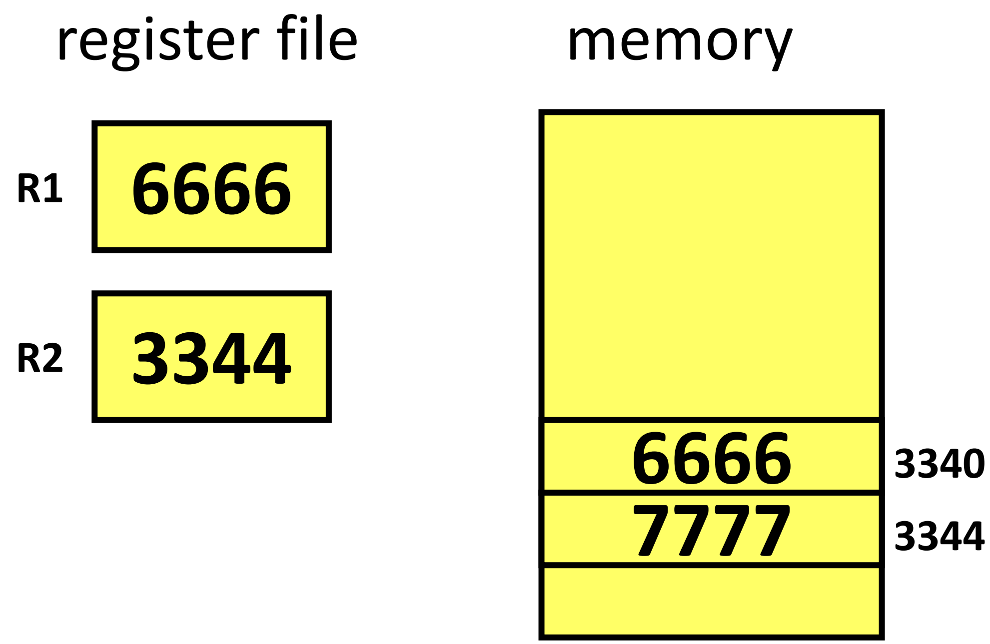
Example ISA (simplified)

Register indirect

- Consider this code:

```
int my_arr[2] = {6666, 7777};
int* ptr = &my_arr[0];
for(int i=0; i<2; i++) {
  int x = *ptr;
  ptr++;
}
```

```
load r1, mem[ r2 ]
add  r2, r2, #4
load r1, mem[ r2 ]
```



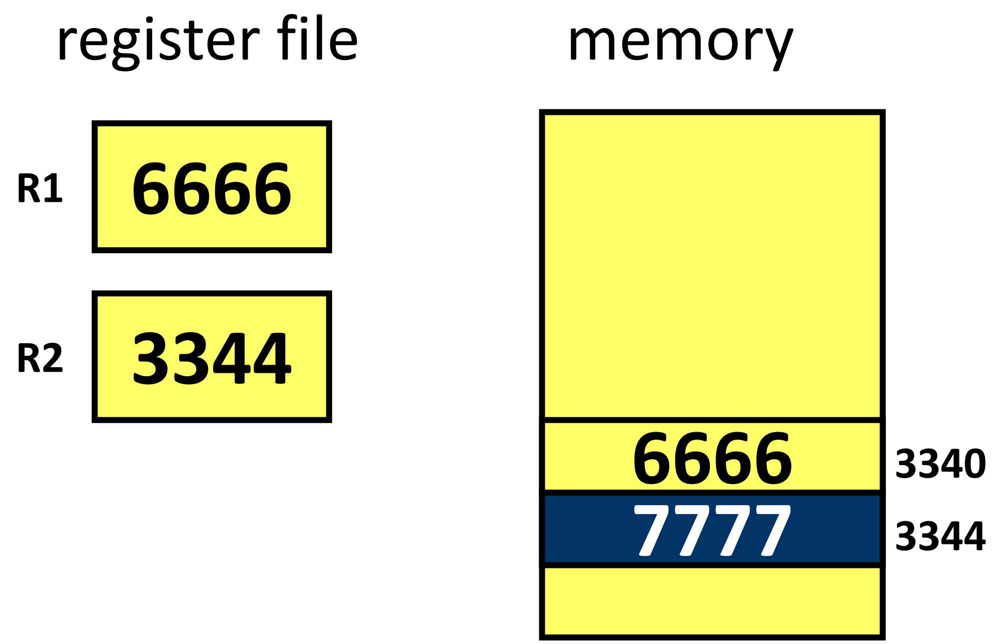
Example ISA (simplified)

Register indirect

- Consider this code:

