

CS 24: INTRODUCTION TO COMPUTING SYSTEMS

Spring 2015
Lecture 1

WELCOME TO CS24!

- Introduction to Computing Systems
- How do modern digital computers work?
- What features, capabilities, and optimizations do processors provide?
- How do we translate programs to run on processors?
 - e.g. intermediate values, looping, subroutines, recursion
- How to provide common runtime support?
 - e.g. memory management
- What do operating systems do for us?!
 - e.g. process isolation, virtualization, input/output

CS24 ADMINISTRIVIA

- Course website: Caltech Moodle
 - <http://courses.caltech.edu>
 - Go to the CS section, then click CS24 (key = segfault)
- **Make sure to enroll in CS24 course today!**
 - Class announcements are made via Moodle
- All lectures, assignments posted on CS24 Moodle
 - Submit homeworks and receive grades via Moodle
 - (I will keep track of your overall grade separately)
- Assignment grading guidelines will be posted
 - **Correct programs are not sufficient!**
 - Style, clarity, commenting, etc. are also important

CS24 ADMINISTRIVIA (2)

- Approximate course weighting:
 - 8 assignments (70% of grade)
 - Each assignment is 8.75% of your grade
 - Midterm (15% of grade)
 - Final exam (15% of grade)
- I will sometimes curve individual assignments or exams, depending on the class' average grade
 - Tends to be done infrequently
 - On average, people do very well on assignments

CS24 LATE POLICY

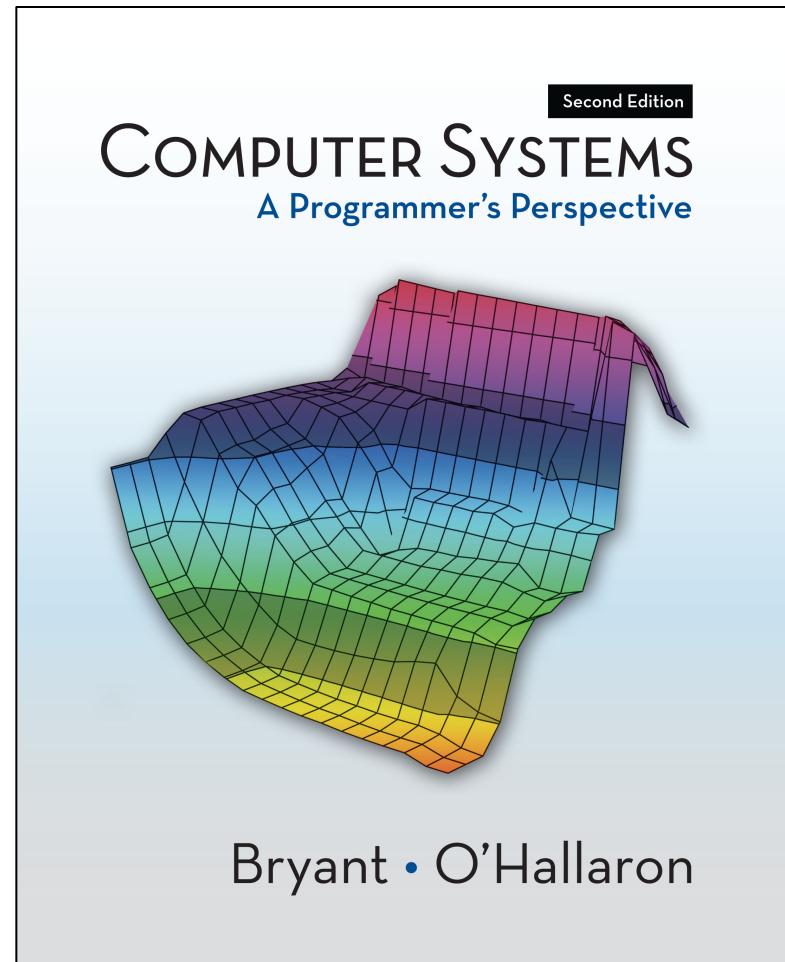
- Late assignments will be penalized:
 - Up to 1 day (24 hours) late: 10% deduction
 - Up to 2 days (48 hours) late: $10 + 20 = 30\%$
 - Up to 3 days (72 hours) late: $10 + 20 + 30 = 60\%$
 - After 3 days: Sorry, don't bother. ☹
- Every student has 4 “late tokens”
 - Each token is worth up to 24 hours of extension, “no questions asked”
 - State on your submission how many tokens you used
- Also, notes from Dean’s Office or Health Center will almost always warrant an extension
 - (no tokens are consumed this way)

CS24 ASSIGNMENTS

- CS24 is a very time-consuming class
- Always start assignments well before they're due!
 - Don't get caught by assignments you didn't expect to be hard for you
 - You will make the most of office hours this way, too
- Some assignments are more involved than others
 - I will warn you ahead of time
- Submit your assignments via Moodle
 - Instructions for packaging at top of each assignment
 - If you don't follow these instructions, you will lose points on your assignments
 - We will be happy to help if you need help with this

TEXTBOOK – CS:APP2E

- Computer Systems: A Programmer's Perspective
 - A *very* good textbook, from Carnegie Mellon ICS class
 - amazon.com: \$126 new (eText edition: \$105)
- Book is optional for the course
 - A great book, but too expensive
 - Copies of most relevant material (and HW problems) are provided
 - Chapters/sections to read are specified for each week



PROGRAMMING LANGUAGES FOR CS24

- Assignments involve programming with C, and IA32 assembly language
 - You are expected to have a general familiarity with C (syntax, pointers, structs, memory management)
 - IA32 is introduced more gently, along the way
- CS:APP contains many helpful hints for C in each chapter
 - A great resource if not intimately familiar with C
 - TAs can help with nuances of using C, but they will not teach you C from scratch – that's not their job.
- IA32 assembly language:
 - Language used for programming Intel x86-family processors

PROGRAMMING ENVIRONMENT

- All assignments must be completed on a 32-bit Linux platform
 - No 64-bit Linux. No MacOS X. No Cygwin.
- Multiple technical reasons:
 - Incompatibilities between 32-bit and 64-bit platforms
 - Variations in how OSes link and load C programs
 - Some assignments use low-level system APIs that are only provided on Linux, not MacOS X or Cygwin
- Use a CS cluster account (recommended)
- Or, install 32-bit Linux in a virtual machine on your computer (e.g. using Virtual Box)
 - We will provide a VM image in the next few days

PROGRAMMING ENVIRONMENT (2)

- GNU toolset:
 - **gcc** for C programming
 - **as** (GNU assembler) for IA32 assembly
 - GNU **make** for building/running programs
 - **gdb** for finding your bugs ☺
- Will provide supplemental material for **gcc**, **gdb** and **make** on Moodle
- Will also provide recordings of two older lectures on how to debug programs with **gdb**
 - You should watch these by end of 2nd week of class
 - You want to learn **gdb** – it will shave *hours* off of your assignments!

MOTIVATIONS FOR CS24

- Why study computing systems in the first place?
- Reason 1:
 - Understanding how the computer works will help you to use it more effectively.
 - You will be a better programmer if you understand the details of how the computer works.

EXAMPLE: MOLECULAR DYNAMICS

- Experiments involve simulating individual atoms

```
#define N_ATOMS 10000
#define DIM 2
/* Array of data for each atom being simulated. */
double atoms[N_ATOMS][DIM][DIM];
```

- Version 1:

```
for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
        for (n = 0; n < N_ATOMS; n++)
            atoms[n][i][j] = ... ;
```

- Version 2:

```
for (n = 0; n < N_ATOMS; n++)
    for (i = 0; i < DIM; i++)
        for (j = 0; j < DIM; j++)
            atoms[n][i][j] = ... ;
```

- Why is version 2 significantly faster than version 1?

EXAMPLE: FINANCIAL COMPUTATIONS

- Candy Shop in the Math Department:
 - First candy costs 10¢
 - Each subsequent candy costs 10¢ more than previous one
 - You have one dollar to spend
 - How many candies can you purchase?
- Write a C program to solve it:

```
float fundsLeft = 1.0, price;
int numCandies = 0;
for (price = 0.1; price <= fundsLeft; price += 0.1) {
    numCandies++;
    fundsLeft -= price;
}
printf("%d candies; %f left over\n",
       numCandies, fundsLeft);
```

- *Why doesn't this blasted program work properly?!*
 - Output: 3 candies; 0.400000 left over

MOTIVATIONS FOR CS24 (2)

- Why study computing systems in the first place?
- Reason 1:
 - Understanding how the computer works will help you to use it more effectively.
 - You will be a better programmer if you understand the details of how the computer works.
- Both examples are very simple to understand...
 - ...if you actually know how the computer works!
- Will see much more sophisticated examples as we go through the term

MOTIVATIONS FOR CS24 (3)

- Why study computing systems in the first place?
- Reason 2:
 - The concepts we will cover are ubiquitous in modern computing systems
 - Have a profound impact on most hardware designs, and also on operating system design/implementation
- If you ever participate in hardware design, or in operating system design:
 - Need to understand the common challenges, and strengths/weaknesses of the common solutions
 - You might even devise new solutions that are better than what we presently use!

EXAMPLE: MEMORY MANAGEMENT

- Operating systems provide a “process” abstraction
 - Allows multiple programs to share a single CPU “at the same time”
 - e.g. a web browser, text editor, and email client
- Want to isolate memory used by different processes
 - An incorrect program should not cause other programs to crash, or corrupt the operating system itself
- Want to provide a “virtual memory” abstraction
 - OS can allow programs to use more memory than the physical hardware actually provides
- *What features should the hardware provide, to make these features fast, secure, and easy to implement?*

INSTRUCTION SET ARCHITECTURES

- Intel IA32 is a specific example of an Instruction Set Architecture (ISA)
 - A specific set of instructions that can be executed by a processor, along with their byte encodings
- Multiple vendors can implement a specific ISA
 - Intel and AMD both implement the IA32 ISA
- Different kinds of instruction set architectures
- CISC: Complex Instruction Set Computer
 - A large number of very powerful instructions
 - Programs require fewer instructions to implement a particular computation
 - Logic for supporting these instructions is more complicated
 - IA32 is a CISC architecture

INSTRUCTION SET ARCHITECTURES (2)

- RISC: Reduced Instruction Set Computer
 - A relatively small number of simpler instructions
 - Programs require more instructions to implement a computation
 - Hardware implementing these instructions can provide more pipelining
- These days, line between RISC/CISC is often blurred
 - CISC processors can internally translate instructions into RISC-like steps to pipeline and execute
 - RISC processors often include more sophisticated CISC-like instructions
- More pure-RISC processors are seen primarily in embedded/mobile systems
 - Simple and low-power are critical requirements

HOW TO BUILD A PROGRAMMABLE COMPUTER?

- Computers are very complex systems!
- What basic concepts underlie programmable computers?
- How are they assembled into a usable system?
- This week: a brief tour of how a programmable computer works
 - What components make up a simple computer?
 - What do the instructions look like?
 - How do we implement a computation?

ABSTRACTION HIERARCHY

- Handle complexity in computing systems with an abstraction hierarchy
- A physical medium of computation
 - ...including a way to represent information
 - We generally use semiconductors these days
 - Vacuum tubes, relays, gears, tinker toys and string
- Simple gates for processing signals
 - AND, OR, NOT, XOR, NAND, etc.
 - Implemented in the physical medium
 - Abstracts away the need to think about physics
- Build basic functional units from our gates
 - Counters, arithmetic/logic unit (ALU), memory, multiplexers, decoders, etc.
 - Don't need to think about gates anymore

ABSTRACTION HIERARCHY (2)

- From functional units, can construct a programmable ISA computer!
 - Provides a very simple, limited instruction set
 - We can program it to implement various computations
- This is good, but not very easy to use.
 - Continue extending abstraction hierarchy
- Runtime support to create larger programs:
 - Stacks, heaps, a means to dynamically allocate memory
 - Ability to create subroutines, implement recursion
- Operating systems:
 - Provide many useful abstractions for programming
 - File IO, processes, threads, isolation, virtual memory, networking, etc., etc.

SIGNALS AND GATES

- We are studying *digital* computers...
- Most fundamental piece of information is a bit
 - A single 0 or 1 value
- Logic gates allow us to process bits
- Simple examples:

AND Gate

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

NOT Gate

A	Output
0	1
1	0



MORE COMPLEX GATES

- Construct more complex gates from simple gates
 - In fact, can construct OR from AND and NOT
 - $a \vee b = \neg(\neg a \wedge \neg b)$ (De Morgan's Law)
- Example: XOR, exclusive OR
 - Output is 1 iff exactly one input is 1
 - $A \text{ XOR } B = (A \text{ AND } \neg B) \text{ OR } (\neg A \text{ AND } B)$
- We can also construct XOR entirely from AND and NOT
 - We know how to make OR from AND and NOT...

XOR Gate

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0



LOGIC AND ARITHMETIC

- With these simple gates, can actually implement addition, subtraction, etc.
- Need a way to represent numeric values with bits
- Need circuits that can manipulate these values
- Data Representation:
 - How do we represent various values in our digital computer?
 - Also, how do we represent different *kinds* of values?
 - Integers, decimal values, characters, etc.
- For now: simple unsigned integers

UNSIGNED INTEGERS IN BINARY

- We're used to representing integers as vectors of decimal digits
 - Each digit is 0..9
- Represent unsigned integers as vectors of bits
 - Each digit is 0 or 1
- Also, constrain ourselves to a specific number of bits w for representing values
 - e.g. 4 bits = 1 nibble, 8 bits = 1 byte, 16 bits, 32 bits
 - Obviously limits the range of values we can represent
- A vector of bits \mathbf{x} maps to an unsigned integer:

$$\text{B2U}_w(\mathbf{x}) = \sum_{i=0}^{w-1} x_i 2^i$$

- Individual bits are numbered 0 to $w-1$

UNSIGNED INTEGERS (2)

- $42_{10} = 00101010_2$
 - $0 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
 - $= 32 + 8 + 2$
- Adding integers in base 2 is also straightforward

$$\mathbf{a} + \mathbf{b} = \sum_{i=0}^{w-1} a_i \ 2^i + \sum_{i=0}^{w-1} b_i \ 2^i$$

- Important detail: need to carry in base 2, just like in base 10!
 - Example: $106_{10} + 105_{10}$
 - $= 01101010_2 + 00101001_2$
 - $= 11010011_2 = 211_{10}$
- | | |
|----|----------------|
| C: | 01101000 |
| A: | 01101010 |
| B: | + 01101001 |
| S: | <hr/> 11010011 |

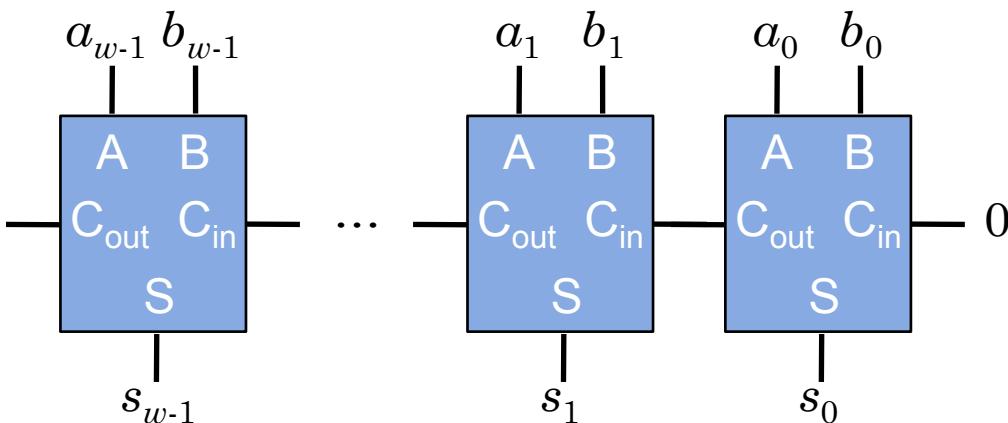
ADDING UNSIGNED INTEGERS

- Simply need to construct necessary machinery for adding bits, using our gates
- Full adder:
 - Takes inputs: A, B, C_{in}
 - Produces outputs: S, C_{out}
- Logic for full adder?
- Sum S is relatively easy:
 - S = A XOR B XOR C_{in}
- Carry-out is more complicated:
 - Carry-out if A and B are 1, or if (A + B) and C_{in} are 1
 - C_{out} = (A AND B) OR (A XOR B) AND C_{in}

A	B	C _{in}	S	C _{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

ADDING UNSIGNED INTEGERS (2)

- Simply need to construct necessary machinery for adding bits, using our gates
- Full adder:
 - Takes inputs: A, B, C_{in}
 - Produces outputs: S, C_{out}
- To add two w -bit unsigned values, hook together w full adders:



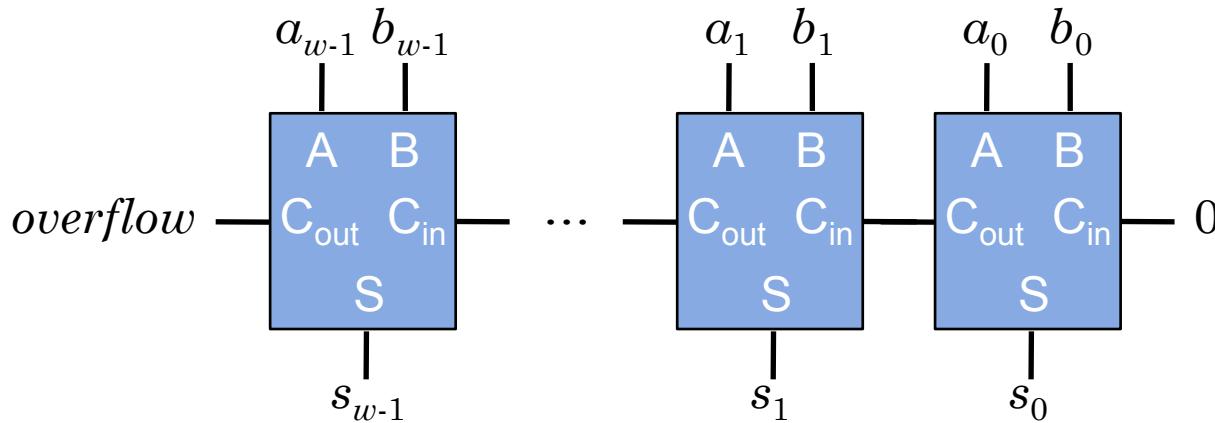
A	B	C _{in}	S	C _{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

RANGES AND OVERFLOW

- For a given w , can only represent unsigned integer values in range $0 .. 2^w - 1$
 - e.g. for $w = 8$, can represent $0 .. 2^8 - 1$, or $0 .. 255$
- What happens if we add these values:
 - $175_{10} + 114_{10} = 10101111_2 + 01110010_2$
 - Result is 289 (100100001_2). This is a problem.
 - Computer adds these values and gets 33 (00100001_2)
- Best case scenario:
 - The computer will tell us when this happens
- Worst case scenario:
 - No way of telling that this problem has occurred

RANGES AND OVERFLOW (2)

- Can the computer tell us there was a problem?



- Yes: topmost carry-out will be 1 when we overflow
- Label topmost carry-out “overflow”
 - When overflow is detected, we can handle the error
- overflow* is a status value
 - It describes additional details of the computation
- One example of how computers can be designed to be more resilient to errors

SIGNED INTEGER REPRESENTATION

- Often need to represent signed values as well
- Most common representation: two's complement
- Most significant bit x_{w-1} becomes the sign bit
 - 0 = positive value
 - 1 = negative value

$$B2T_w(\mathbf{x}) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

- Given w bits, can represent $-2^{w-1} \dots 2^{w-1} - 1$
 - e.g. for $w = 8$, can represent values -128 to +127
- Smallest negative value: $10000000_2 = -128$
- Largest positive value: $01111111_2 = 127$

SIGNED INTEGER REPRESENTATION (2)

- Easy trick for converting an integer into its two's complement representation:
 - Invert the bits, then add one
- Example:
 - Find two's complement representation for -42
 - Unsigned representation for 42 is 00101010_2
 - Invert the bits: 11010101_2
 - Add one: 11010110_2
- Converting back, following $B2T_{w=8}$ function:
 - $= -1 \times 2^7 + 1 \times 2^6 + 1 \times 2^4 + 1 \times 2^2 + 1 \times 2^1$
 - $= -128 + 64 + 16 + 4 + 2$
 - $= -42$

SIGNED INTEGERS AND OVERFLOW

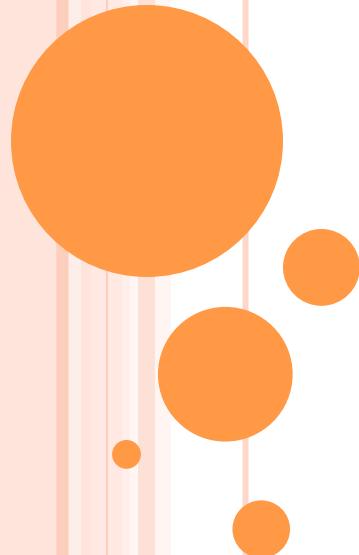
- Rules for overflow flag clearly have to change.
- -1 in two's complement representation ($w = 8$):
 - 11111111
- Adding 1 to this value clearly results in a carry-out!

$$\begin{array}{r} \text{C: } 11111110 \\ \text{A: } 11111111 \\ \text{B: } + 00000001 \\ \hline \text{S: } 00000000 \end{array}$$

- Need to redefine overflow test for signed integers:
 - e.g. for addition, if inputs are same sign, and output is opposite sign, then a signed overflow has occurred

SUMMARY

- Have a data representation for signed and unsigned numbers now...
- Next time, begin discussing basic processor components
 - What they provide
 - How to assemble them into a simple processor
 - How to program the simple processor
- **Your action items:**
 - Enroll in CS24 Moodle course.
 - If using Annenberg Lab, get your CS account set up.
 - If you want, get a copy of CS:APP2e; read Chapter 1.