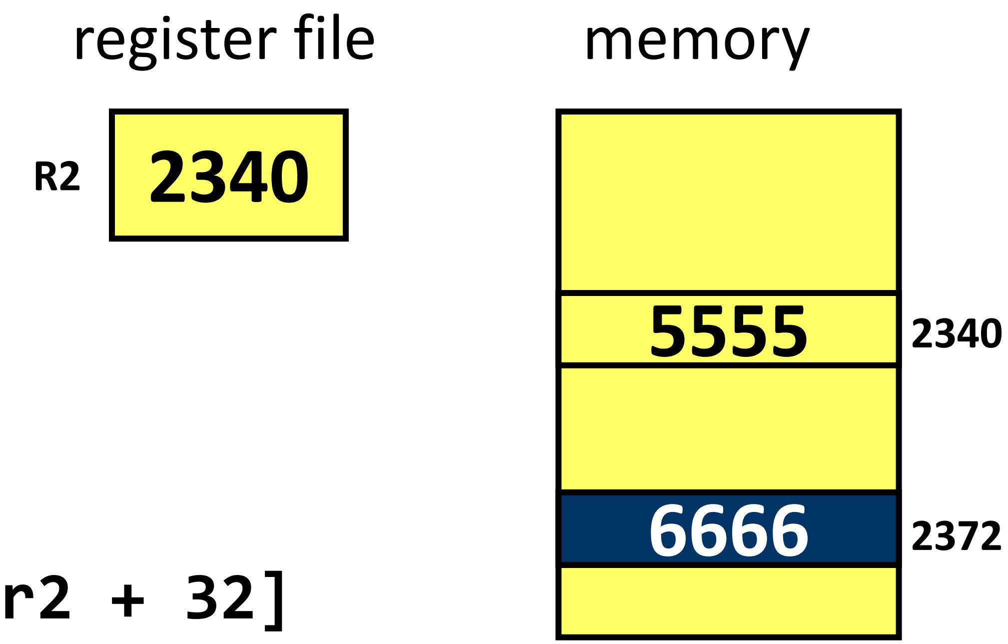
```
int my_arr[2] = {6666, 7777};
int* ptr = &my_arr[0];
for(int i=0; i<2; i++) {
  int x = *ptr;
  ptr++;
}
```

```
load r1, mem[ r2 ]
add  r2, r2, #4
load r1, mem[ r2 ]
```



This is better, but we can be more general

Base + Displacement



- Consider this code:

```
struct My_Struct {
  int tot;
  //...
  int val;
};

My_Struct a;
//...
a.tot += a.val;
```

```
load r1, mem[r2 + 32]
```

- If a register holds the starting address of "a"...
  - Then the specific values needed are a slight **offset**
- **Base + Displacement**
  - reg value + immed

Very general, most common addressing mode today

Class Problem

a. What are the contents of register/memory after executing the following instructions

```
r2 =  load mem[r3]
r3 =  load mem[r2+4]
store mem[r2+8], r3
```

register file

R1	0
R2	10
R3	108

memory

108	100
-1	104
100	108

<u>Poll:</u> What are the contents of register / memory?

PC-relative addressing

- **Relevant for P1.a!**

- Variant on base + displacement
- Remember PC is "Program Counter", keeps track of which line (memory address) of the program we're at
- PC register is base, longer displacement possible since PC is assumed implicitly (more bits available)
  - Used for branch instructions
    - jump [- 8] ; jump back 2 instructions (32-bit instructions)

ISA Types

Reduced Instruction Set Computing (RISC)

- Fewer, simpler instructions
- Encoding of instructions are usually the same size
- Simpler hardware
- Program is larger, more tedious to write by hand
- E.g. LC2K, RISC-V, ARM (kinda)
- More popular now

Complex Instruction Set Computing (CISC)

- More, complex instructions
- Encoding of instructions are different sizes
- More complex hardware
- Short, expressive programs, easier to write by hand
- E.g. x86
- Less popular now

LC2K ISA

Programming Assignment #1

- Write an assembler to convert input (assembly language program) to output (machine code version of program)
  - “1a”
- Write a behavioral simulator to run the machine code version of the program (printing the contents of the registers and memory after each instruction executes
  - “1s”
- Write an efficient LC2K assembly language program to multiply two numbers
  - “1m”

Programming Assignment #1

- Where to start…
  - **<u>Write some test cases</u>** to check your code
    - Program 1: halt
    - Program 2: noop
    - halt
    - Program 3: add 1 1 1
    - halt
    - Program 4: nor 1 1 1
    - halt

LC2K Processor

LC2K ISA

- 32-bit processor
  - Instructions are 32 bits
  - Integer registers are 32 bits
- 8 registers
  - register 0 always gives the value 0
- supports 65536 words of memory (addressable space)
- 8 instructions in the following common categories:
  - Arithmetic: add
  - Logical: nor
  - Data transfer: lw, sw
  - Conditional branch: beq
  - Unconditional branch (jump) and link: jalr
  - Other: halt, noop