CS 24: INTRODUCTION TO COMPUTING SYSTEMS

Spring 2014
Lecture 2

LAST TIME

- Began exploring the concepts behind a simple programmable computer
- Construct the computer using Boolean values (a.k.a. “bits”) and logic gates to process them
- Represent unsigned and signed integers as vectors of bits

$$\text{B2U}_w(\mathbf{x}) = \sum_{i=0}^{w-1} x_i 2^i$$

$$\text{B2T}_w(\mathbf{x}) = -x_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

- Briefly explored how to construct more complex computations using gates
 - e.g. unsigned and signed arithmetic

SIGNED INTEGER REPRESENTATION (2)

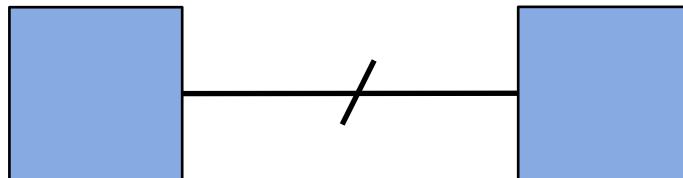
- Easy trick for converting an integer to its two's complement representation:
 - Invert the bits, then add one
- Example:
 - Find two's complement representation for -42
 - Unsigned representation for 42 is 00101010_2
 - Invert the bits: 11010101_2
 - Add one: 11010110_2
- Converting back, following $B2T_{w=8}$ function:
 - $= -1 \times 2^7 + 1 \times 2^6 + 1 \times 2^4 + 1 \times 2^2 + 1 \times 2^1$
 - $= -128 + 64 + 16 + 4 + 2$
 - $= -42$

FUNCTIONAL COMPONENTS OF A SIMPLE PROCESSOR

- Can use our logic gates to construct various components to use in a processor
 - Already saw how to implement addition with logic
- Minimal components for a simple processor:
- Signal Buses
 - Ability to route signals within our processor
- Arithmetic/Logic Unit (ALU)
 - Performs various arithmetic and logical operations on data inputs, based on control inputs
- Memory
 - Addressable locations to store and retrieve values

BUSES

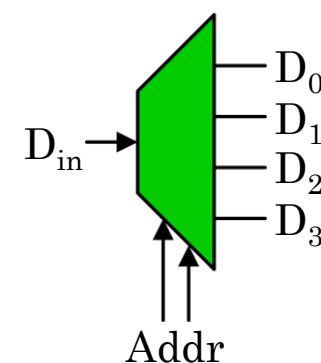
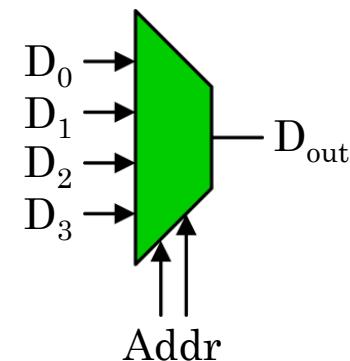
- A bus is a set of wires that transfer signals from one component to another
 - Transmits values of a fixed bit-width, e.g. 32 bits
- Common uses for buses in a computer:
 - Transfer data between CPU and memory
 - Transfer data between CPU and peripherals
- Buses often drawn as a single line with a slash across it



- Individual signals drawn as a line with no slash

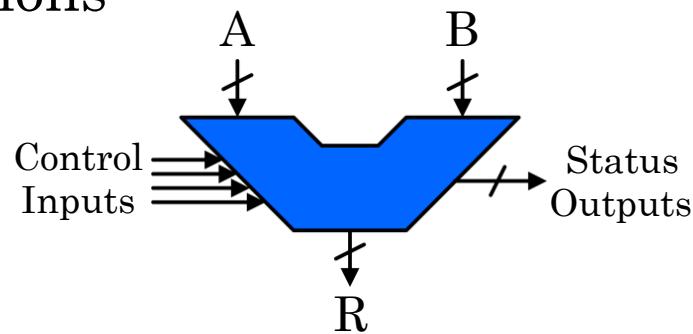
ROUTING BUSES

- Multiplexers and demultiplexers (decoders) are used to route buses between multiple components
- Example: a 4-input multiplexer (MUX)
 - Has two address inputs
 - Address selects one of 4 data inputs
 - Corresponding data input is fed to the data output
- A 4-input demultiplexer (DEMUX)
 - Again, two address inputs
 - Address selects one of 4 data outputs
 - Single data input fed to corresponding data output



ARITHMETIC/LOGIC UNIT

- A component that can perform various arithmetic and logic functions
- Symbol:



- Given two w -bit inputs and a set of control inputs
 - Control inputs specify the operation to perform
- Produces a w -bit result, and status outputs
 - Example status outputs:
 - sign flag (topmost bit of R)
 - carry-out flag (unsigned overflow)
 - zero flag (is R == 0?)
 - overflow flag (signed overflow)

EXAMPLE ALU OPERATIONS

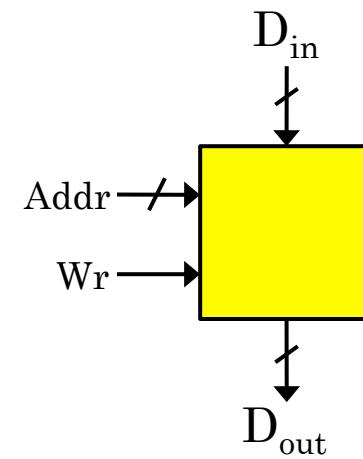
- Control signals specify what operation to perform
- Example: for our contrived ALU

Control	Operation	A	B
0001	ADD	A	B
0011	SUB	A	B
0100	NEG	A	
1000	AND	A	B
1001	OR	A	B
1010	XOR	A	B
1011	INV	A	
1100	SHL	A	
1110	SHR	A	

- By feeding appropriate control and data signals to ALU in sequence, can perform computations
- Some operations require only one argument
 - Second argument ignored

MEMORY

- Need a component to store both instructions and data to feed to the ALU
- Memory:
 - An array of linearly addressable locations
 - Each location has its own address
 - Each location can hold a single w -bit value
- Inputs and outputs:
 - Address of location to read or write
 - Read/Write control signal
 - Data-input bus
 - Data-output bus

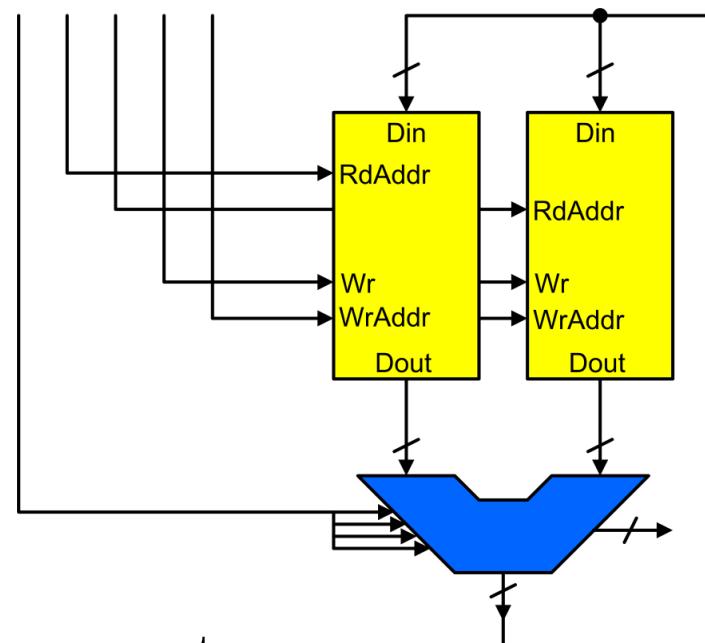


ASIDE: CPU COMPONENTS AND GATES

- It is a big claim that we can construct all of these components entirely from logic gates...
- Unfortunately, beyond the scope of CS24 to explore all the ways such components can be constructed, different approaches, etc.
- If you are curious how these things work, see the primer on the CS24 Moodle
 - Shows some *basic* ways these components can be constructed
- **Don't need to know this material in depth!**
 - For CS24, really only need to understand the basics of how to implement logic equations with gates

ASSEMBLING THE COMPUTER

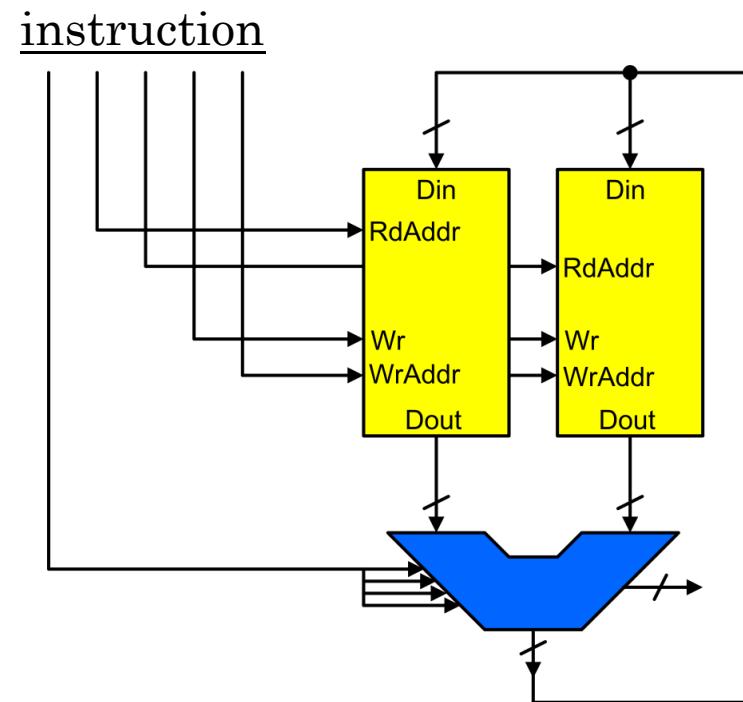
- Hook these components together, like this:



- Simplifications in our computer:
 - Two memory banks; identical copies of each other
 - Don't care about ALU status outputs

INSTRUCTING THE COMPUTER

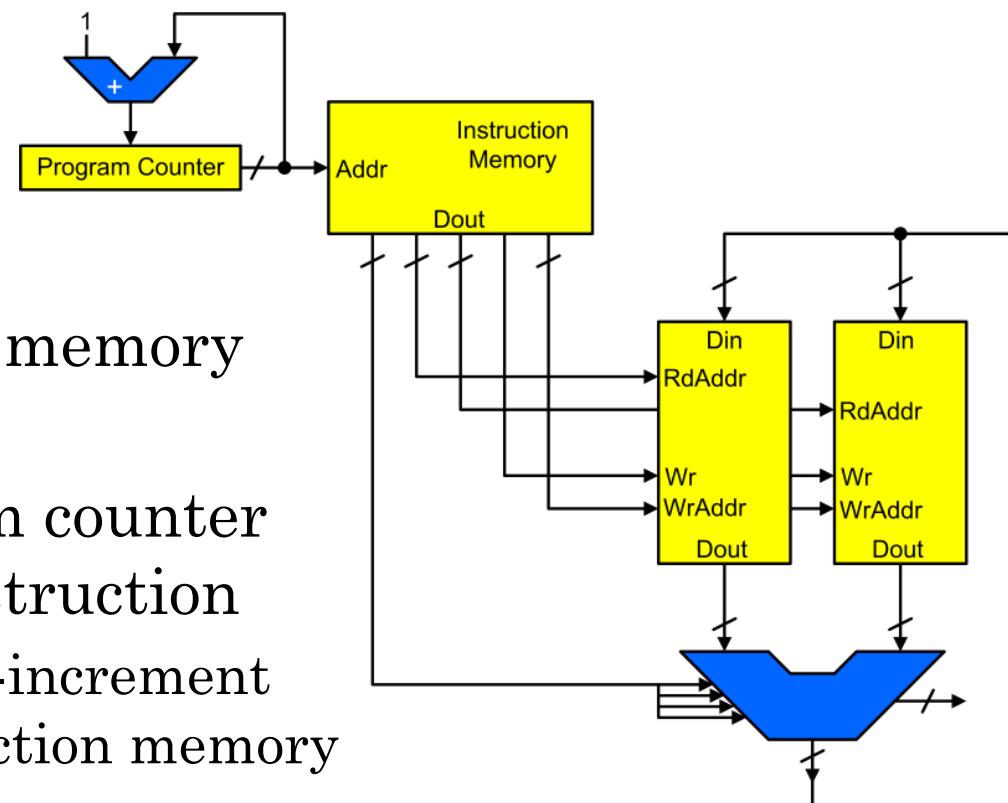
- This set of inputs forms an instruction
- Consists of:
 - The operation the ALU should perform
 - Addresses of two input values
 - Whether result should be stored
 - If so, what address to store the result at



- To program our computer:
 - Devise a set of instructions to implement our desired computation

INSTRUCTING THE COMPUTER (2)

- Need a way to feed instructions to our computer



- Add an instruction memory to our system
- Also, add a program counter to track current instruction
 - Configured to auto-increment through the instruction memory

WRITING A PROGRAM

- Instructions for the processor are very limited
 - Can only compute one value, from one or two values
- Usually can't implement a program in only one instruction
- Instead:
 - String together a sequence of instructions to implement the computation
 - Instructions will communicate via memory locations
- Computation we will implement:
 - $C = (A - 2B) \& 00001111_2$
 - Given inputs A and B
 - Multiply B by 2, subtract result from A, then bitwise-AND with a mask

IMPLEMENTING OUR COMPUTATION (1)

- Computation: $C = (A - 2B) \& 00001111_2$
- Step 1: Assign locations for inputs and outputs
- Inputs:
 - A and B (obvious)
 - Also, our mask: 00001111_2
 - Program needs to include our constants, too
- Output:
 - C
- Givens:
 - Our memory has 8 locations
 - Memory addresses are 3 bits wide
 - Data values are 8 bits wide ($w = 8$)

ASSIGNING DATA LOCATIONS

- Locations for our initial and final data values:

Address	Value
0	A
1	B
2	
3	
4	00001111_2
5	
6	
7	C

IMPLEMENTING OUR COMPUTATION (2)

- Step 2: decompose our program into instructions the processor can actually handle
- Program:
 - $C = (A - 2B) \& 00001111_2$
- Need to know processor's operations for this step.
- Steps:
 - Perform $2B$ first, as $B + B$
 - Then, subtract previous result from A
 - Finally, bitwise-AND this with mask to produce C

Control	Operation		
0001	ADD	A	B
0011	SUB	A	B
0100	NEG	A	
1000	AND	A	B
1001	OR	A	B
1010	XOR	A	B
1011	INV	A	
1100	SHL	A	
1110	SHR	A	

IMPLEMENTING OUR COMPUTATION (3)

- Step 3: need to assign locations to these intermediate values!
- Result of $B + B =$ location 2
- Result of $A - 2B =$ location 3
- Result of bitwise-AND stored in location 7
 - This is our result

Address	Value
0	A
1	B
2	2B
3	A - 2B
4	0000111_2
5	
6	
7	C

IMPLEMENTING OUR COMPUTATION (4)

- Step 4: Translate our program into instructions!
- Need to know form of instructions:
 - Operation Rd1Addr Rd2Addr Wr WrAddr
- Also need our memory layout and operation codes

Control	Operation		
0001	ADD	A	B
0011	SUB	A	B
0100	NEG	A	
1000	AND	A	B
1001	OR	A	B
1010	XOR	A	B
1011	INV	A	
1100	SHL	A	
1110	SHR	A	

Address	Value
0	A
1	B
2	2B
3	A - 2B
4	00001111 ₂
5	
6	
7	C

IMPLEMENTING OUR COMPUTATION (5)

Control	Operation	A	B
0001	ADD	A	B
0011	SUB	A	B
0100	NEG	A	
1000	AND	A	B
1001	OR	A	B
1010	XOR	A	B
1011	INV	A	
1100	SHL	A	
1110	SHR	A	

Address	Value
0	A
1	B
2	2B
3	A - 2B
4	00001111 ₂
5	
6	
7	C

- Operation Rd1Addr Rd2Addr Wr WrAddr
- Writing our program:
 - Slot 2 = 2B 000: 0001 001 001 1 010
 - Slot 3 = A - 2B 001: 0011 000 010 1 011
 - Slot 7 = (...) & mask 010: 1000 011 100 1 111

RUNNING OUR PROGRAM

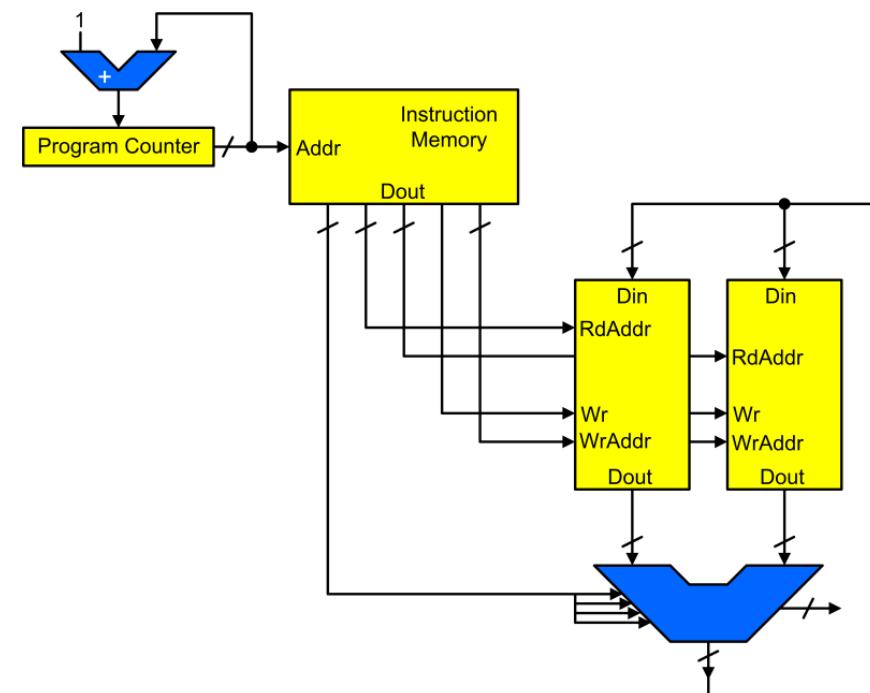
- To run our program:
 - Load instructions into instruction memory
 - Load initial data into data memory
 - Start program counter at 0
- Each instruction executes in sequence, updating memory locations
 - Uses results of previous instructions
- State of our computer:
 - Instruction memory
 - Data memory
 - Program counter

RUNNING OUR PROGRAM: INITIAL STATE

- Instruction memory:
000: 0001 001 001 1 010
001: 0011 000 010 1 011
010: 1000 011 100 1 111

- Data memory:

0: A
1: B
2: ???
3: ???
4: 00001111_2
5: ???
6: ???
7: ???



STEP 1: SLOT 2 = B + B

- Instruction memory:

000: 0001 001 001 1 010

001: 0011 000 010 1 011

010: 1000 011 100 1 111

- Data memory:

0: A

1: B

2: **$2B = B + B$**

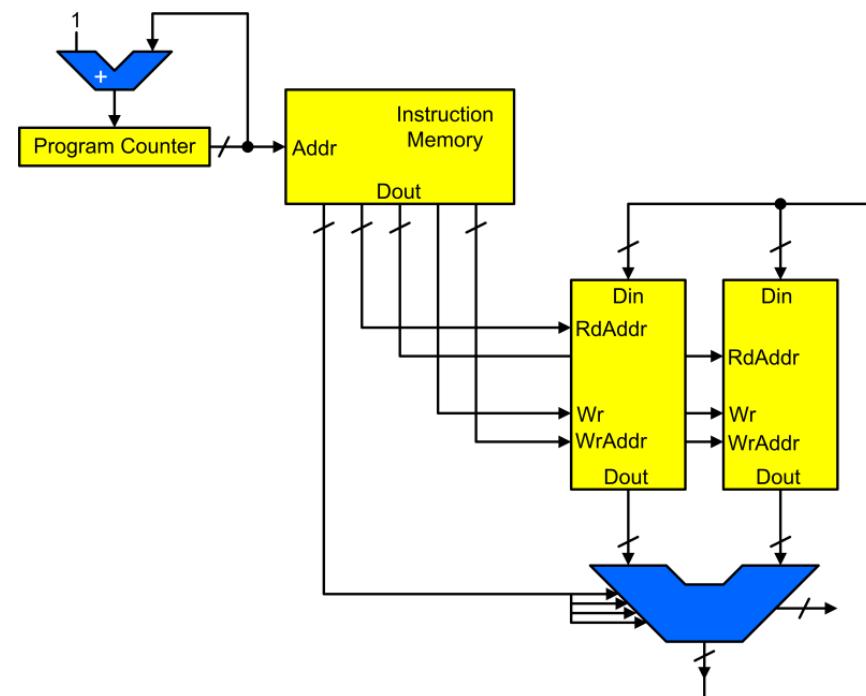
3: ???

4: 00001111_2

5: ???

6: ???

7: ???



Program Counter: 000

STEP 1: UPDATE PROGRAM COUNTER

- Instruction memory:

000: 0001 001 001 1 010

001: 0011 000 010 1 011

010: 1000 011 100 1 111

- Data memory:

0: A

1: B

2: 2B

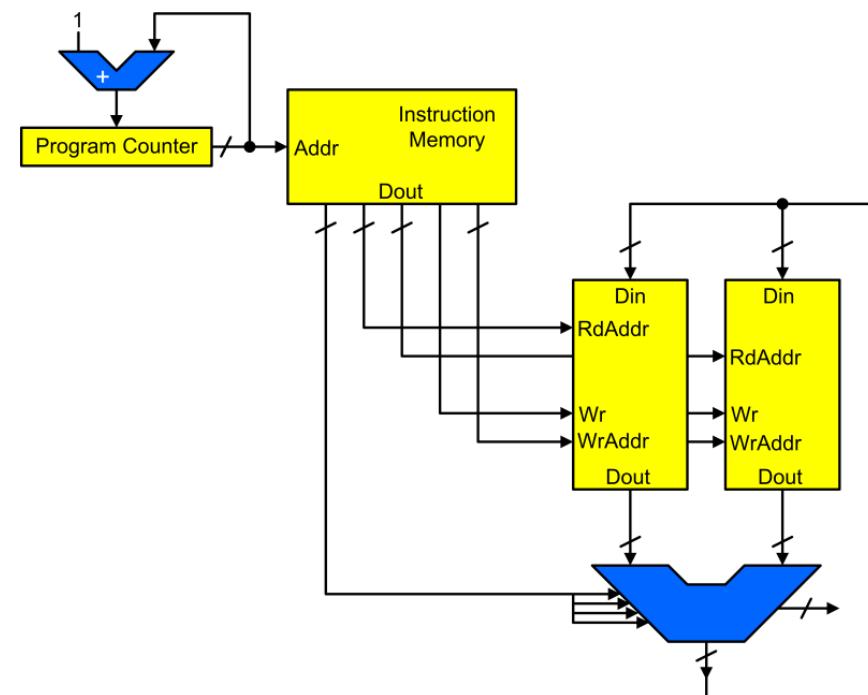
3: ???

4: 00001111_2

5: ???

6: ???

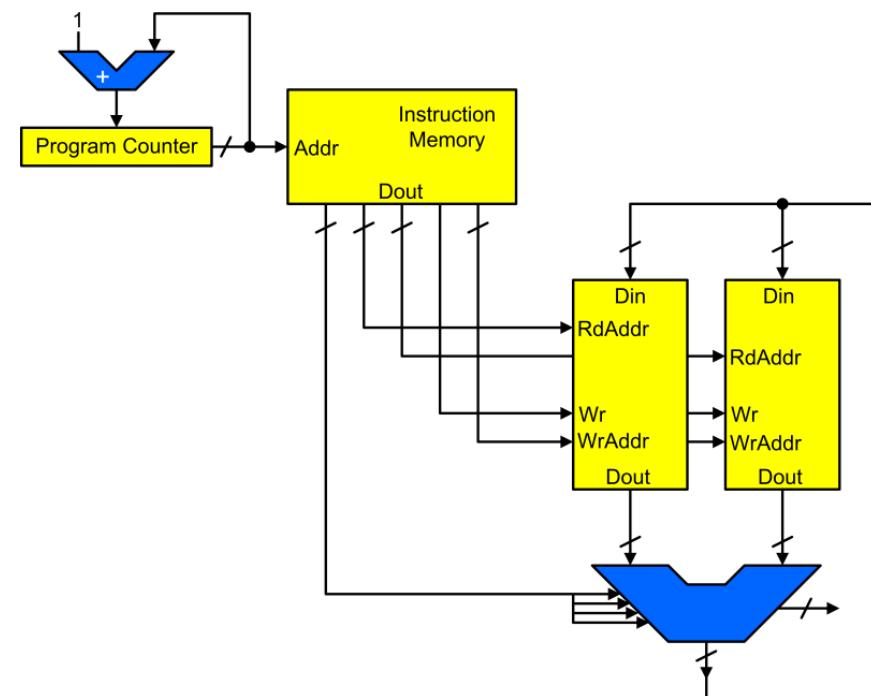
7: ???



Program Counter: 001

STEP 2: SUBTRACT 2B FROM A

- Instruction memory:
000: 0001 001 001 1 010
001: 0011 000 010 1 011
010: 1000 011 100 1 111



Program Counter: 001

- Data memory:
0: A
1: B
2: 2B
3: **A – 2B**
4: 0000111_2
5: ???
6: ???
7: ???

STEP 2: UPDATE PROGRAM COUNTER

- Instruction memory:

000: 0001 001 001 1 010

001: 0011 000 010 1 011

010: 1000 011 100 1 111

- Data memory:

0: A

1: B

2: 2B

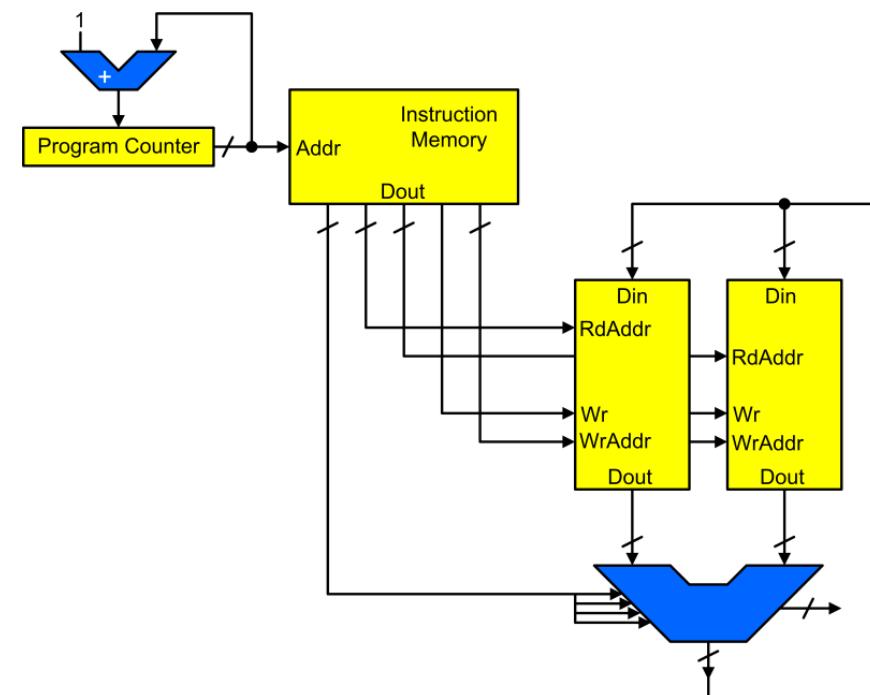
3: A – 2B

4: 00001111_2

5: ???

6: ???

7: ???

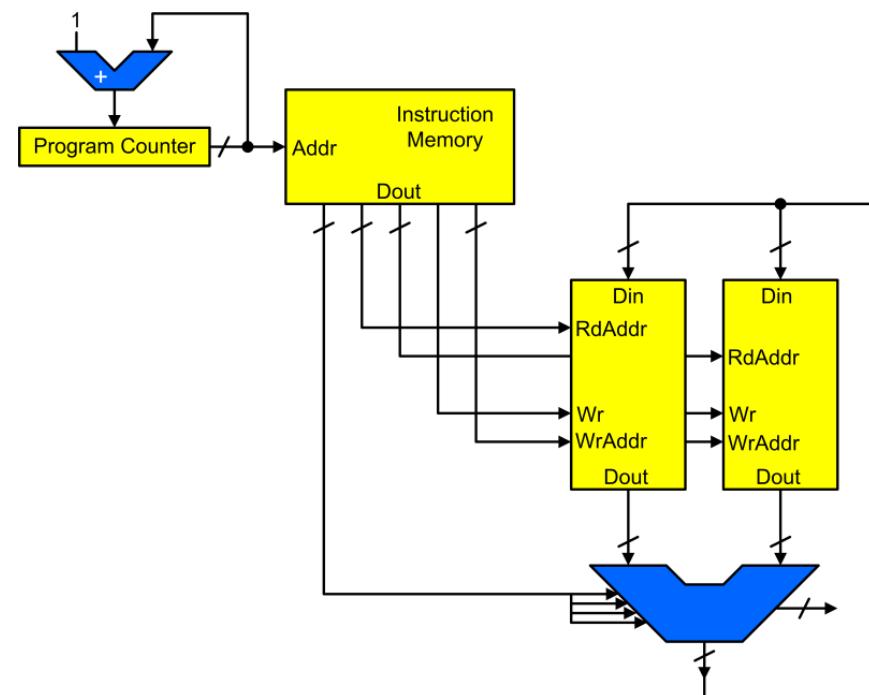


STEP 3: $C = (A - 2B) \& 00001111_2$

- Instruction memory:
000: 0001 001 001 1 010
001: 0011 000 010 1 011
010: 1000 011 100 1 111

- Data memory:

0: A
1: B
2: 2B
3: $A - 2B$
4: 00001111_2
5: ???
6: ???
7: $(A - 2B) \& 00001111_2$



Program Counter: 010

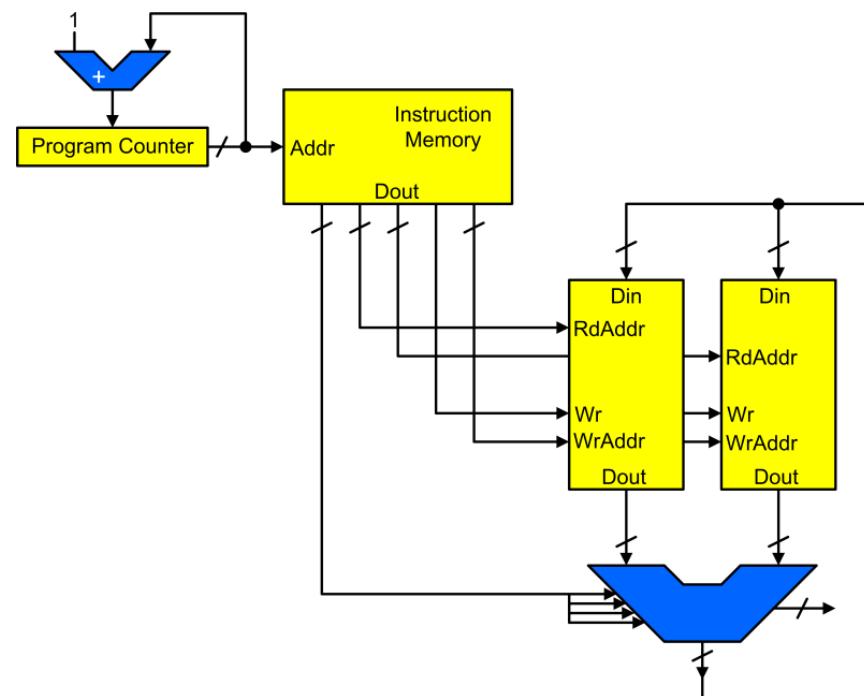
RUNNING OUR PROGRAM: FINAL RESULT

- Instruction memory:

000: 0001 001 001 1 010
001: 0011 000 010 1 011
010: 1000 011 100 1 111

- Data memory:

0: A
1: B
2: 2B
3: A – 2B
4: 00001111_2
5: ???
6: ???
7: $(A - 2B) \& 00001111_2$



A PROGRAMMABLE COMPUTER!

- Using our basic functional components, we are able to build a simple programmable computer!
- Implemented a computation using our processor's instruction set:
 1. Assigned memory locations to inputs and output
 2. Decomposed computation into processor instructions
 3. Assigned memory locations for intermediate values
 4. Encoded sequence of instructions for our program
- By feeding instructions to computer in sequence, we can perform our computation
 - Individual instructions communicate by reading and writing various memory locations

MACHINE CODE, ASSEMBLY LANGUAGE

- Our program:

000: 0001 001 001 1 010

001: 0011 000 010 1 011

010: 1000 011 100 1 111

- This is called machine code

- The actual data values that comprise the program
- Hard to read and write!

- Humans normally use assembly language

- A more human-readable language that is translated into machine code using an assembler

ADD R1, R1, R2 # R2 = 2B

SUB R0, R2, R3 # R3 = A – 2B

AND R3, R4, R7 # C = R3 & 00001111

- Allows human-readable names, operations, comments

C LOGICAL AND BITWISE OPERATIONS

- Before going forward, need to review what C offers for logical and bitwise operations
- C uses integers to represent Boolean values
 - 0 = false; any nonzero value = true
- Logical Boolean operators:
 - Logical AND: **a && b**
 - Logical OR: **a || b**
 - Logical NOT: **!a**
 - Result is 1 if true, 0 if false
- **&&** and **||** are short-circuit operators
 - Evaluated left-to-right
 - For **&&**, if LHS is false then RHS is not evaluated
 - For **||**, if LHS is true then RHS is not evaluated

C LOGICAL AND BITWISE OPERATIONS (2)

- C also has many bit-manipulation operations
- Given $a = 00010100_2 (20_{10})$, $b = 00110010_2 (50_{10})$
 - $a \& b = 00010000$ Bitwise AND
 - $a | b = 00110110$ Bitwise OR
 - $\sim a = 11101011$ Bitwise negation (invert)
 - $a ^ b = 00100110$ Bitwise XOR
- Note:
 - C has no way of specifying base-2 literals
 - Normal approach: use *hexadecimal* literals instead
- Hexadecimal: base-16 numbers
 - Digits are 0..9, A..F (or a..f, makes no difference)
 - A = 10, B = 11, ..., F = 15

HEXADECIMAL VALUES AND BIT-MASKS

- Example: **0x0F** is a hexadecimal literal in C
 - Each digit of a hexadecimal value represents 4 bits – a compact, simple way to write bit-fields
 - **0x0F** = 0000 1111 (also **0x0f**)
 - **0x03C7** = 0000 0011 1100 0111 (also **0x03c7**)
- Use bitwise AND to mask out or clear specific bits
 - **a & 0x0F**
 - Clears high nibble of **a**; retains low nibble of **a**
- Use bitwise OR to set specific bits
 - **a = a | 0x28**
 - Sets bits 3 and 5 of value in **a** (**0x28** = 0010 1000)
 - Other bits in **a** remain unchanged
- Use bitwise XOR to toggle specific bits
 - **a = a ^ 0x28**
 - Toggles bits 3 and 5 of value in **a**; other bits are left unchanged

C BIT-SHIFTING OPERATIONS

- C also includes bit-shifting operations
- Shift bits in **a** left by **n** bits: **a << n**
 - New bits on right are 0
 - Shifting left by n bits is identical to multiplying by 2^n
`a = 42; /* a = 00101010 = 42 */
a = a << 1; /* a = 01010100 = 84 */`
- Shift bits in **a** right by **n** bits: **a >> n**
 - Shifting right by n bits is identical to dividing by 2^n
- Question: What should new bits on left be?
 - Depends on whether **a** is signed or unsigned!
`a = -24; /* Two's complement: 11101000 */
a = a >> 1; /* Should be -12 (11110100) now */`
 - Leftmost bit represents sign
 - Preserve sign by using same value as original sign-bit

ARITHMETIC VS. LOGICAL SHIFT-RIGHT

- Distinguish between arithmetic shift-right and logical (i.e. bitwise) shift-right
 - Arithmetic shift-right preserves the value's sign
 - Logical shift-right always adds 0-bits to left of value
- Some languages make this distinction
 - Java: `>>` is arithmetic, `>>>` is logical
 - IA32 assembly: **SAR** is arithmetic, **SHR** is logical
- In C:
 - If argument is signed, shift-right is arithmetic

```
char a = -24;          /*      -24 = 11101000 */  
printf("%d", a >> 1); /* Prints -12 = 11110100 */
```
 - If argument is unsigned, shift-right is logical

```
/* Prints 116 = 01110100; topmost bit is 0 */  
printf("%d", (unsigned char) a >> 1);
```

BIT-SHIFTS AND MASKS

- Can use bit-shifts with masks to extract sub-byte values
- `(a >> 4) & 0x0F`
 - Retrieves high nibble of **a**
- Does it matter if **a** is signed or unsigned?
 - Nope. We chop off the sign bit after we shift.

MULTIPLICATION?

- Our processor's instruction set:
- Hmm, no multiply instruction.
- No problem; implement multiply with addition and shifting

$$\text{mul}_w(\mathbf{a}, \mathbf{b}) = \sum_{i=0}^{w-1} a_i \mathbf{b} 2^i$$

```
int mul(int a, int b) {
    int p = 0;
    while (a != 0) {
        if (a & 1 == 1)
            p = p + b;
        a = a >> 1;
        b = b << 1;
    }
    return p;
}
```

Control	Operation		
0001	ADD	A	B
0011	SUB	A	B
0100	NEG	A	
1000	AND	A	B
1001	OR	A	B
1010	XOR	A	B
1011	INV	A	
1100	SHL	A	
1110	SHR	A	

MULTIPLICATION ???

- Our multiply program:

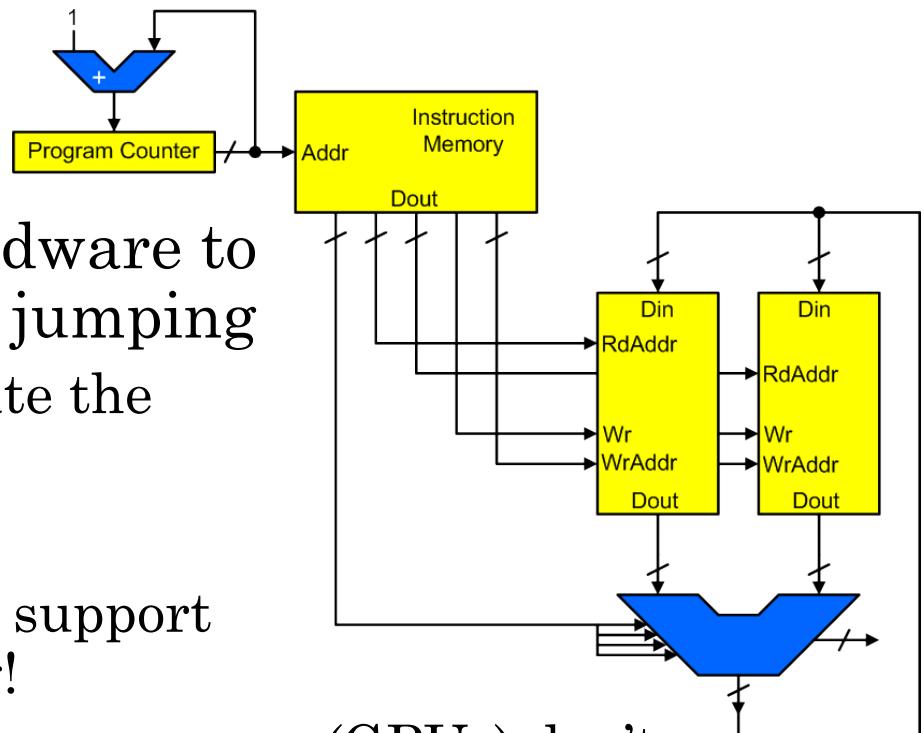
```
int mul(int a, int b) {  
    int res = 0;  
    while (a != 0) {  
        if (a & 1 == 1)  
            p = p + b;  
        a = a >> 1;  
        b = b << 1;  
    }  
    return p;  
}
```

Control	Operation	
0001	ADD	A B
0011	SUB	A B
0100	NEG	A
1000	AND	A B
1001	OR	A B
1010	XOR	A B
1011	INV	A
1100	SHL	A
1110	SHR	A

- Can we write a program to execute this code?
 - NO! 😞
 - Our processor doesn't support any branching or jumping operations

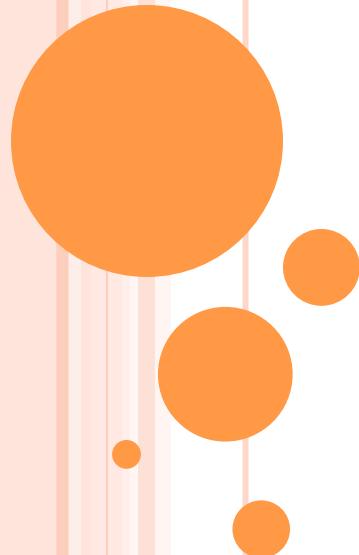
BRANCHING SUPPORT

- Our current processor architecture *can't* support branching or jumping!
 - Can only execute code in sequential order
- Need to extend the hardware to support branching and jumping
 - Need to be able to update the Program Counter field
- Note:
 - Not always essential to support branching and jumping!
 - Most dedicated graphics processors (GPUs) don't support looping or branching at all
 - However, is essential for a general-purpose processor



SUMMARY

- We designed a simple programmable computer!
 - Assembled functional components so we can perform a variety of simple computations
 - Feed instructions into our processor in sequence, from instruction memory
 - Instructions communicate by reading and writing various memory locations
- But, our computer has substantial limitations...
 - Can't even implement a simple loop yet. 😞
 - Need to extend our processor to support branching
- Also, our computer only has 8 bytes of memory
 - Need to examine impact of increasing memory size



CS 24: INTRODUCTION TO COMPUTING SYSTEMS

Spring 2015
Lecture 3

LAST TIME

- Basic components of processors:
 - Buses, multiplexers, demultiplexers
 - Arithmetic/Logic Unit (ALU)
 - Addressable memory
- Assembled components into a simple processor
 - Can perform a wide range of operations, but all very simple
 - Need to string together a sequence of instructions, which communicate via memory locations
- Implemented a computation on this processor:
 - Assigned memory locations for inputs, outputs, constants
 - Decomposed the computation into steps the processor could actually handle
 - Assigned locations to intermediate values
 - Translated the program into machine-code instructions

MULTIPLICATION...

- Then, we wanted to implement multiplication:

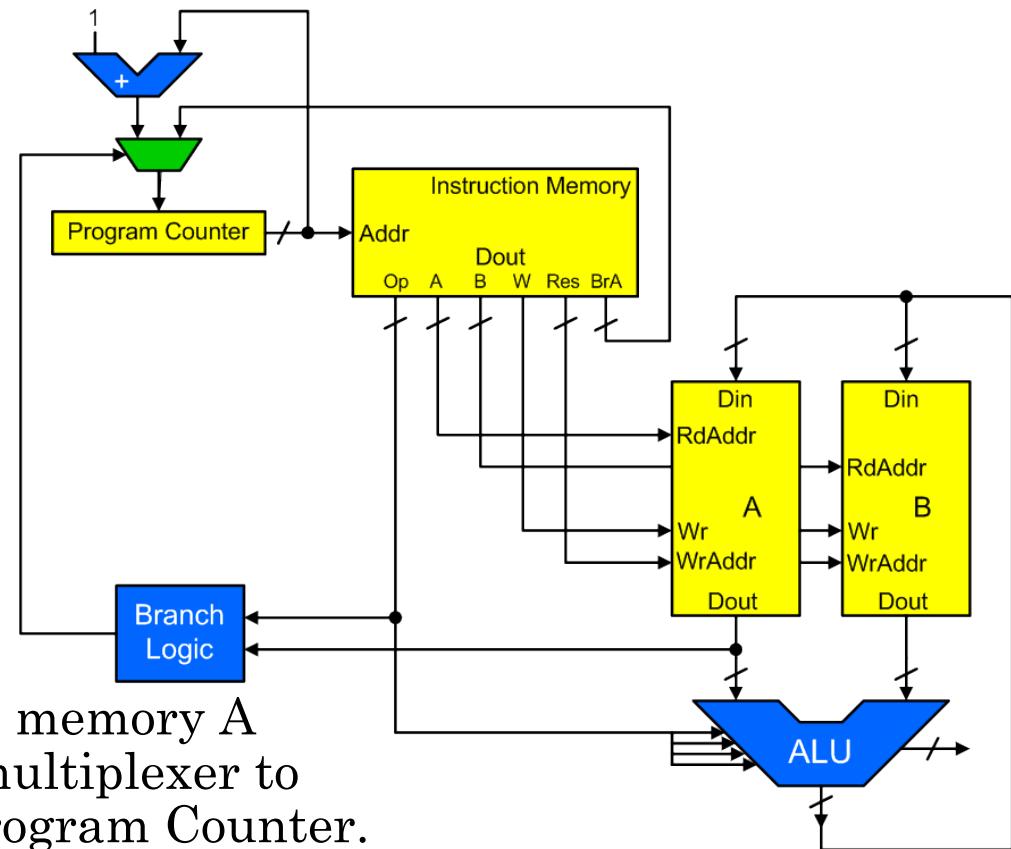
```
int mul(int a, int b) {  
    int res = 0;  
    while (a != 0) {  
        if (a & 1 == 1)  
            p = p + b;  
        a = a >> 1;  
        b = b << 1;  
    }  
    return p;  
}
```

Control	Operation	
0001	ADD	A B
0011	SUB	A B
0100	NEG	A
1000	AND	A B
1001	OR	A B
1010	XOR	A B
1011	INV	A
1100	SHL	A
1110	SHR	A

- But, we couldn't write this program:
 - Our processor doesn't support branching operations!

UPDATE OUR ISA AND PROCESSOR

- Add a new instruction: BRZ A, Addr (Branch if Zero)
 - If value in slot A is 0, change Prog Ctr to address Addr
- New logic to support this instruction:



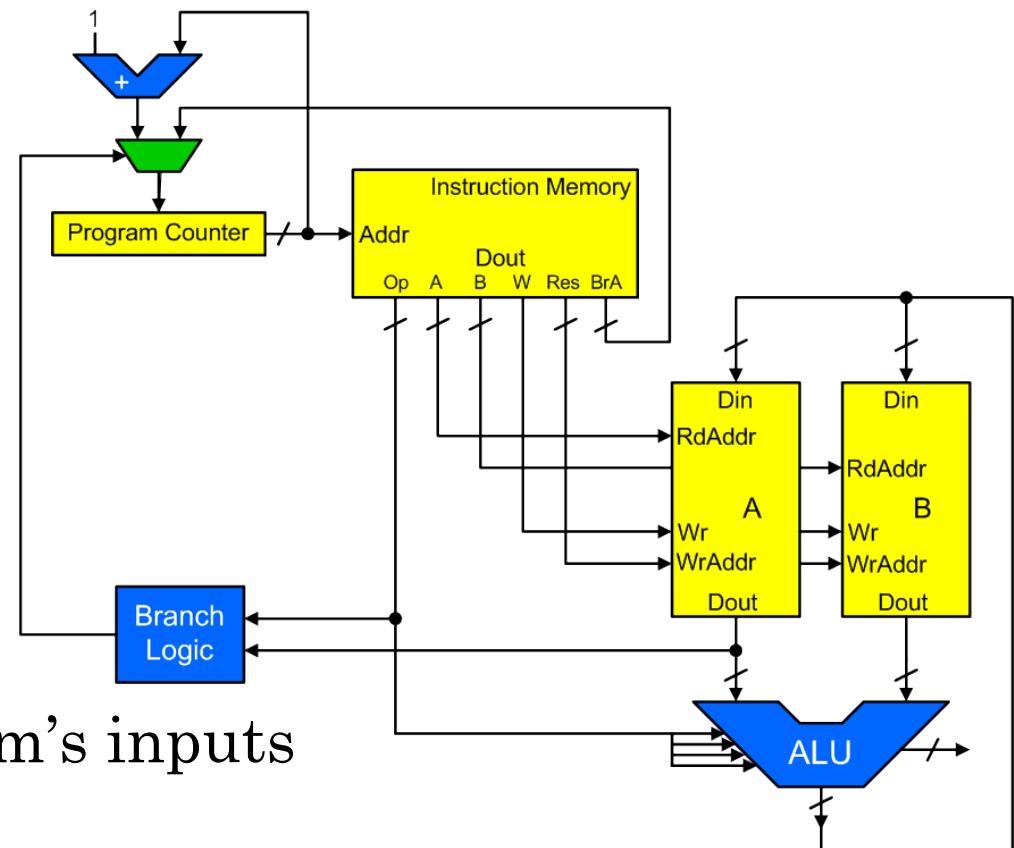
- Branch Logic:
 - If opcode is BRZ, and memory A outputs 0, then tell multiplexer to load Addr into the Program Counter.

UPDATED PROCESSOR

- With updated processor, can reuse instructions by creating loops in our programs

- Perform many computations with just a few instructions

- Can now implement computations where number of steps is dependent on program's inputs



BACK TO MULTIPLICATION

- Should have enough capability now to encode our program

```
int mul(int a, int b) {  
    int p = 0;  
    while (a != 0) {  
        if (a & 1 == 1)  
            p = p + b;  
        a = a >> 1;  
        b = b << 1;  
    }  
    return p;  
}
```

Control	Operation	
0001	ADD	A B
0011	SUB	A B
0100	NEG	A
0111	BRZ	A Addr
1000	AND	A B
1001	OR	A B
1010	XOR	A B
1011	INV	A
1100	SHL	A
1110	SHR	A

- Coding is more complex now!
 - Need to plan out our loops...
 - Need to know the *addresses* to jump to!