LC2K ISA

Instruction Encoding

- Instruction set architecture defines the mapping of assembly instructions to machine code

add 1 2 3 // r3 = r1 + r2

31-25	24-22	21-19	18-16	15-3	2-0
unused	**opcode**	**regA**	**regB**	*unused*	**destR**

LC2K ISA

Instruction Formats

- Tells you which bit positions mean what
- R (register) type instructions (add ‘000’, nor ‘001’)

31-25	24-22	21-19	18-16	15-3	2-0
unused	**opcode**	**regA**	**regB**	*unused*	**destR**

- I (immediate) type instructions (lw ‘010’, sw ‘011’, beq ‘100’)

31-25	24-22	21-19	18-16	15-0
unused	**opcode**	**regA**	**regB**	**offset**

LC2K ISA

Instruction Formats

- J-type instructions (jalr ‘101’)

31-25	24-22	21-19	18-16	15-0
unused	**opcode**	**regA**	**regB**	*unused*

- O type instructions (halt ‘110’, noop ‘111’)

31-25	24-22	21-0
unused	**opcode**	*unused*

LC2K ISA

Bit Encodings

- Opcode encodings
  - add (000), nor (001), lw (010), sw (011), beq (100), jalr (101), halt (110), noop (111)
- Register values
  - Just encode the register number (r2 = 010)
- Immediate values
  - Just encode the values in **2's complement format**

Example Encoding - nor

LC2K ISA

- nor 3 4 7 (r7 = r3 nor r4)

31-25	24-22	21-19	18-16	15-3	2-0
unused	**opcode**	**regA**	**regB**	*unused*	**destR**
0000000	001	011	100	0000000000000	111

Convert to Hex → 0x005C0007

Convert to Dec → 6029319

LC2K ISA

Example Encoding - lw

- lw 5 2 -8 (r2 = mem[r5 + -8])

31-25	24-22	21-19	18-16	15-0
unused	**opcode**	**regA**	**regB**	**offset**
0000000	010	101	010	1111111111111000

Convert to Hex → 0x00AAFFF8

Convert to Dec → 11206648

Note that we "bit-extend" 1 for negative numbers

Another way to think about the assembler

- Each line of assembly code corresponds to a number
  - "add 0 0 0" is just 0.
  - "lw 5 2 -8" is 11206648
- We only write in assembly because it's easier to read.

.fill

- I also might want a number, to be, well, a number.
  - Maybe I want the number 7 as a data element I can use.
- .fill tells the assembler to put a number instead of an instruction
- The syntax is just ".fill 7".
- Question:
  - What do ".fill 7" and "add 0 0 7" have in common?

.fill

- <u>**.fill is NOT an instruction**</u>
- It does not have a corresponding opcode
- It should be used to initialize data in your program
  - If your PC ever points to it, something wrong has probably happened
- But if the PC **DOES** point to it, it will treat it as whatever type of instruction encodes to that number

Labels in LC2K

- Labels are used in lw/sw instructions or beq instruction
- For lw or sw instructions, the assembler should compute offsetField to be equal to the address of the label
  - i.e. offsetField = address of the label
- For beq instructions, the assembler should translate the label into the numeric offsetField needed to branch to that label
  - i.e. PC+1+ offsetField = address of the label

Labels in LC2K

- Labels are a way of referring to a line number in an assembly program.

```
loop   beq   3   4   end
       noop
       beq   0   0   loop
end    halt
```

- Here **loop** is 0 and **end** is 3.

```
// this is the
assembly for:
while(x != y) {
  x *= 2;
}
```

- What are the values of the labels here?

```
loop   beq   3   4   end
       add   3   3   3
tom    noop
       beq   0   0   loop
end    halt
```

<u>Poll:</u> What are the labels replaced with?

Next Time

- The ARM ISA
- Lingering questions / feedback? I'll include an anonymous form at the end of every lecture: https://bit.ly/3oXr4Ah

Extra Problem 1

LC2K ISA

- Compute the encoding in Hex for:
  - add 3 7 3 (r3 = r3 + r7) (add = 000)
  - sw 1 5 67 (M[r1+67] = r5) (sw = 011)

31-25	24-22	21-19	18-16	15-3	2-0
unused	**opcode**	**regA**	**regB**	*unused*	**destR**
0000000	000	011	111	000…000	011

31-25	24-22	21-19	18-16	15-0
unused	**opcode**	**regA**	**regB**	**offset**
0000000	011	001	101	0000000001000011

Extra problem 2

```
loop   lw    0   1   one
       add   1   1   1
       sw    0   1   one
       halt
one    .fill 1
```

<u>Poll:</u> What's the first line in binary?

- What does that program do?
- Be aware that a beq uses PC-relative addressing.
  - Be sure to carefully read the example in project 1.