WRITING OUR PROGRAM

- Use same general process as before
- Step 1: identify inputs and outputs

```
int mul(int a, int b) {  
    int p = 0;  
    while (a != 0) {  
        if (a & 1 == 1)  
            p = p + b;  
        a = a >> 1;  
        b = b << 1;  
    }  
    return p;  
}
```

- Inputs: A, B, some constant(s)
- Outputs: P

WRITING OUR PROGRAM (2)

- Step 2:
 - Decompose program into processor instructions.
- This step will be much more involved now:
 - May need quite a few temporary values
 - (Don't know how many though...)
 - Need to keep track of various addresses for branching
 - (Also don't know how many...)
- Strategy: use names for unknown values
 - Put in placeholders for temp-variables, places to jump
 - Write the code you *do* understand...
 - Once code is written, replace names with addresses

WRITING OUR PROGRAM (3)

- Code:

```
int p = 0;  
while (a != 0) {  
    if (a & 1 == 1)  
        p = p + b;  
    a = a >> 1;  
    b = b << 1;  
}
```

Control	Operation	
0001	ADD	A B
0011	SUB	A B
0100	NEG	A
0111	BRZ	A Addr
1000	AND	A B
1001	OR	A B
1010	XOR	A B
1011	INV	A
1100	SHL	A
1110	SHR	A

- First step: Set P = 0

- Options?

- Subtract P from itself: **SUB P, P, P**

- XOR P with itself: **XOR P, P, P**

- XOR option appears frequently in assembly code

WRITING OUR PROGRAM (4)

- Code:

```
int p = 0;  
while (a != 0) {  
    if (a & 1 == 1)  
        p = p + b;  
    a = a >> 1;  
    b = b << 1;  
}
```

- Steps:

```
XOR P, P, P  
WHILE:  
...
```

- Will certainly need to go back to start of loop
- Add a *label* to the code
 - Use the label in branching instructions
 - At end of translation, replace label with an actual address
- We already do this with our variables...

WRITING OUR PROGRAM (5)

- Code:

```
int p = 0;  
while (a != 0) {  
    if (a & 1 == 1)  
        p = p + b;  
    a = a >> 1;  
    b = b << 1;  
}
```

- Steps:

```
XOR P, P, P  
WHILE:  
BRZ A, DONE  
...  
DONE:
```

- Don't know what will go inside loop yet, but we know we will need to exit it when finished!
- Add a **DONE** label, and use **BRZ** instruction to test A.

WRITING OUR PROGRAM (6)

- Code:

```
int p = 0;  
while (a != 0) {  
    if (a & 1 == 1)  
        p = p + b;  
  
    a = a >> 1;  
    b = b << 1;  
}
```

- Steps:

```
XOR P, P, P  
WHILE:  
    BRZ A, DONE  
    AND A, 1, Tmp  
    BRZ Tmp, SKIP
```

...

SKIP:

...

DONE:

- Compute A & 1, then store in a temporary location
- Then, use branching instruction to skip body of if statement
 - (Can use **BRZ** since Tmp will be 0 or 1 here...)

WRITING OUR PROGRAM (7)

- Code:

```
int p = 0;  
while (a != 0) {  
  
    if (a & 1 == 1)  
  
        p = p + b;  
  
    a = a >> 1;  
    b = b << 1;  
}
```

- Steps:

```
XOR P, P, P  
WHILE:  
    BRZ A, DONE  
    AND A, 1, Tmp  
    BRZ Tmp, SKIP  
    ADD P, B, P  
SKIP:  
    SHR A, A  
    SHL B, B  
    BRZ 0, WHILE  
DONE:
```

- Remaining operations are mostly easy to write...
- Need an *unconditional* branching operation
 - Just use **BRZ** with a value of 0

CONTROL-FLOW IN OUR PROGRAM

- Code:

```
int p = 0;  
while (a != 0) {  
  
    if (a & 1 == 1)  
  
        p = p + b;  
  
    a = a >> 1;  
    b = b << 1;  
}
```

- Steps:

```
XOR P, P, P  
WHILE:  
    BRZ A, DONE  
    AND A, 1, Tmp  
    BRZ Tmp, SKIP  
    ADD P, B, P  
SKIP:  
    SHR A, A  
    SHL B, B  
    BRZ 0, WHILE  
DONE:
```

- Manually implemented our **while** loop:
 - Test condition at start; exit if result is false
 - At end of loop, unconditionally return to start

CONTROL-FLOW IN OUR PROGRAM (2)

- Code:

```
int p = 0;  
while (a != 0) {  
  
    if (a & 1 == 1)  
  
        p = p + b;  
  
    a = a >> 1;  
    b = b << 1;  
}
```

- Steps:

```
XOR P, P, P  
WHILE:  
    BRZ A, DONE  
    AND A, 1, Tmp  
    BRZ Tmp, SKIP  
    ADD P, B, P  
SKIP:  
    SHR A, A  
    SHL B, B  
    BRZ 0, WHILE  
DONE:
```



- Also manually implemented the **if** statement:
 - Tested inverse of the condition, and skip over if-body if result is false

FINISHING OUR PROGRAM

- Our assembly code so far:
- Assign locations to values:

Address	Value
0	A
1	B
2	Tmp
3	1
4	0
5	
6	
7	P

```
XOR P, P, P
WHILE:
    BRZ A, DONE
    AND A, 1, Tmp
    BRZ Tmp, SKIP
    ADD P, B, P
SKIP:
    SHR A, A
    SHL B, B
    BRZ 0, WHILE
DONE:
```

- Note: two constants to include as well...

FINISHING OUR PROGRAM (2)

- Update our code with the memory addresses:

Address	Value
0	A
1	B
2	Tmp
3	1
4	0
5	
6	
7	P

```
XOR 111, 111, 111
WHILE:
    BRZ 000, DONE
    AND 000, 011, 010
    BRZ 010, SKIP
    ADD 111, 001, 111
SKIP:
    SHR 000, 000
    SHL 001, 001
    BRZ 100, WHILE
DONE:
```

FINISHING OUR PROGRAM (3)

- Now to figure out the instruction addresses

WHILE = address 1

SKIP = address 5

DONE = address 8

```
0: XOR 111, 111, 111
WHILE:
    1: BRZ 000, DONE
    2: AND 000, 011, 010
    3: BRZ 010, SKIP
    4: ADD 111, 001, 111
SKIP:
    5: SHR 000, 000
    6: SHL 001, 001
    7: BRZ 100, WHILE
DONE:
    8:
```

FINISHING OUR PROGRAM (3)

- Update code with the instruction addresses

WHILE = address 1

SKIP = address 5

DONE = address 8

```
0: XOR 111, 111, 111
1: BRZ 000, 1000
2: AND 000, 011, 010
3: BRZ 010, 0101
4: ADD 111, 001, 111
5: SHR 000, 000
6: SHL 001, 001
7: BRZ 100, 0001
8:
```

FINISHING OUR PROGRAM (4)

- Finally, encode each instruction into machine code

Control	Operation		
0001	ADD	A	B
0011	SUB	A	B
0100	NEG	A	
0111	BRZ	A	Addr
1000	AND	A	B
1001	OR	A	B
1010	XOR	A	B
1011	INV	A	
1100	SHL	A	
1110	SHR	A	

XOR 111, 111, 111	1010 111 111 111 0000
BRZ 000, 1000	0111 000 000 000 1000
AND 000, 011, 010	1000 000 011 010 0000
BRZ 010, 0101	0111 010 000 000 0101
ADD 111, 001, 111	0001 111 001 111 0000
SHR 000, 000	1110 000 000 000 0000
SHL 001, 001	1100 001 000 001 0000
BRZ 100, 0001	0111 100 000 000 0001

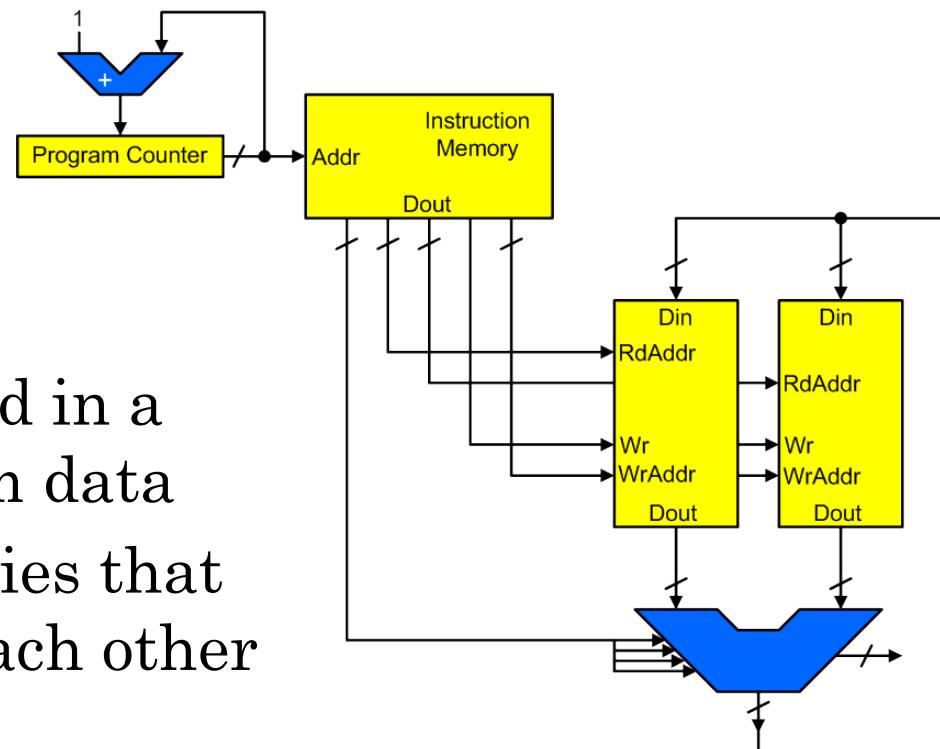
THE FINAL PROGRAM

- Our final machine code:
 - Unused/ignored bits are grayed out
- 136 bits; 44 bits unused
 - 30% of our program's space is unused!
- This instruction encoding is not very efficient
- Many processors employ variable-length instruction encodings to reduce program size
 - Particularly common in CISC processors
 - Increases complexity in instruction processing logic

1010	111	111	111	0000
0111	000	000	000	1000
1000	000	011	010	0000
0111	010	000	000	0101
0001	111	001	111	0000
1110	000	000	000	0000
1100	001	000	001	0000
0111	100	000	000	0001

PROCESSOR MEMORIES

- Our processor has a curious memory layout
 - Doesn't seem much like our modern computers...



MEMORY ARCHITECTURES

- Several different memory architectures have been employed in computer systems
- Harvard Architecture:
 - Instruction memory and data memory are separate
 - May even have different word sizes from each other
 - Instruction memory is read-only
 - Data memory is read/write
 - *Cannot* treat instructions as data.
 - Constants must be explicitly loaded into data memory
- Named after Harvard Mark I computer
 - A relay-based computer with separate instruction and data memories
- Our example processor uses a Harvard architecture

MEMORY ARCHITECTURES (2)

- Von Neumann Architecture:
 - Instructions and data are stored in a single read/write memory
- A big benefit over Harvard architecture:
 - Much easier to load and manage programs on the computer
- Some big problems too:
 - Programs can easily corrupt their own code!
 - (Many exploits take advantage of this characteristic...)
 - CPU only has one bus to read both instructions and data
 - Harvard architecture has two buses; can use them in parallel
- Modern processors often blend these approaches
 - Overall system design uses von Neumann architecture
 - Internally, CPU has separate instruction, data memories
 - Used in very different ways, so separating them allows for much greater hardware optimization

INDIRECT MEMORY ACCESS

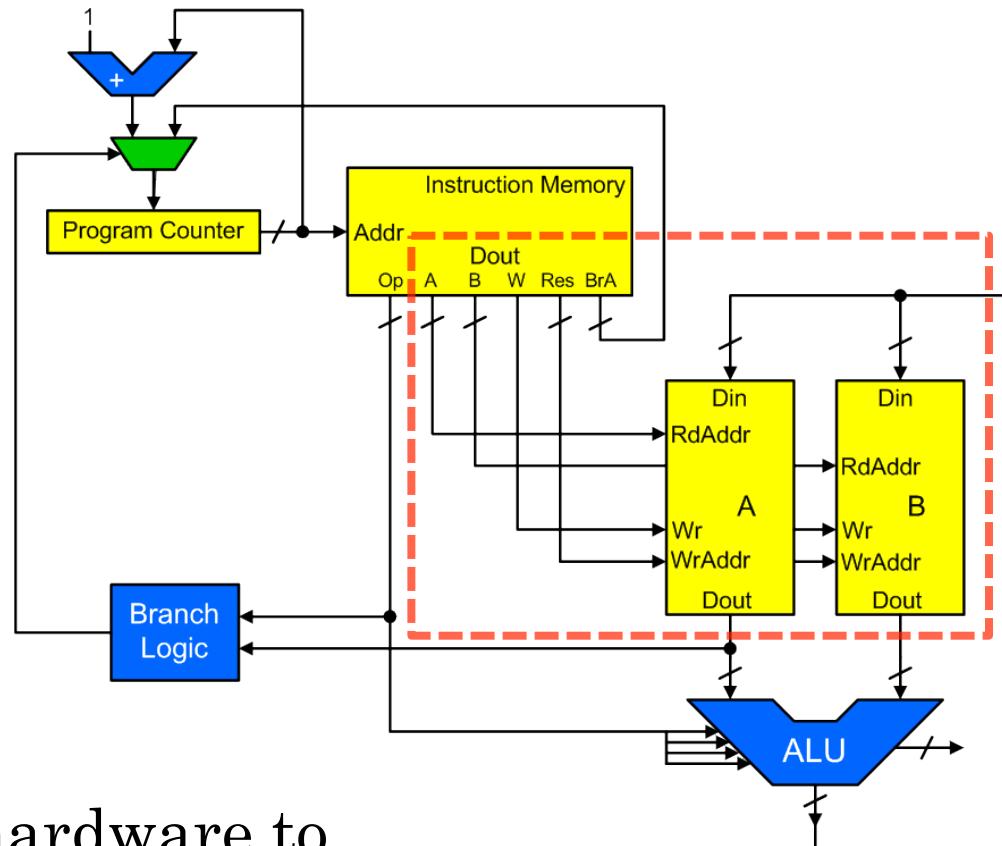
- Still plenty of problems our simple processor design can't handle
- Example function:
 - Add two vectors together, component by component, storing the result into a third vector
 - Vectors to use are arguments to the function
- C code:

```
void vector_add(int *a, int *b, int *r, int length) {  
    int i;  
    for (i = 0; i < length; i++)  
        r[i] = a[i] + b[i];  
}
```

- This code is *independent* of where **a**, **b**, **r** are stored
 - Code can be reused on *any* set of three vectors
 - Addresses of the vectors are specified in the arguments

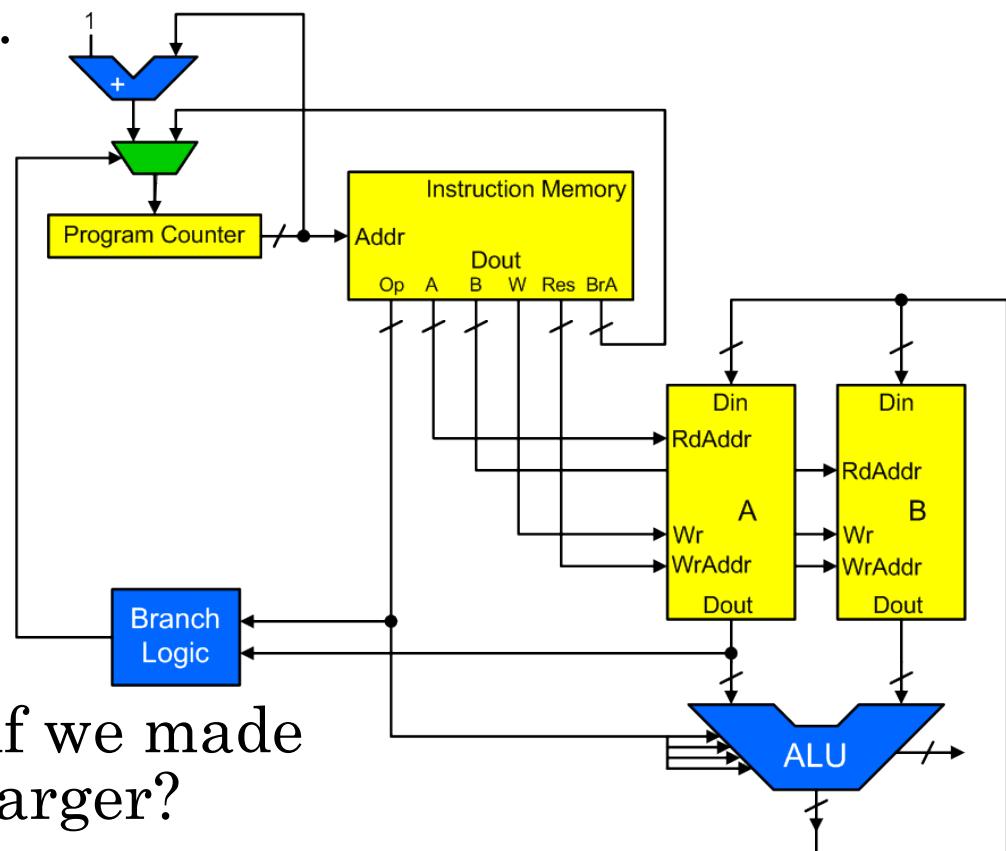
INDIRECT MEMORY ACCESS (2)

- Problem:
 - This processor can only specify literal address-values in the instructions
- Need to introduce a way to get memory addresses from variables instead...
- Need to update the hardware to support *indirect* memory access



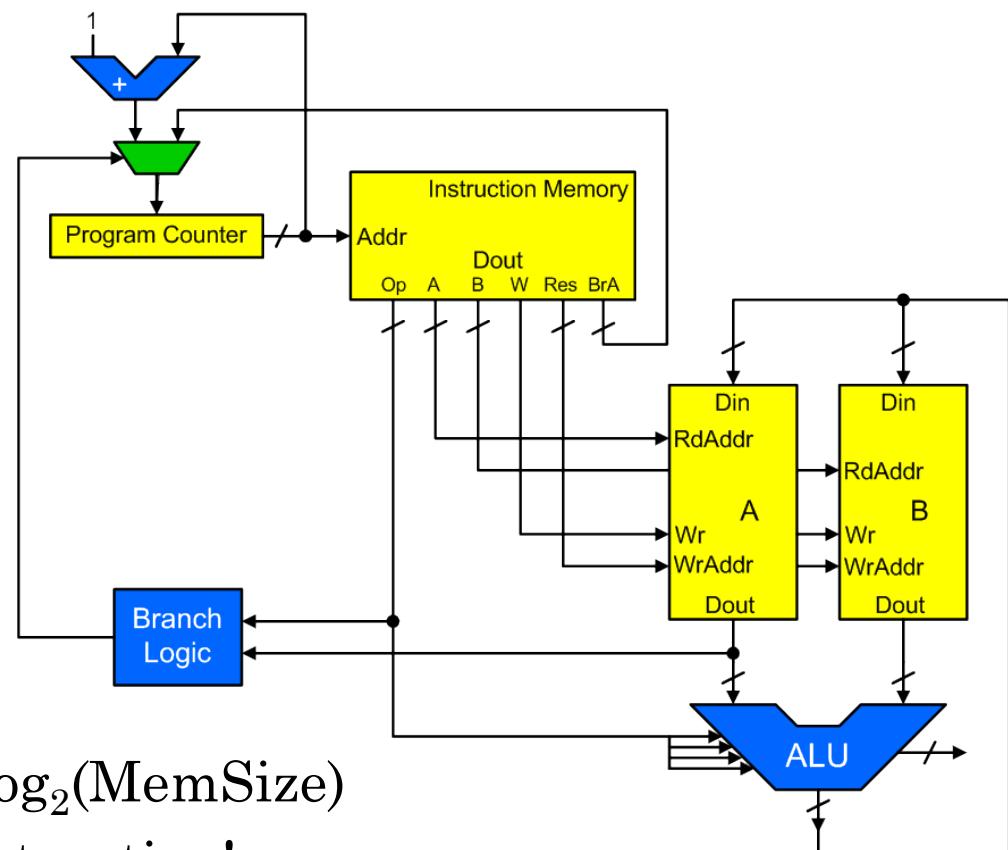
LARGE MEMORIES

- But, before we talk about new functionality, this design has a problem.
- Our current memory only has 8 locations
 - Not very useful...
- Really need *much* larger memories than this!
- What would happen if we made our memories *much* larger?
 - e.g. 4GB, like modern 32-bit architectures



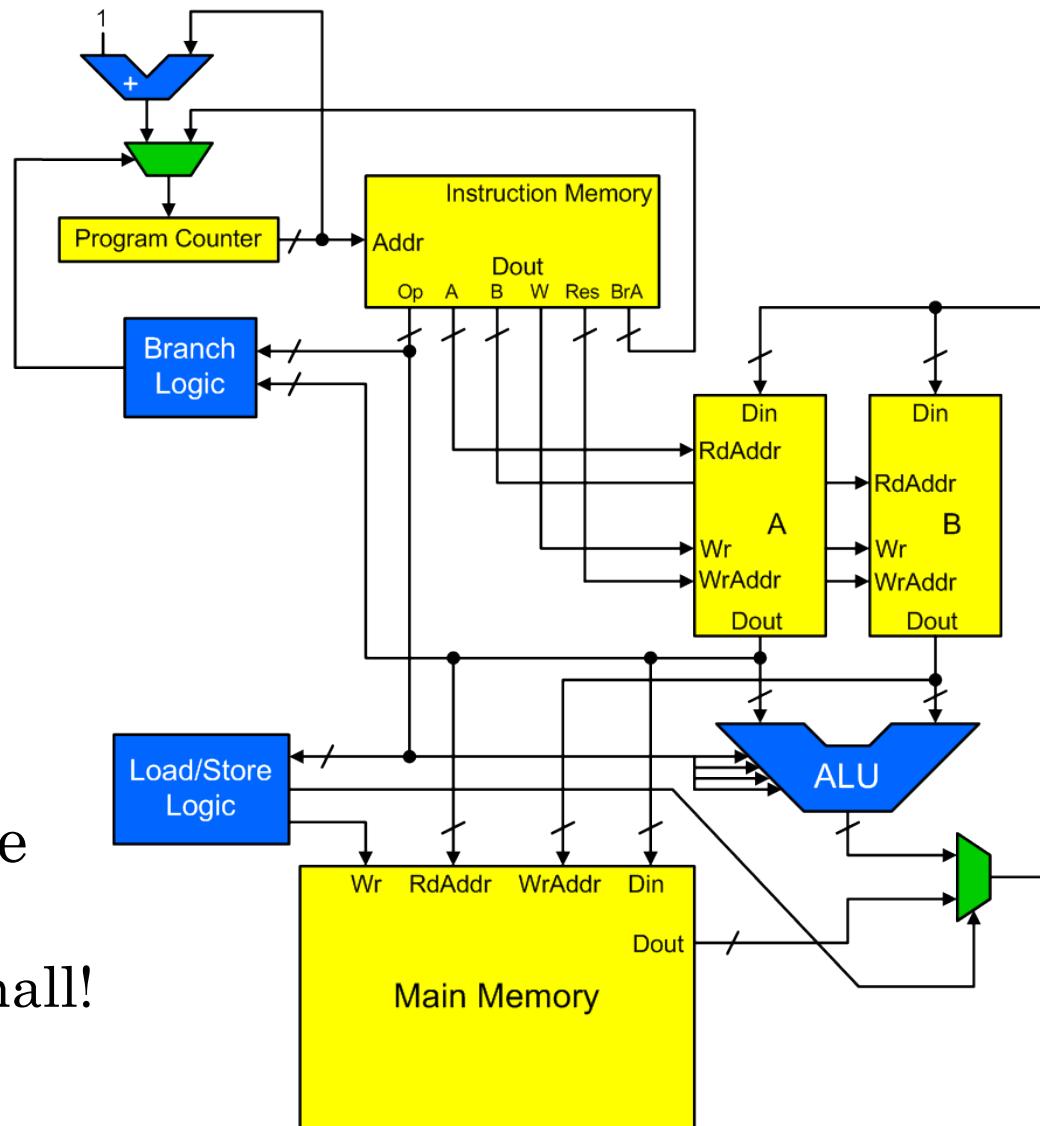
LARGE MEMORIES AND INSTRUCTION SIZE

- We *directly encode* memory addresses into our instructions.
 - Memory has 8 slots
 - $\log_2(8) = 3$ bits for memory addresses
- If memory is 4GB in size, we need 32 bits to specify addresses!
- Instruction size \approx
 - $\log_2(\text{NumOps}) + 3 \times \log_2(\text{MemSize})$
 - ≈ 100 bits for each instruction!



SUPPORTING LARGE MEMORIES

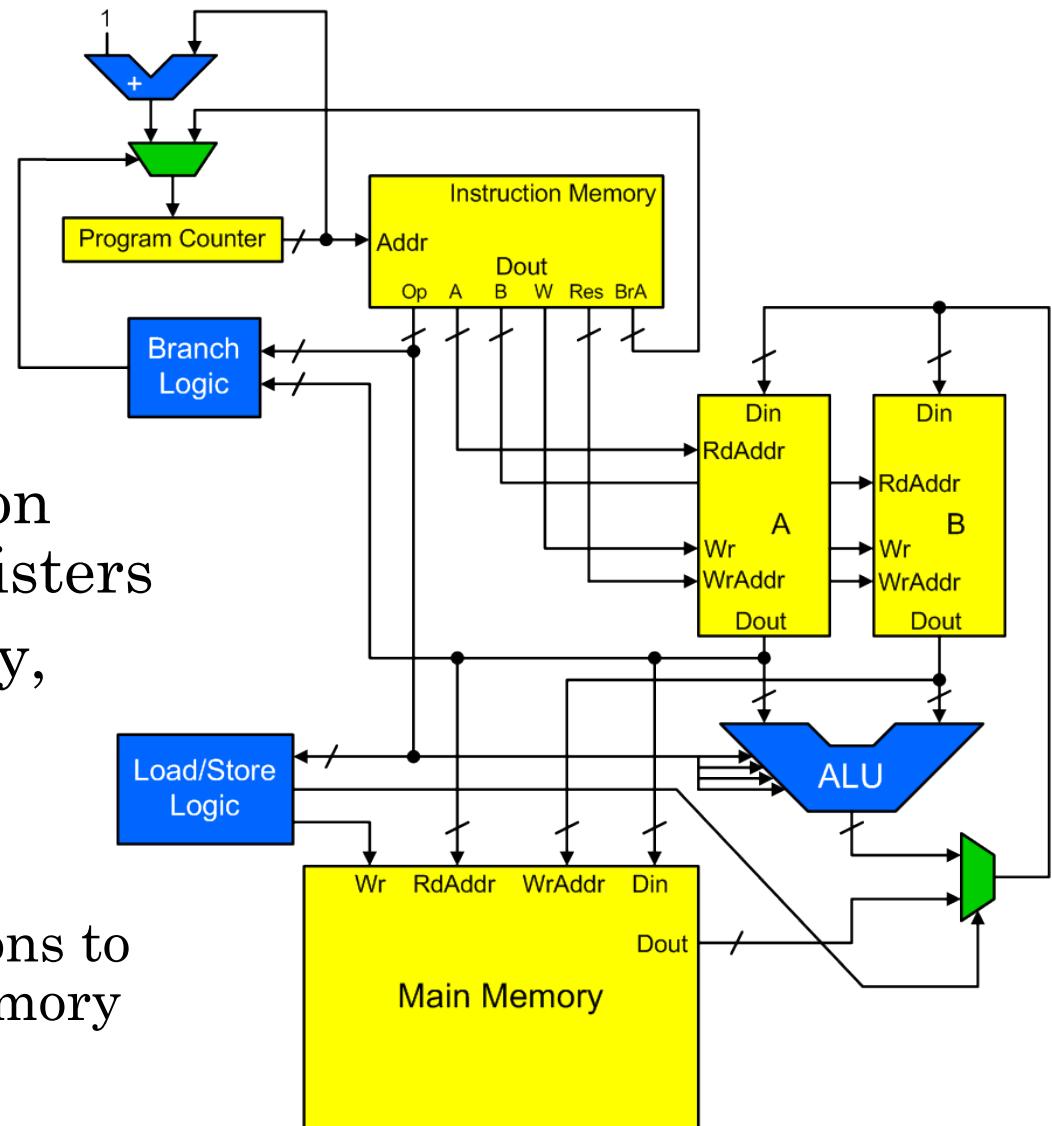
- A better design for large memories:
- Small memory is called a *register file*
 - Individual locations are called *registers*
 - Small number of registers (e.g. 8, 64)
- Instructions only use register addresses
 - Instructions stay small!



SUPPORTING LARGE MEMORIES (2)

- A better design for large memories:

- ALU only operates on values stored in registers
- To use main memory, must explicitly load or store values into registers
 - Need new instructions to work with main memory



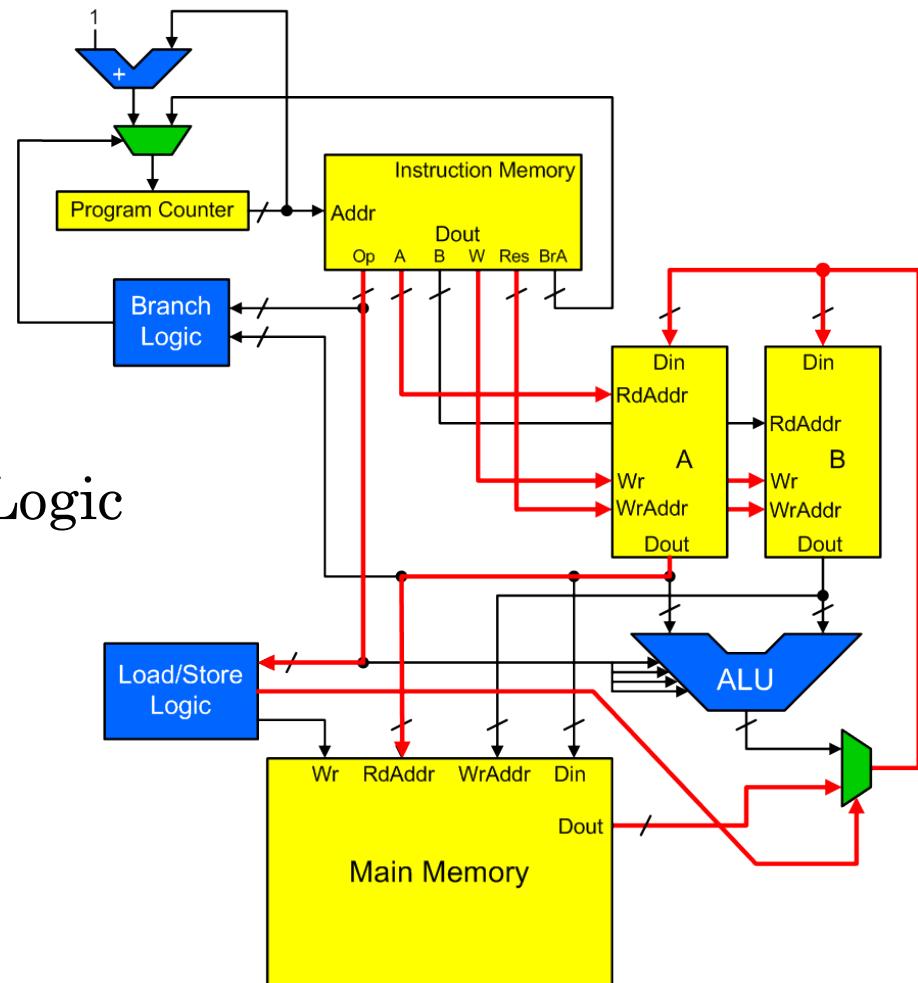
LOADING VALUES FROM MAIN MEMORY

- To load values from main memory, introduce:

- LD SRC, DST
- SRC, DST are registers

- Meaning:
 - DST = Memory[SRC]

- LD activates Load/Store Logic
 - SRC value fed to RdAddr of main memory
 - Output of main memory fed to DST register
 - ALU is not utilized at all



STORING VALUES TO MAIN MEMORY

- To store values to main memory, introduce:

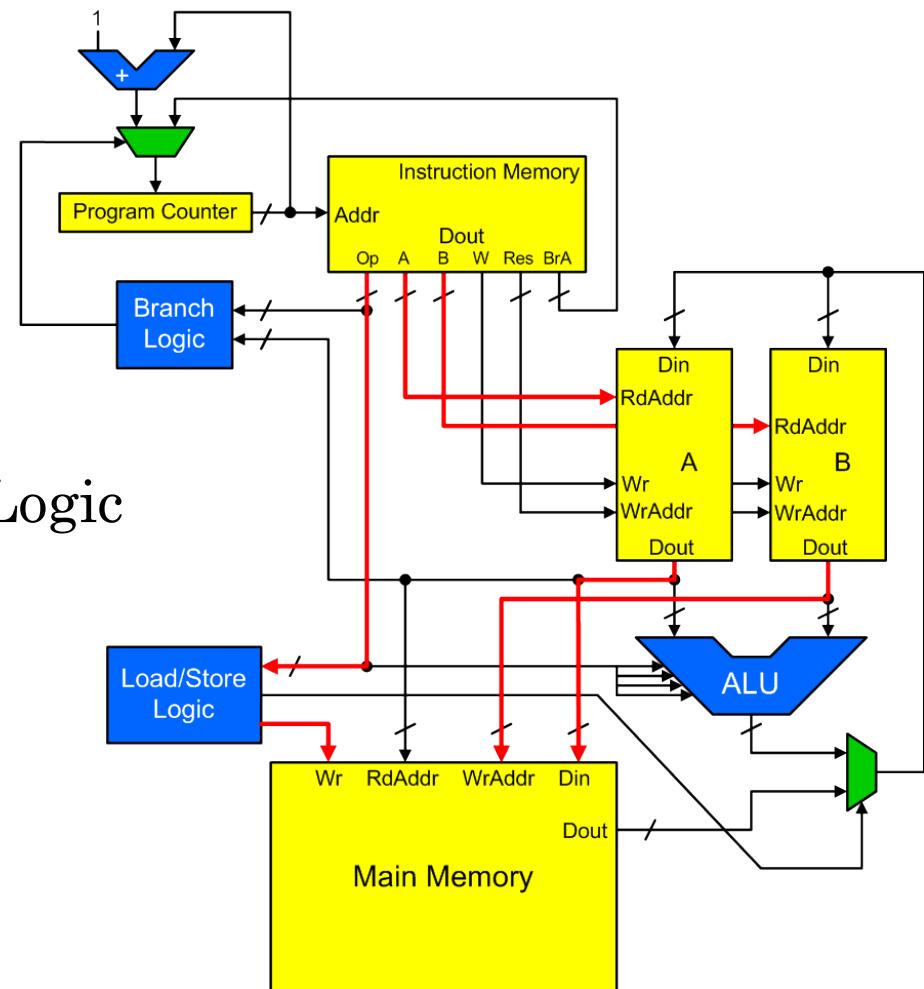
- ST SRC, DST**
- Again, SRC, DST are registers

- Meaning:

- $\text{Memory}[\text{DST}] = \text{SRC}$

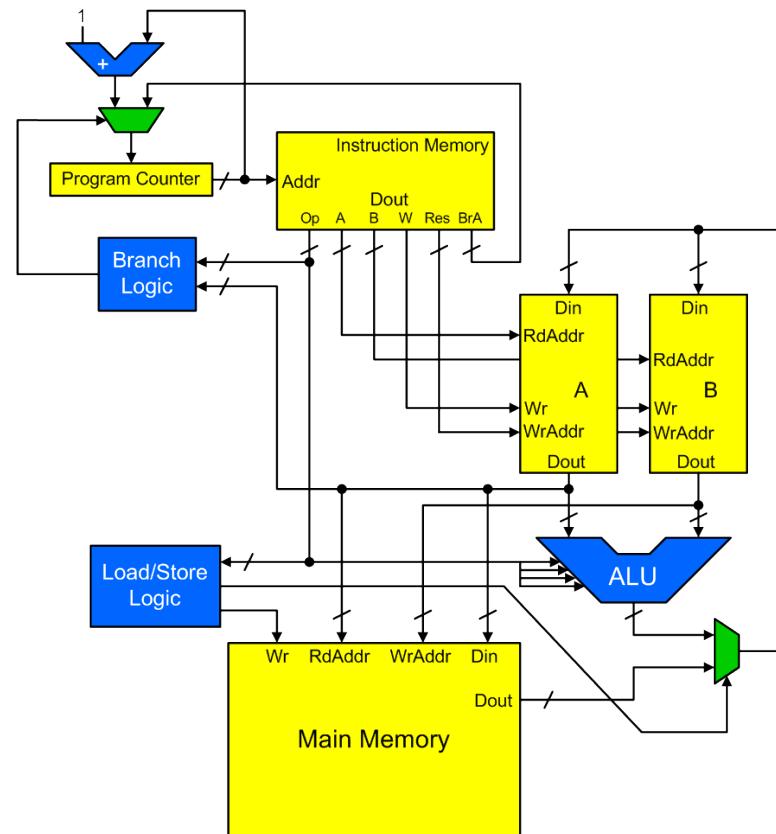
- ST activates Load/Store Logic**

- DST value fed to WrAddr of main memory
- SRC register fed to input of main memory
- Again, ALU not utilized



LOAD/STORE ARCHITECTURE

- This is called a Load/Store Architecture
 - ALU only operates on registers
 - Must explicitly load/store values to main memory
 - (Often seen in RISC ISAs)
- Benefits:
 - Very simple architecture
 - All instructions are same size
 - Instructions execute quickly!
- Drawbacks:
 - Memory-intensive programs require many operations to implement!



EXAMPLE: VECTOR-ADD FUNCTION

- Consider the body of our vector-add function:

```
int i;  
for (i = 0; i < length; i++)  
    r[i] = a[i] + b[i];
```

- Arguments **a**, **b**, **r** are addresses of the *start* of their respective arrays

- One implementation of **r[i] = a[i] + b[i]**:

```
ADD A, I, ADDR      # Compute address of a[i]  
LD  ADDR, AI        #     and retrieve it.  
ADD B, I, ADDR      # Compute address of b[i]  
LD  ADDR, BI        #     and retrieve it.  
ADD AI, BI, RI      # Compute a[i] + b[i].  
ADD R, I, ADDR      # Compute address of r[i]  
ST  RI, ADDR        #     and store sum.
```

- Hope the processor can make this fast, somehow...*

ALTERNATIVE TO LOAD/STORE

- Instead of Load/Store architecture, we can also support a rich set of operand-types
- For example, an operand could be:

- A constant (i.e. an immediate value)
- A register

```
ADD 14, R1, R2          # R2 = 14 + R1
```

- A direct address

```
ADD [150], 14, R7        # R7 = Memory[150] + 14
```

- An indirect address

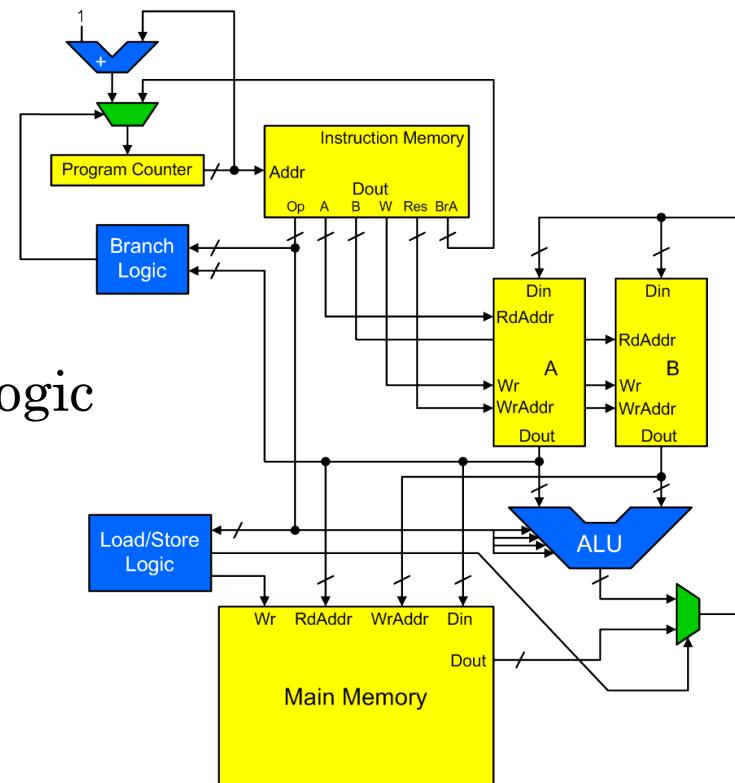
```
ADD [R5 + 150], 14, R6  # R6 = Memory[R5+150] + 14
```

- Instruction encodings may need to include:

- Immediate value, Register index, Address, Address + Register index
- Instruction encodings are variable-length, and must also include operand-mode bits to indicate types of operands

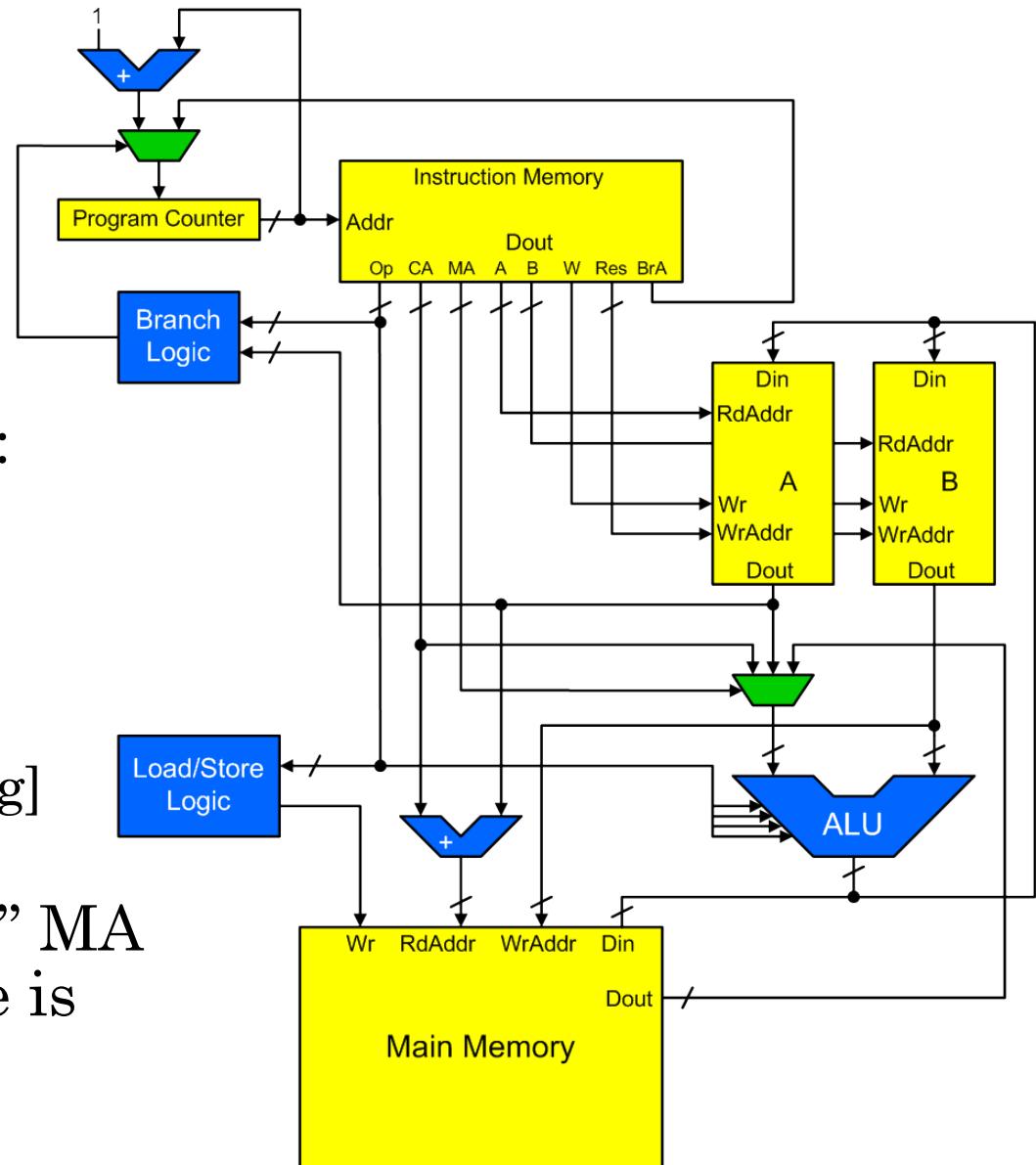
SUPPORTING VARIOUS OPERAND TYPES

- Instruction encodings can include:
 - Immediate value, Register index, Address, Address + Register index
- Clearly, we will need new logic to feed the ALU!
- Also, instructions become much more complicated
 - (CISC processors employ multiple operand types)
 - Requires complex, dedicated instruction-fetch and decode logic
- Finally, may need to do arithmetic on addresses fed to the main memory



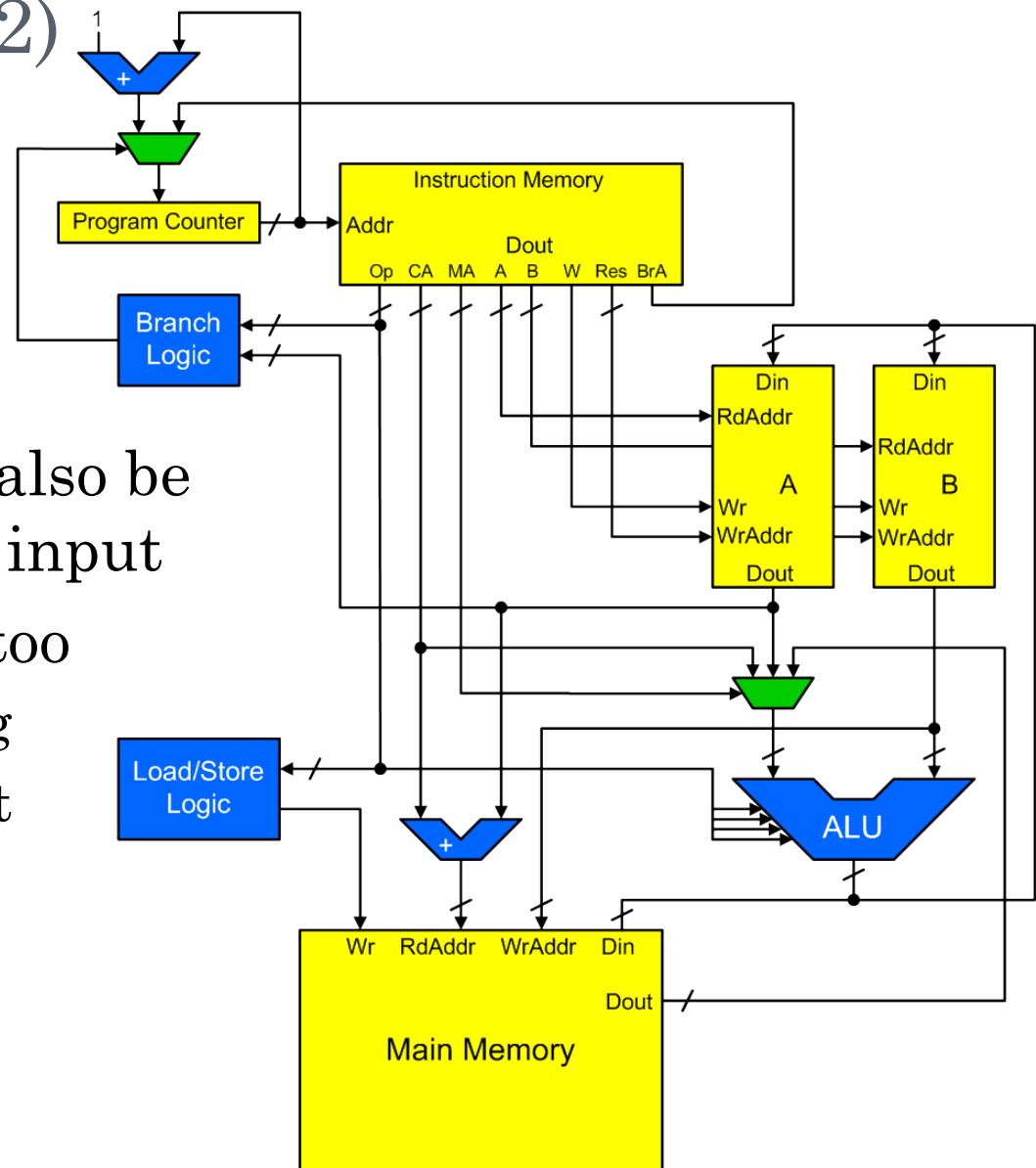
OPERAND TYPES

- Example logic for supporting multiple operand types:
- Operand A supports three operand types:
 - Constant
 - Encoded in CA value
 - Register
 - Encoded in A value
 - Memory[Const + Reg]
 - Uses both A and CA
- New “operand mode” MA controls which value is fed into the ALU



OPERAND TYPES (2)

- Example logic for supporting multiple operand types:
- Similar logic would also be added to other ALU input
- Many other details too
 - Instruction decoding
 - Routing ALU output to registers/memory
 - etc.

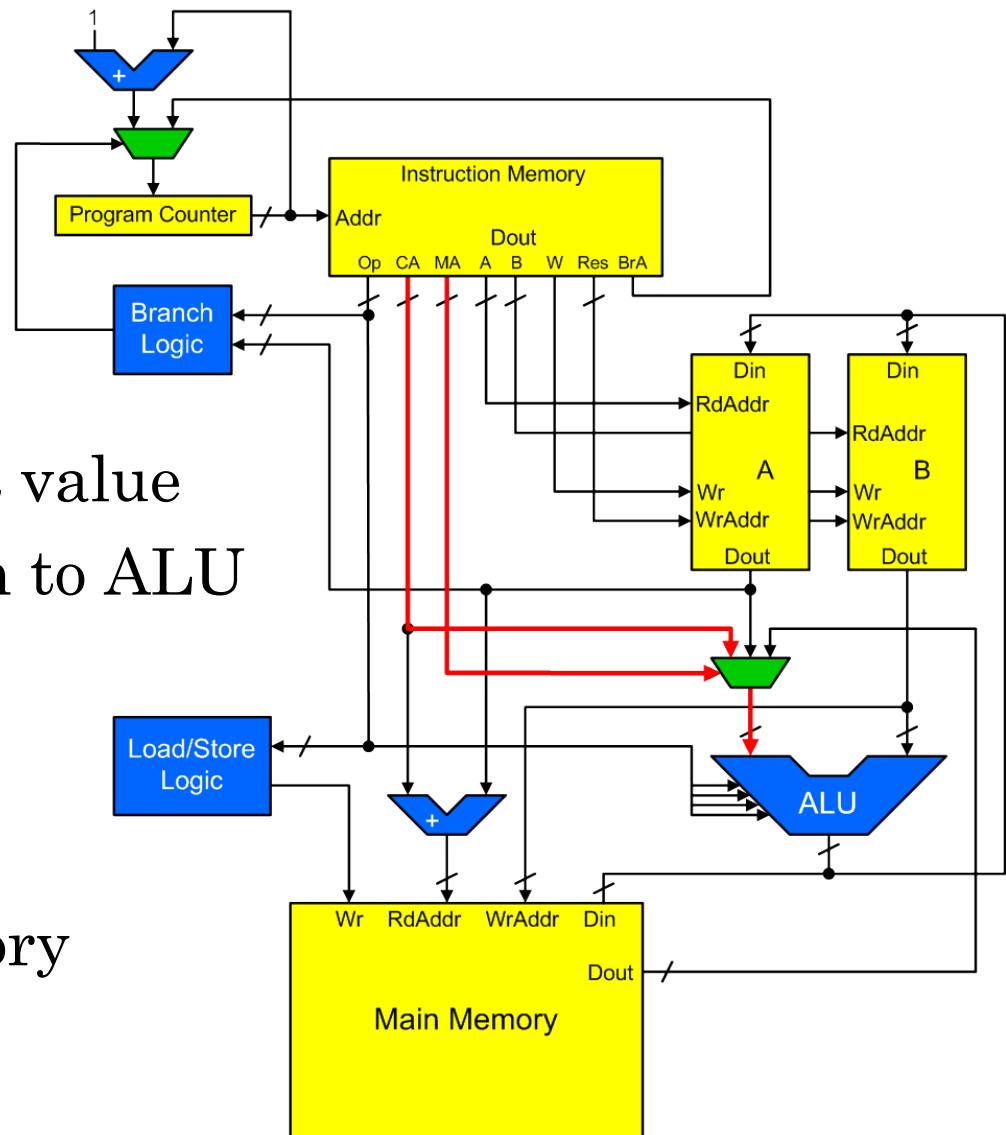


EXAMPLE OPERAND TYPES

- Constant operand:

- CA specifies constant value
- MA feeds CA through to ALU

- Register A and memory are both unused

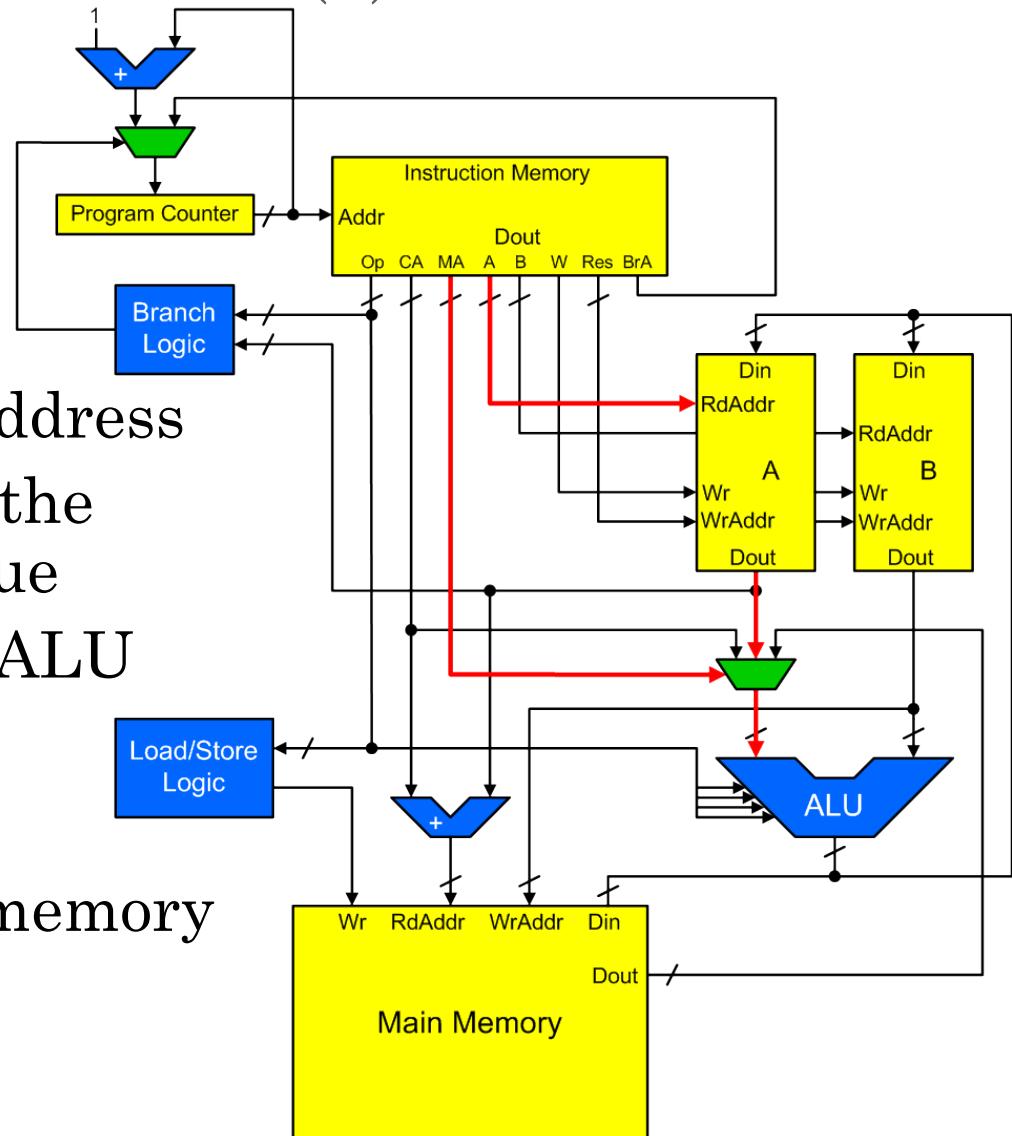


EXAMPLE OPERAND TYPES (2)

- Register operand:

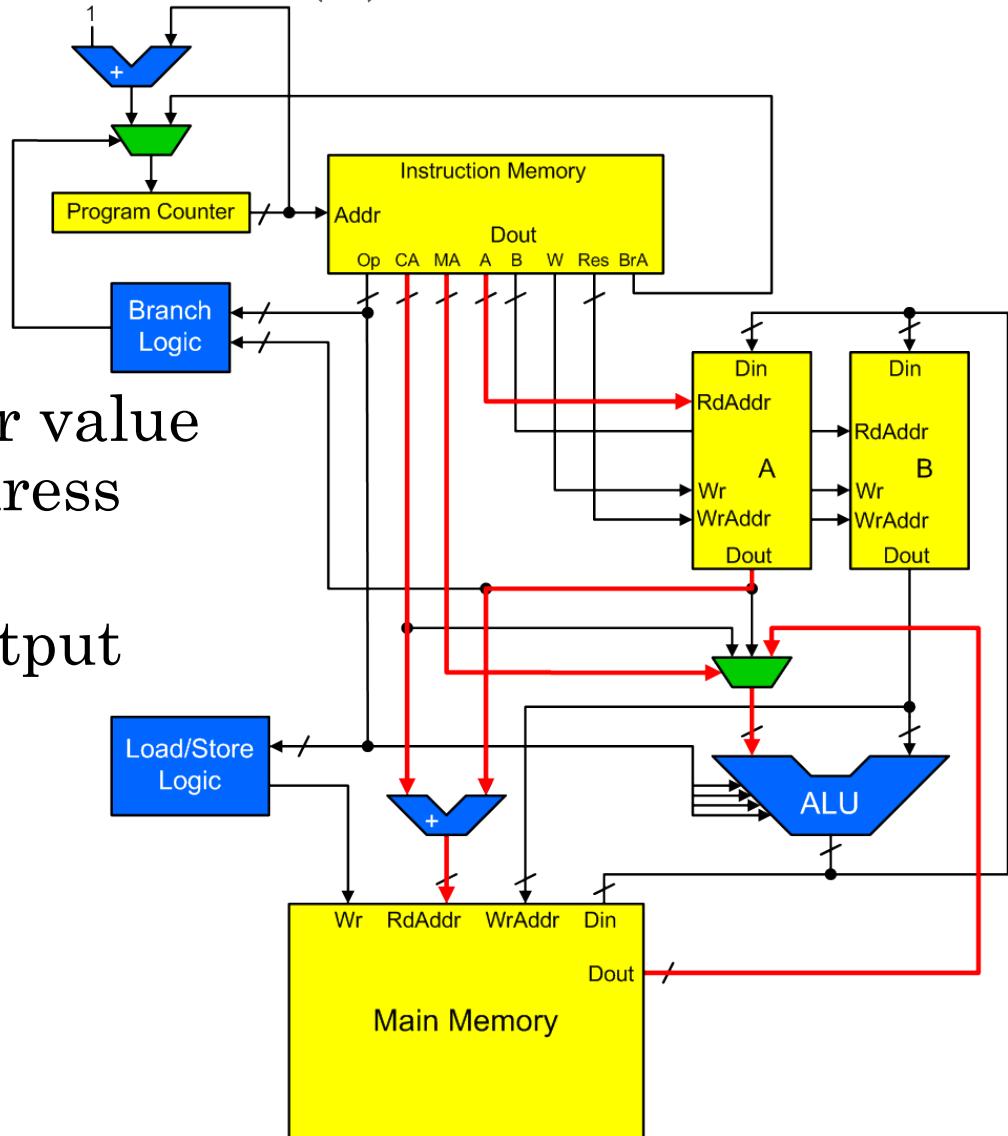
- A specifies register address
- Register file outputs the specified register value
- MA feeds register to ALU

- Constant value and memory are both unused



EXAMPLE OPERAND TYPES (3)

- Indirect memory access operand:



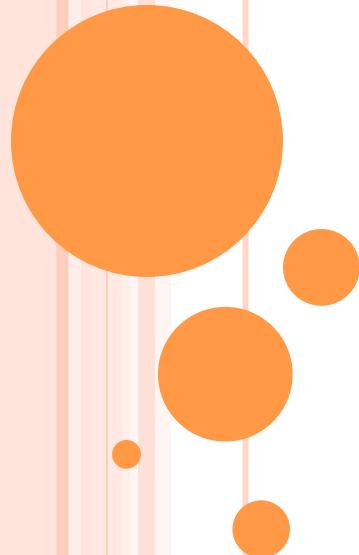
- Constant and register value added to produce address for main memory
- MA feeds memory output to ALU

SUMMARY

- Added branching to our processor
 - Can implement larger computations with fewer instructions
 - Reuse instructions on different data by looping
 - Tailor computations based on the specific data values
 - Allows us to perform computations where the work performed is proportional to the input values
- Explored various memory architecture details
 - Harvard vs. Von Neumann architectures
 - Separating instruction and data processing paths
 - Load/Store architecture vs. multiple operand types
 - Allows us to implement computations that are independent of the specific memory location
 - Allows us to access *much* larger memories, using a relatively small instruction set

NEXT TIME

- Start looking at the Intel IA32 instruction set
- Start writing programs you can run on *real* computers! ☺
- Begin to develop abstractions to facilitate construction of larger programs
 - Subroutines, stacks, recursion

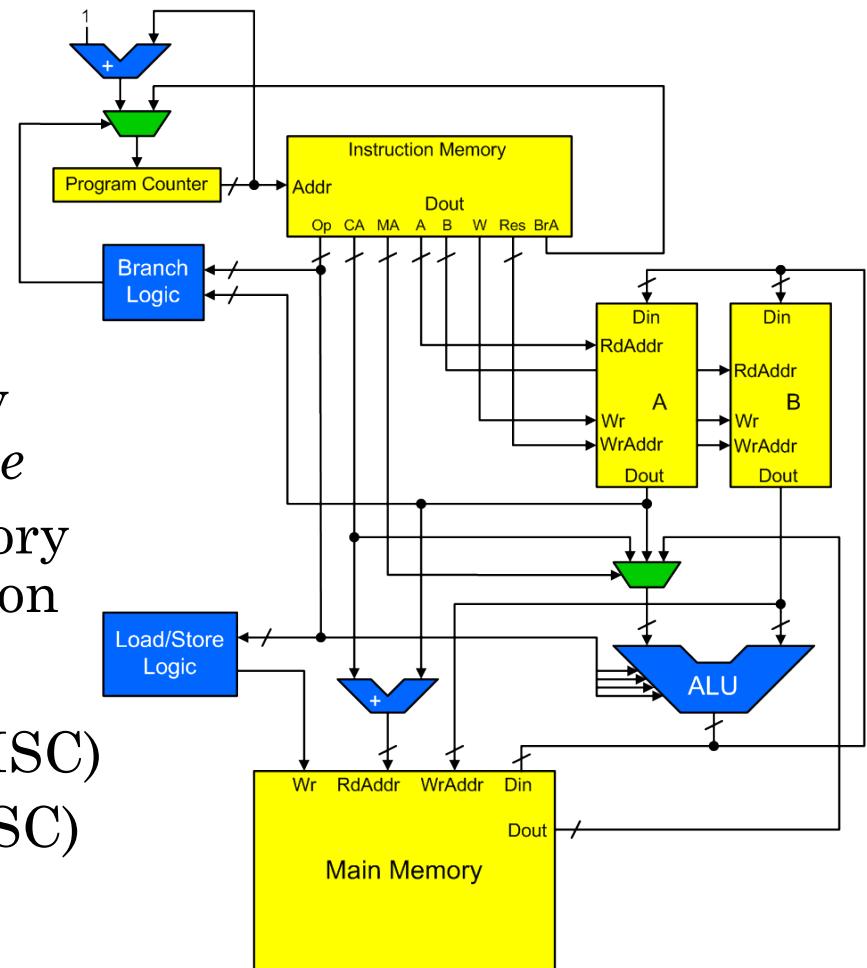


CS24: INTRODUCTION TO COMPUTING SYSTEMS

Spring 2015
Lecture 4

LAST TIME

- Enhanced our processor design in several ways
- Added branching support
 - Allows programs where work is proportional to the input values
- Added a large memory
 - Small, in-processor memory is now called the *register file*
 - Move data from main memory into registers for computation
- Two architectures:
 - Load/Store Architecture (RISC)
 - Multiple operand types (CISC)



PROGRAMS WITH LOOPS

- Can implement more interesting programs now
 - e.g. multiplication, using our processor's simple instructions

```
int mul(int a, int b) {  
    int p = 0;  
    while (a != 0) {  
        if (a & 1 == 1)  
            p = p + b;  
        a = a >> 1;  
        b = b << 1;  
    }  
    return p;  
}
```

XOR P, P, P
WHILE:
BRZ A, DONE
AND A, 1, Tmp
BRZ Tmp, SKIP
ADD P, B, P
SKIP:
SHR A, A
SHL B, B
BRZ 0, WHILE
DONE:

Control	Operation	
0001	ADD	A B
...	...	
0111	BRZ	A Addr
1000	AND	A B
...	...	
1100	SHL	A
1110	SHR	A

Register	Value
0	A
1	B
2	Tmp
3	1
4	0
7	P

BUILDING BLOCKS

- Multiply function is a useful building block for other programs!
- Example: discriminant of quadratic fn. $ax^2 + bx + c$

```
int discriminant(int a, int b, int c) {  
    return b * b - 4 * a * c;  
}
```

- We know we can implement this in our instruction set
 - Would like to reuse our **mul()** function for this
 - Can implement $4 * (...)$ by shifting left by 2 bits
 - Still need two multiplies to implement this function
- ```
int discriminant(int a, int b, int c) {
 return mul(b, b) - mul(a, c) << 2;
}
```
- *How do we do this?*

## BUILDING BLOCKS (2)

- How do we use **mul ()** as a subroutine?
- Need to know how **mul ()** takes its arguments, and returns its result
  - Decided that R0 and R1 were inputs, and R7 is the product
  - Just pass **mul ()** our inputs, then get result out of R7
- Need a way to transfer control to **mul ()**
  - ...then, **mul ()** has to get back to our code somehow
  - *Hmm...*
- Is this the whole picture?
  - No! **mul ()** also uses R2, R3, R4 internally
  - The calling code needs to avoid using these registers

| Register | Value |
|----------|-------|
| 0        | A     |
| 1        | B     |
| 2        | Tmp   |
| 3        | 1     |
| 4        | 0     |
| 7        | P     |

# SUBROUTINES

- Three major problems we need to solve:
  - Need a way to pass arguments and return values between a caller and the subroutine
  - Need a way to transfer control from a caller to the subroutine, then return back to caller
  - Need to isolate subroutine's state from caller's state
- First problem is primarily a design issue
  - Figure out a convention, then stick with it
- The second and third points are the harder ones

# SUBROUTINES AND CALLERS

- Our program:

```
int discriminant(int a, int b, int c) {
 return mul(b, b) - mul(a, c) << 2;
}
```

- Need to invoke our **mul ()** function twice
  - Hard part is not jumping *to* the **mul ()** function...
  - Need to *get back to* wherever we called it from!

- Can we do this with our current processor architecture?

- No! ☹

- Processor only supports constants for branch addresses

| Control | Operation |        |
|---------|-----------|--------|
| 0001    | ADD       | A B    |
| 0011    | SUB       | A B    |
| ...     | ...       |        |
| 0111    | BRZ       | A Addr |
| ...     | ...       |        |
| 1100    | SHL       | A      |
| 1110    | SHR       | A      |

# SUBROUTINE RETURN-ADDRESSES

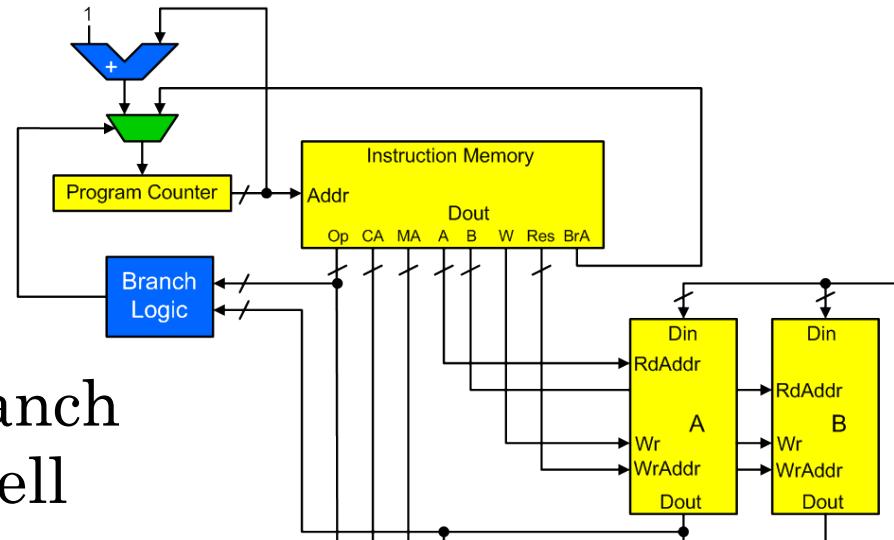
- Problem:

- Can only load constants into the program counter
- $PC = PC + 1$
- $PC = \text{branch\_addr}$

- Need ability to specify branch address in a register as well

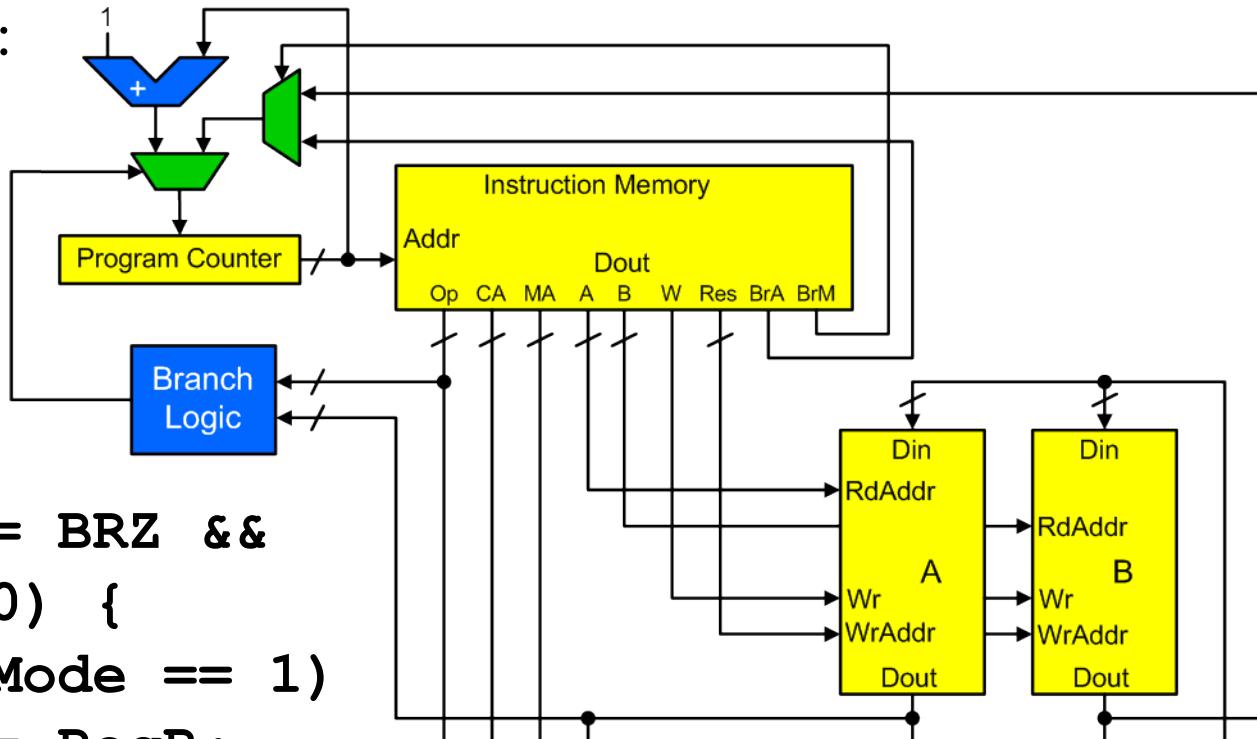
- To call a subroutine:

- Pass the return address in another register
- Subroutine jumps back to return-address at end



# BRANCH-TO-REGISTER LOGIC

- Updated logic:



```
if (Opcode == BRZ &&
 RegA == 0) {
 if (BranchMode == 1)
 ProgCtr = RegB;
 else
 ProgCtr = BranchAddr;
}
```

# USING BRANCH-TO-REGISTER

- Now we can use **mul()** as a subroutine
  - mul()** convention: expect the return-address in R6
- To call our subroutine:
  - Move return-address into R6, then jump to **MUL** address

```
... # Set up other args
MOV RET, R6
BRZ 0, MUL
```

**RET:**

```
... # Result is in R7!
```

- Note: Introduced **MOV** instruction
- (Could write **ADD RET, 0, R6** but that is a bit silly...)

**mul()** parameters:

| Register | Value               |
|----------|---------------------|
| 0        | A                   |
| 1        | B                   |
| 6        | <i>return addr</i>  |
| 7        | P ( <i>output</i> ) |

**MUL:**

```
XOR R7, R7, R7
```

**WHILE:**

```
BRZ R0, DONE
AND R0, 1, R2
BRZ R2, SKIP
ADD R7, R1, R7
```

**SKIP:**

```
SHR R0, R0
SHL R1, R1
BRZ 0, WHILE
```

**DONE:**

```
BRZ 0, R6
```

# COMPUTING THE DISCRIMINANT

- What about our discriminant function?

```
int discriminant(a, b, c) {
 return mul(b, b) - mul(a, c) << 2;
}
```

- Still a huge pain to implement!!

- Only have 8 registers
- **mul()** now uses 7 registers
  - (...if our instructions could encode constants, we would use only 5...)

- Actually, why should callers of **mul()** have to care what registers **mul()** uses internally?!
  - Abstraction: Subroutine's caller shouldn't have to understand subroutine's internals in order to use it

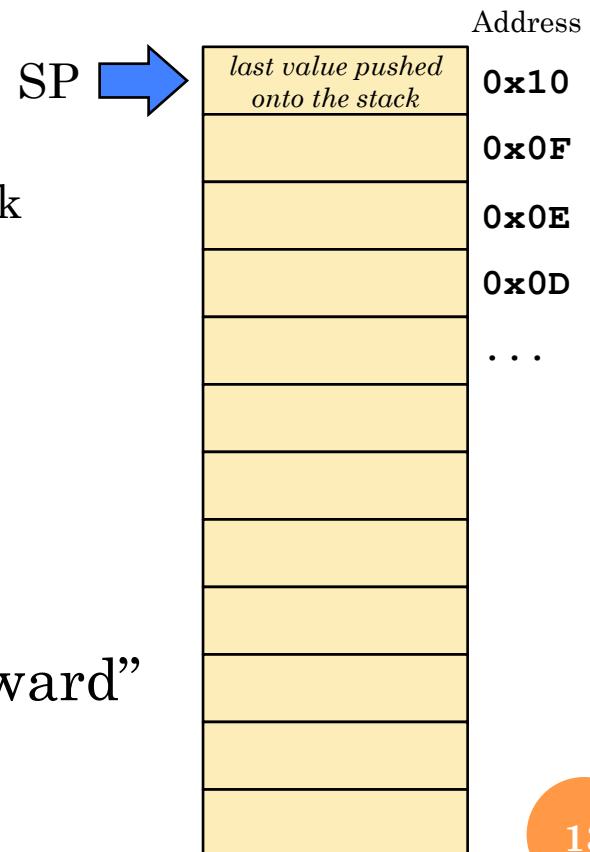
| Register | Value       |
|----------|-------------|
| 0        | A           |
| 1        | B           |
| 2        | Tmp         |
| 3        | 1           |
| 4        | 0           |
| 5        | (free)      |
| 6        | return addr |
| 7        | P           |

# SUBROUTINES AND REGISTERS

- In fact, we would really only like to think about:
  - How to pass arguments to subroutine
  - How to get return-value back from subroutine
- Ideally, would like subroutines to use registers however they want to
  - Somehow, save registers at start of subroutine call
  - Restore registers when subroutine returns to caller
- If a complex subroutine runs out of registers:
  - Save values of some registers, then reuse them
  - When finished, can restore old values of registers
- Can implement these features with a stack

# STACKS

- A Last In, First Out (LIFO) data structure
- Components:
  - A region of memory
  - A *stack pointer* SP
    - Invariant: SP always points to top of stack
- Two operations:
  - **PUSH Reg** – pushes **Reg** onto stack
    - $SP = SP - 1$
    - $Memory[SP] = Reg$
  - **POP Reg** – pops top of stack into **Reg**
    - $Reg = Memory[SP]$
    - $SP = SP + 1$
- IA32 convention: stack grows “downward”
  - Pushing a value decrements SP
  - Popping a value increments SP
  - Will use this convention in our examples



# USING THE STACK

- Can simplify our subroutine implementations

- Pass arguments, return-values via registers
- Subroutines will save and restore other registers they use
- Subroutines must leave stack in the same state they found it

- Example: Updated **mul()**

- R0, R1 are arguments
- R6 is return address
- R7 is result
- Function uses R2, so save it at start, then restore at end

**MUL:**

```
PUSH R2
XOR R7, R7, R7
```

**WHILE:**

```
BRZ R0, DONE
AND R0, 1, R2
BRZ R2, SKIP
ADD R7, R1, R7
```

**SKIP:**

```
SHR R0, R0
SHL R1, R1
BRZ 0, WHILE
```

**DONE:**

```
POP R2
BRZ 0, R6
```

# DISCRIMINANT FUNCTION

- Our discriminant function:

```
int discriminant(int a, int b, int c) {
 return b * b - 4 * a * c;
}
```

- Register usage:

- a = R0
- b = R1
- c = R2
- Result into R7

- Example code for function:

```
DISCR:
 PUSH R0 # Save A
 MOV R1, R0 # R0 = B, R1 = B
 MOV RET1, R6 # Set up for call
 BRZ 0, MUL # mul(B, B)

RET1:
 POP R0 # Restore A
 PUSH R7 # Save B*B
 MOV R2, R1 # R1 = C
 MOV RET2, R6 # Set up for call
 BRZ 0, MUL # mul(A, C)

RET2:
 POP R1 # Restore B*B
 SHL R7, R7 # Multiply A*C by 4
 SHL R7, R7
 SUB R1, R7, R7 # R7 = B*B - 4*A*C
 DONE
```

# DISCRIMINANT FUNCTION (2)

- Significantly easier to implement than before...
- Computed  $b^2$  first
  - Needed to save  $a$  before calling `mul()`
- Saved result of first multiply operation
  - Pushed R7 onto stack
  - Popped into R1
- An example of using stack to save and restore intermediate values

DISCR:

```
 → PUSH R0 # Save A
 MOV R1, R0 # R0 = B, R1 = B
 MOV RET1, R6 # Set up for call
 BRZ 0, MUL # mul(B, B)

RET1:
 → POP R0 # Restore A
 → PUSH R7 # Save B*B
 MOV R2, R1 # R1 = C
 MOV RET2, R6 # Set up for call
 BRZ 0, MUL # mul(A, C)

RET2:
 → POP R1 # Restore B*B
 SHL R7, R7 # Multiply A*C by 4
 SHL R7, R7
 SUB R1, R7, R7 # R7 = B*B - 4*A*C
DONE
```

# ARGUMENTS AND RETURN-ADDRESS

- There's no reason not to pass the arguments and return address on the stack as well!
- Code for calling **mul (b, b)**:

```
MOV R1, R0 # R0 = B, R1 = B
MOV RET1, R6 # Set up for call
BRZ 0, MUL # mul(B, B)
```

**RET1:**

- Instead, introduce two new instructions:
  - **CALL Addr**
    - Pushes PC of *next* instruction onto stack
    - Then sets PC = Addr
  - **RET**
    - Pops top of stack into PC
- No longer need our **RET1**, **RET2**, ... labels, etc.

## ARGUMENTS AND RETURN-ADDRESS (2)

- New strategy for subroutine calls:
  - Caller pushes subroutine arguments onto stack
  - Caller uses **CALL** to invoke subroutine
  - Subroutine uses stack to perform its computations
    - Access arguments, use stack for temporary storage
    - At end, restore stack to original state at time of call
  - Subroutine uses **RET** to return to the caller

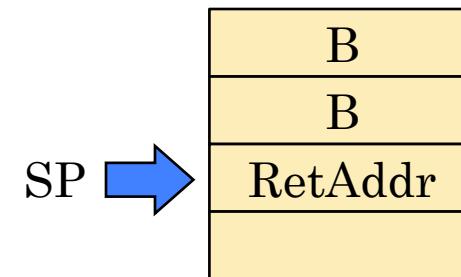
# ACCESSING ARGUMENTS

- How does subroutine access its arguments?
- Our discriminant function:

**DISCR:**

```
PUSH R1 # R7 = mul(B, B)
PUSH R1
CALL MUL

...
```



- For subroutine to access arguments, definitely need indirect memory access support!
- Multiply function arguments:
  - $[SP + 2]$  = first argument
  - $[SP + 1]$  = second argument
  - Remember: our stack grows downward
    - Values pushed earlier are at *higher* addresses

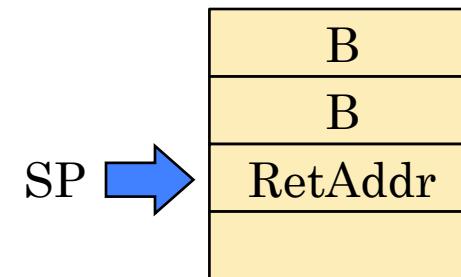
## ACCESSING ARGUMENTS (2)

- How does subroutine access its arguments?
- Our discriminant function:

**DISCR:**

```
PUSH R1 # R7 = mul(B, B)
PUSH R1
CALL MUL

...
```



- Alternative to indirect memory access?

- Subroutine pops off return-address to access args, then later restores stack for return to caller
- “*What could possibly go wrong?*”

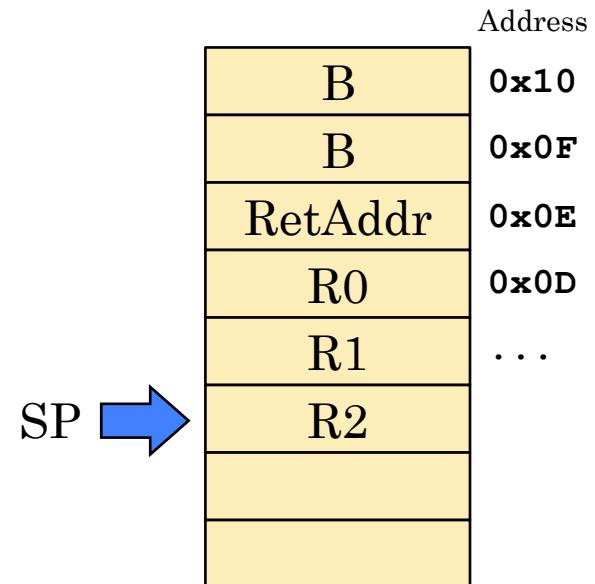
# ACCESSING ARGUMENTS (3)

- mul () routine also modifies certain registers

- e.g. R2 is used to compute (A & 1) temporary value
- R0 and R1 are also modified
- Need to push old values onto stack so we can restore these values later

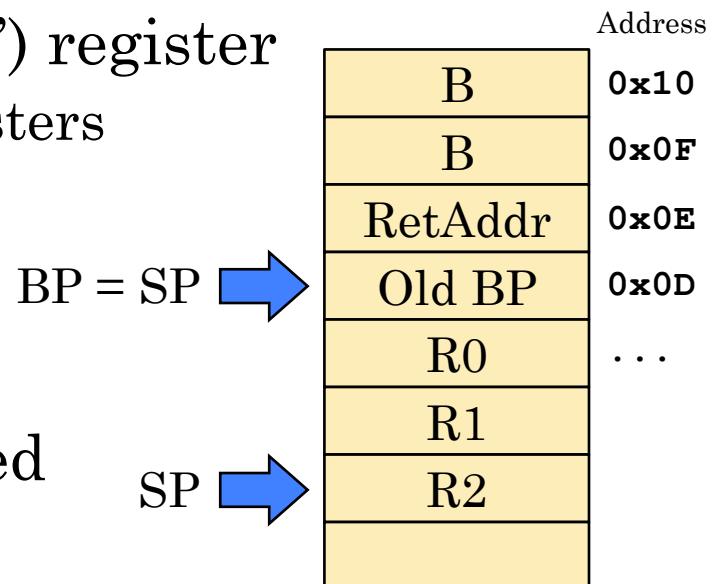
- Problem:

- Makes it *much* harder to reference our function arguments!
  - Now args are at [SP + 5] and [SP + 4]
- If subroutine has to push other values onto stack as it executes, these offsets change again



## ACCESSING ARGUMENTS (4)

- Solution: introduce a reference-point on the stack for accessing arguments
- Example: a BP (“base pointer”) register
  - Set  $BP = SP$ , before saving registers that are locally modified
  - Since we change BP, need to save it first before we store SP into it
- Now arguments can be accessed using BP as a reference-point
  - Argument 1 is at location  $[BP + 3]$
  - Argument 2 is at location  $[BP + 2]$
  - Return address is at location  $[BP + 1]$
  - Locally modified registers stored below BP on stack



# ACCESSING ARGUMENTS (5)

- Our discriminant function:

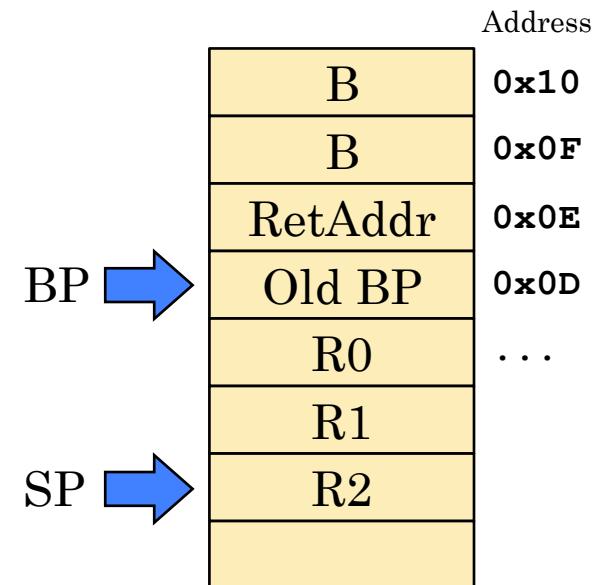
DISCR:

```
PUSH R1 # R7 = mul(B, B)
PUSH R1
CALL MUL
...
```

- mul() routine, updated with new argument-passing mechanism:

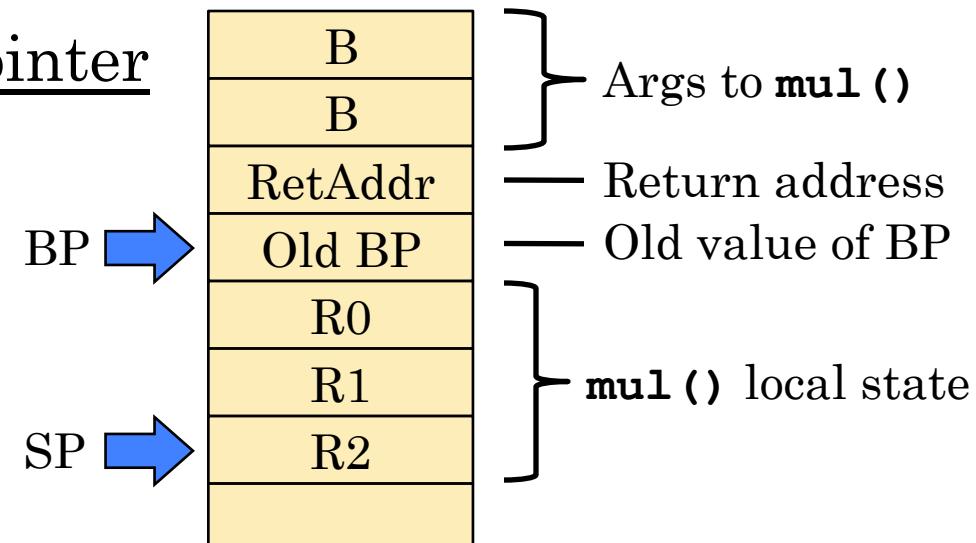
MUL:

```
PUSH BP # Save old BP
MOV SP, BP # Copy SP to BP
PUSH R0 # Save registers
PUSH R1 # that we modify
PUSH R2 # locally.
MOV [BP + 3], R0 # Arg 1
MOV [BP + 2], R1 # Arg 2
...
```



# STACK FRAMES

- A stack frame is the portion of the stack allocated for a specific procedure call
- Includes arguments, return address, and local state used by the subroutine
- BP called the frame pointer
  - Since SP can move, values are accessed via the frame pointer
  - Since number/size of arguments is known, can tell where stack frame starts



- Very common strategy for supporting procedures
  - IA32 has a BP register for storing frame pointer

# TRANSITION TO IA32

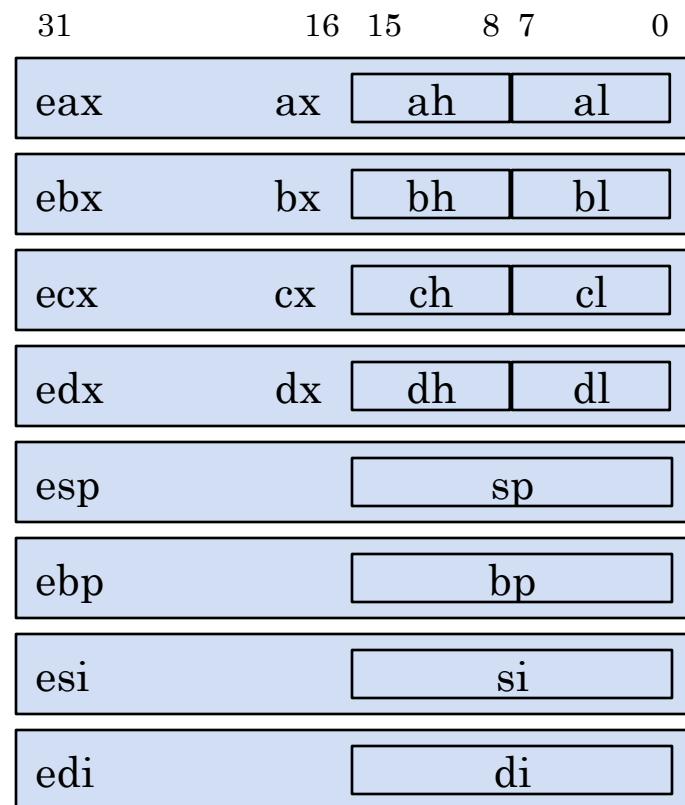
- Our model has gotten quite sophisticated
  - Memory access, including indirect memory access
  - Branching instructions, plus “branch-to-register”
  - Introduce a “stack” abstraction for saving registers, managing procedure arguments, return addresses
  - Introduce “stack frames” and frame pointers for easy access to procedure arguments and local variables
- Time to move to IA32 instruction set architecture
  - Like example, has only 8 general-purpose registers...
  - Provides rich support for all of these abstractions
    - Includes many special-purpose registers devoted to these abstractions
    - A very rich instruction set that makes it easy to use them

# IA32 OVERVIEW

- IA32 is the instruction-set architecture for Intel Pentium-family processors
  - 32 bit and 64 bit processors
- Also known as x86 family
  - First processor was 8086 (released 1978)
  - 16 bit processor; 29K transistors
- Intel continued to develop this series
  - 80186, 80286 – 16 bit; various addressing modes
  - 80386, 80486 – 32 bit; 486 integrated floating-point
  - Pentium series – instruction-set upgrades, optimizations
  - Pentium 4 – first introduction of 64 bit support
    - AMD developed x86-64 extensions first; Intel adopted them
    - P4 also introduced “hyper-threading” architecture: allows interleaved execution of two threads on one processor
  - Core 2 (multicore), Core i7 (multicore + hyper-threading)
- Backward-compatibility preserved throughout series

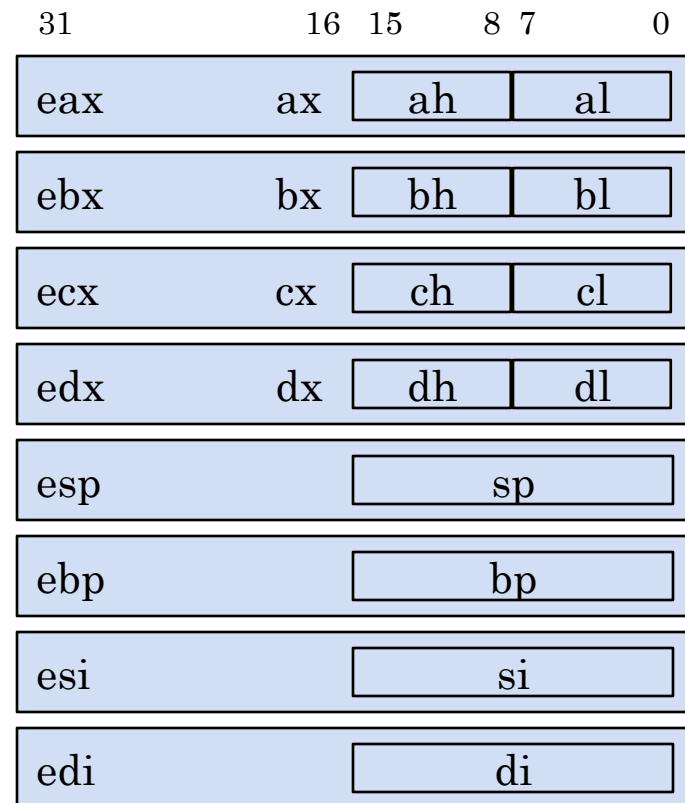
# IA32 REGISTERS

- IA32 has 8 general-purpose registers, and a wide variety of specialized registers
- eax, ebx, ecx, edx
  - General 32-bit registers for computations
- esp = stack pointer
  - Used with PUSH, POP, CALL, RET, etc.
- ebp = base pointer
  - For stack frame pointer
- esi, edi
  - Used for string load, move, store operations



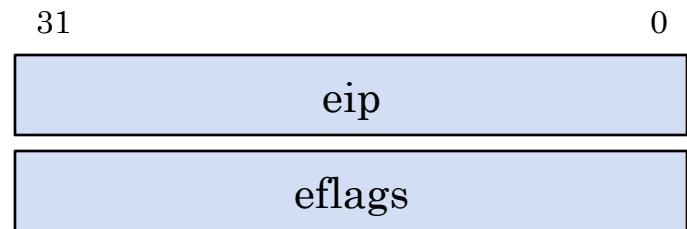
# IA32 REGISTERS (2)

- Most operations support args of varying widths
  - eax is 32 bits
  - ax is low 16 bits of eax
  - ah, al are high/low 8 bits of ax
- Code written for 8086 thru 286 only accesses ax, ah, al
  - Still available if necessary, but not used very often
- Also 64-bit registers:
  - rax, rbx, rcx, rdx
  - rsp, rbp, rsi, rdi



# IA32 REGISTERS (3)

- Two other important registers:
- eip = instruction pointer
  - 32-bits
  - Cannot access this register directly!
  - Modify eip using branching instructions
  - rip = 64-bit version
- eflags = flags register
  - Many different flags
    - e.g. carry flag, zero flag, sign flag, overflow flag, direction flag
  - Cannot access/manipulate directly
  - Many operations for loading/saving/manipulating the flags register



# IA32 REGISTERS (4)

- Many other interesting registers
  - See IA32 manuals if you are curious!
  - Won't use these registers for assignments this term
- Registers for segmented memory models
  - cs, ds, es, fs, gs, ss – all 16 bit
- Registers for floating-point arithmetic
  - 32-bit, 64-bit, 80-bit floating point values
- Registers for SIMD and MMX instructions
  - Single Instruction, Multiple Data – instructions for processing vectors of data very rapidly
  - MMX – more SIMD instructions for hardware media processing acceleration

# IA32 AND WORD SIZE

- Word size in computing systems *usually* refers to unit of data processor is designed to work with
  - Can vary widely depending on system/application
- For IA32, word size defined to be 2 bytes (16 bits)
  - Original word size of 8086/8088
  - Even on 32/64-bit processors, word size is still 2 bytes
- Doubleword (dword) = 4 bytes (32 bits)
  - C **int** and **long int** data types are usually doublewords for IA32, **gcc**
- Quadword (qword) = 8 bytes (64 bits)
  - C **long long int** is a quadword for IA32, **gcc**

# IA32 WORDS AND BYTE-ORDERING

- For multibyte values, order of bytes in value becomes important
- Example: store value 0x12345678 in memory
  - Big endian: most significant byte at lowest address

| Address | 0x100 | 0x101 | 0x102 | 0x103 |
|---------|-------|-------|-------|-------|
| Value   | 0x12  | 0x34  | 0x56  | 0x78  |

- Little endian: least significant byte at lowest address

| Address | 0x100 | 0x101 | 0x102 | 0x103 |
|---------|-------|-------|-------|-------|
| Value   | 0x78  | 0x56  | 0x34  | 0x12  |

- IA32 uses little-endian byte ordering
  - Can make it confusing to look at memory dumps ☹
  - Address of multibyte value is address of lowest byte

# WORDS AND POINTERS

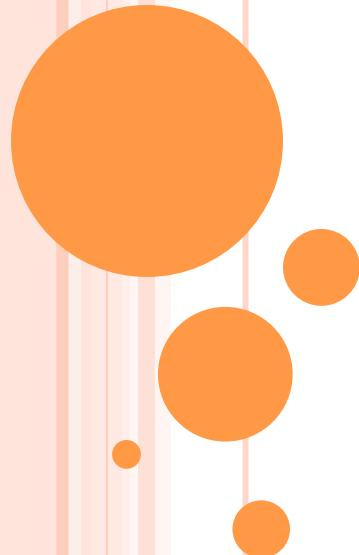
- Word size has an important impact on system!
  - Directly affects how much memory the system can access
- For IA32 (and x86-64):
  - 32-bit processors can access up to  $2^{32}$  bytes (4 GB)
  - 64-bit processors can access *much* more memory
    - Currently can address up to  $2^{48}$  bytes (256 TB)
  - Other hardware may impose greater restrictions
    - e.g. motherboard supports 64-bit processor, only 16GB RAM
- Pointer:
  - The address or location of a value in main memory
  - Pointers also have a type, which specifies number of bytes that the value occupies (among other things)

# IA32 INSTRUCTIONS

- Instructions follow this pattern:
  - *opcode      operand, operand, ...*
- Examples:
  - **add %ax, \$5**
  - **mov %ecx, %edx**
  - **push %ebp**
- **Important note!**
  - Above assembly-code syntax is called AT&T syntax
  - GNU assembler uses this syntax by default
  - Intel IA32 manuals, other assemblers use Intel syntax
- Some big differences between the two formats!
  - **mov %ecx, %edx # AT&T:** Copies ecx to edx
  - **mov edx, ecx # Intel:** Copies ecx to edx

# IA32 INSTRUCTIONS (2)

- Some general categories of instructions:
  - Data movement instructions
  - Arithmetic and logical instructions
  - Flow-control instructions
  - (many others too, e.g. floating point, SIMD, etc.)
- Next time:
  - Dive into the details of IA32 instruction set
  - Examine how to integrate C and IA32 programs

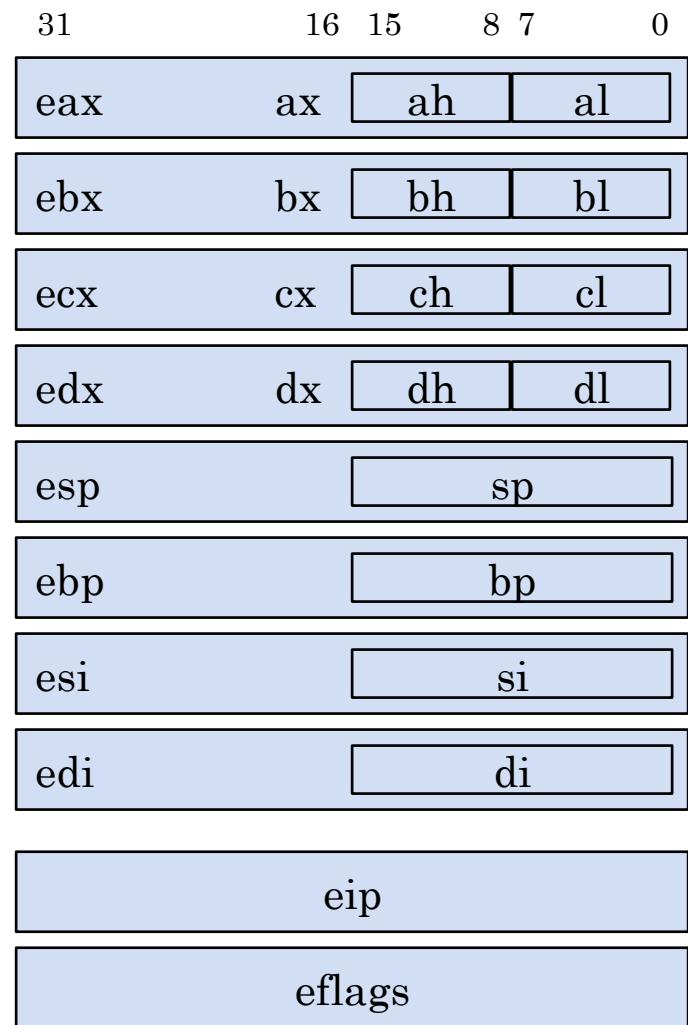


# CS24: INTRODUCTION TO COMPUTING SYSTEMS

Spring 2015  
Lecture 5

# LAST TIME

- Began our tour of the IA32 instruction set architecture
- IA32 provides 8 general-purpose registers
  - eax, ebx, ecx, edx are used for general operations
  - esp is the stack pointer, ebp is the frame pointer (a.k.a. “base pointer”)
  - esi, edi used for string operations
- Two additional registers:
  - eip is the instruction pointer
  - eflags contains status flags



# IA32 INSTRUCTIONS

- Instructions follow this pattern:
  - *opcode      operand, operand, ...*
- Examples:
  - **add \$5, %ax**
  - **mov %ecx, %edx**
  - **push %ebp**
- **Important note!**
  - Above assembly-code syntax is called AT&T syntax
  - GNU assembler uses this syntax by default
  - Intel IA32 manuals, other assemblers use Intel syntax
- Some big differences between the two formats!
  - **mov %ecx, %edx # AT&T:** Copies ecx to edx
  - **mov edx, ecx # Intel:** Copies ecx to edx

# IA32 INSTRUCTIONS (2)

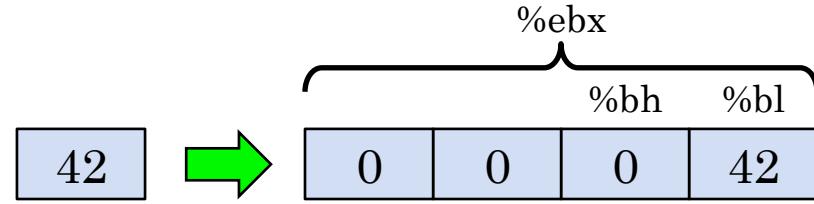
- Some general categories of instructions:
  - Data movement instructions
  - Arithmetic and logical instructions
  - Flow-control instructions
  - (many others too, e.g. floating point, SIMD, etc.)
- Data movement:
  - **mov** Move data value from source to destination
  - **movs** Move value with sign-extension
  - **movz** Move value with zero-extension
  - **push** Push value onto the stack
  - **pop** Pop value off of the stack

# IA32 DATA MOVEMENT INSTRUCTIONS

- Data movement instructions can specify a suffix to indicate size of operand(s)
  - b = byte, w = word, l = doubleword, q = quadword
- Some instructions work with one data size:
  - **movl %ecx, %edx**
    - Moves doubleword (4 byte) register **ecx** into **edx**
  - **pushb %al**
    - Pushes register **al** (1 byte) onto stack
- Move with sign/zero extension takes two sizes:
  - **movsbl %al, %edx**
    - Moves byte **al** into doubleword (4 bytes) register **edx**, extending sign-bit of value into remaining bytes
  - **movzwq %cx, %rax**
    - Moves word (2 bytes) **cx** into quadword (8 bytes) register **rax**, zeroing out higher-order bytes in destination

# IA32 OPERAND TYPES

- Many different operand types and combinations supported by IA32 instruction set
- Immediate values – numeric constants:
  - Must specify \$ prefix to use a numeric constant
  - \$5, \$-37, \$0xF005B411
- Registers:
  - Specify % prefix on register name
  - %ebp, %eax, %rcx
- Example:
  - **movl \$42, %ebx**
    - Moves the value  $42_{10}$  into ebx register



# IA32 MEMORY-REFERENCE OPERANDS

- IA32 has very rich support for memory references
  - Denote memory access as M[Address]
- Absolute memory access
  - Immediate value with no \$ prefix
  - **movl 0xE700, %edx**
    - Retrieves memory value M[0xE700] into **edx**
- Indirect memory access
  - Register name, enclosed with parens: **(Reg)**
  - **movw %cx, (%ebx)**
    - Stores word (2 bytes) in **%cx** into memory location M[%**ebx**]
- Base + displacement memory access
  - **Imm(Reg)** accesses M[Imm + Reg]
  - **movl -8(%ebp), %eax**
    - Retrieves dword M[-8 + %**ebp**] into **%eax**
    - (Presumably, %**ebp** – 8 > 0)

# IA32 MEMORY-REFERENCE OPERANDS (2)

- Indexed memory access

- **(RegB, RegI)** accesses  $M[RegB + RegI]$ 
  - RegB is the base (i.e. starting) address of a memory array
  - RegI is an index into the memory array
- **Imm(RegB, RegI)** accesses  $M[Imm + RegB + RegI]$
- Assumes that array elements are bytes

- Examples:

- **%eax = 150, %ebx = 400**
- **movl (%eax, %ebx), %edx**
  - Retrieves value at  $M[150 + 400] = M[550]$  into **edx**
- **movl %ecx, -200(%ebx, %eax)**
  - Stores **ecx** into location  $M[-200 + 400 + 150] = M[350]$

# IA32 MEMORY-REFERENCE OPERANDS (3)

- Scaled indexed memory access
  - With scale factor  $s = 1, 2, 4, 8$ :
    - $(, \text{Reg}, s)$   $M[\text{Reg} \times s]$
    - $\text{Imm}(, \text{Reg}, s)$   $M[\text{Imm} + \text{Reg} \times s]$
    - $(\text{RegB}, \text{RegI}, s)$   $M[\text{RegB} + \text{RegI} \times s]$
    - $\text{Imm}(\text{RegB}, \text{RegI}, s)$   $M[\text{Imm} + \text{RegB} + \text{RegI} \times s]$
    - For arrays with elements that are 1/2/4/8 bytes
- Examples:
  - $\%eax = 150, \%ebx = 400$
  - **movl (, %eax, 4), %edx**
    - Retrieves value at  $M[150 \times 4] = M[600]$  into **edx**
  - **movl %ecx, 350(%ebx, %eax, 2)**
    - Stores **ecx** into  $M[350 + 400 + 150 \times 2] = M[1050]$

# IA32 MEMORY-REFERENCE SUMMARY

- Summary chart, from IA32 manual:

| Base       | + | Index               | × | Scale | + | Displacement |
|------------|---|---------------------|---|-------|---|--------------|
| <b>eax</b> |   | <b>eax</b>          |   |       |   |              |
| <b>ebx</b> |   | <b>ebx</b>          |   |       |   |              |
| <b>ecx</b> |   | <b>ecx</b>          |   | 1     |   | None         |
| <b>edx</b> |   | <b>edx</b>          |   | 2     |   | 8-bit        |
| <b>esp</b> | + | ( <i>not esp!</i> ) | × | 4     | + | 16-bit       |
| <b>ebp</b> |   | <b>ebp</b>          |   | 8     |   | 32-bit       |
| <b>esi</b> |   | <b>esi</b>          |   |       |   |              |
| <b>edi</b> |   | <b>edi</b>          |   |       |   |              |

- Base, Index, Displacement are all optional
- Scale is only allowed when Index is specified
- Note that **esp** can only be used as a base value, but never as an index value

# IA32 OPERAND COMBINATIONS

- Important constraints on combinations of operand types
- Source argument can be:
  - Immediate, Register, Memory (direct or indirect)
- Destination argument can be:
  - Register, Memory (direct or indirect)
- Both arguments cannot be memory references
  - To move data from one memory location to another, must move Mem1 → Register, then Register → Mem2
- These constraints apply to data movement instructions, and most other instructions too

# IA32 ARITHMETIC/LOGICAL OPERATIONS

- Unary arithmetic/logical operations:
  - **inc Dst**  $Dst = Dst + 1$
  - **dec Dst**  $Dst = Dst - 1$
  - **neg Dst**  $Dst = -Dst$
  - **not Dst**  $Dst = \sim Dst$
- Binary arithmetic/logical operations:
  - **add Src, Dst**  $Dst = Dst + Src$
  - **sub Src, Dst**  $Dst = Dst - Src$
  - **xor Src, Dst**  $Dst = Dst \wedge Src$
  - **or Src, Dst**  $Dst = Dst \vee Src$
  - **and Src, Dst**  $Dst = Dst \wedge Src$
- Specify byte-width of operands via suffixes, as usual
  - **decb %cl**
    - Decrements the 1-byte value in **cl** register
  - **addl 4(%ebp), %eax**
    - Adds M[4 + **ebp**] to contents of **eax**

# IA32 SHIFT OPERATIONS

- Shift operations:
  - **shl k, Dst**  $Dst = Dst \ll k$
  - **shr k, Dst**  $Dst = Dst \gg k$  (logical)
  - **sal k, Dst**  $Dst = Dst \ll k$
  - **sar k, Dst**  $Dst = Dst \gg k$  (arithmetic)
  - **shl, sal** are identical
  - **k** is a constant, or **%cl** register
  - Can only shift values by up to 32 bits
    - ...even when Dst is a 64-bit register!
- Also rotate operations
  - See docs for **rol**, **ror**, **rcl**, **rcr**
  - Similar form, constraints as shift operations

# IA32 MULTIPLY, DIVIDE OPERATIONS

- Multiplication and division are more challenging
  - $32\text{-bit value} \times 32\text{-bit value} = 64\text{-bit value}$
  - $64\text{-bit value} \div 32\text{-bit value} = 32\text{-bit quotient},$   
 $32\text{-bit remainder}$
- Two-argument multiplication:
  - **imul Src, Dst**
  - Use width modifier, as usual: **imull (%ebx), %ecx**
  - For Src, Dst of bit-width  $w$ , produces result also of width  $w$
  - $\text{Dst} = (\text{Src} \times \text{Dst}) \bmod 2^w$
- Also three-argument multiplication:
  - **imul Src1, Src2, Dst**
  - $\text{Dst} = (\text{Src1} \times \text{Src2}) \bmod 2^w$

# IA32 MULTIPLY, DIVIDE OPERATIONS (2)

- One-argument multiplication:
  - **imull Src** – 32-bit signed multiplication
    - **edx:eax** = Src × **eax**
    - **edx** is top 4 bytes of result, **eax** is bottom 4 bytes of result
  - **mull Src** – 32-bit unsigned multiplication
- One-argument division:
  - **idivl Src** – 32-bit signed division
    - **eax** = **edx:eax** ÷ Src
    - **edx** = **edx:eax mod Src**
  - **divl Src** – 32-bit unsigned division
- Can use **cltd** to set up **edx:eax** for division
  - **cltd** – “convert long-word to double-word”
    - Sign-extends **eax** into **edx**, creating **edx:eax**
    - **Note:** in Intel manual, this instruction is called **cdq**

# IA32 MULTIPLY, DIVIDE OPERATIONS (3)

- Can perform multiplication and division on varying input widths, too
- Examples:
  - **imulw Src** – 16-bit signed multiplication
    - **dx:ax** = Src × ax
  - **idivb Src** – 8-bit signed division
    - **al** = ax ÷ Src
    - **ah** = ax mod Src
- Also variants of **cltd** to set up for division on different input widths
  - **cbtw** – sign-extends al into ax
  - **cwtl** – sign-extends ax into eax
  - **cwtd** – sign-extends ax into dx:ax

# IA32 MULTIPLY/DIVIDE EXAMPLES

- Values:
  - x at location **8 (%ebp)**, y at location **12 (%ebp)**
  - Both signed values, doublewords (4 bytes)

- Compute signed product of x and y

```
movl 8(%ebp), %eax # eax = x
imull 12(%ebp) # edx:eax = x * y
pushl %edx # Save 64-bit result
pushl %eax # onto stack.
```

- Compute signed quotient and remainder of  $x \div y$

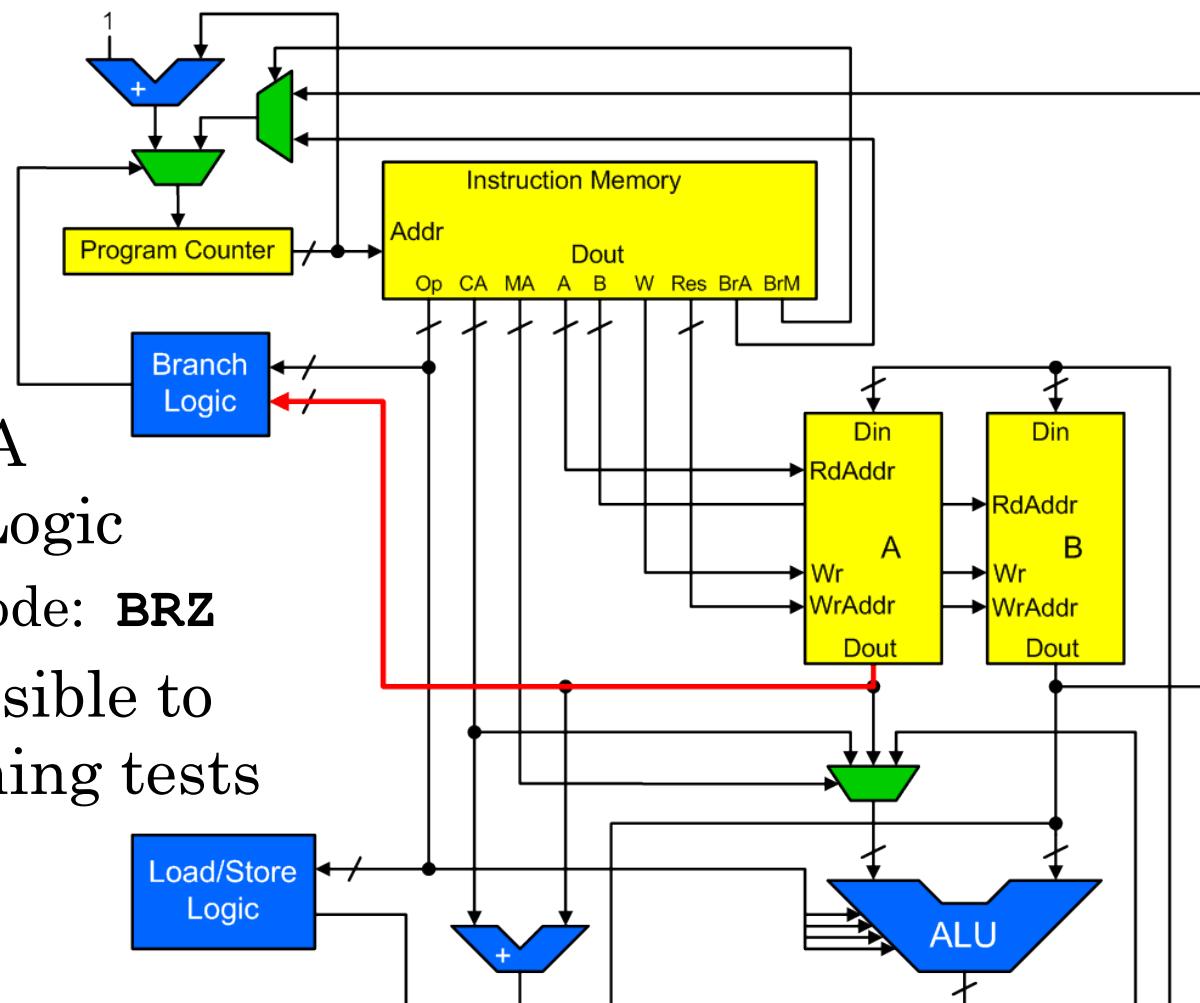
```
movl 8(%ebp), %eax # eax = x
cltd # edx:eax = x
idivl 12(%ebp) # Compute x / y
pushl %eax # eax = quotient
pushl %edx # edx = remainder
```

# IA32 FLOW-CONTROL INSTRUCTIONS

- Many different instructions for branching in IA32
- Simplest version: unconditional jump
  - **jmp Label** Direct jump to address
  - **jmp \*Operand** Indirect jump to address
    - **jmp \*%eax** – jumps to address stored in **eax**
    - **jmp \*(%eax)** – jumps to address stored at M[**eax**]
  - Can use indirect addressing with unconditional jumps! Very useful in many situations:
    - Implementing switch statements: jump tables
    - Object oriented programming: virtual function ptr tables
    - Other flow-control mechanisms in high-level languages
- Other jumps are conditional jumps
  - Jump occurs based on flags in eflags status register

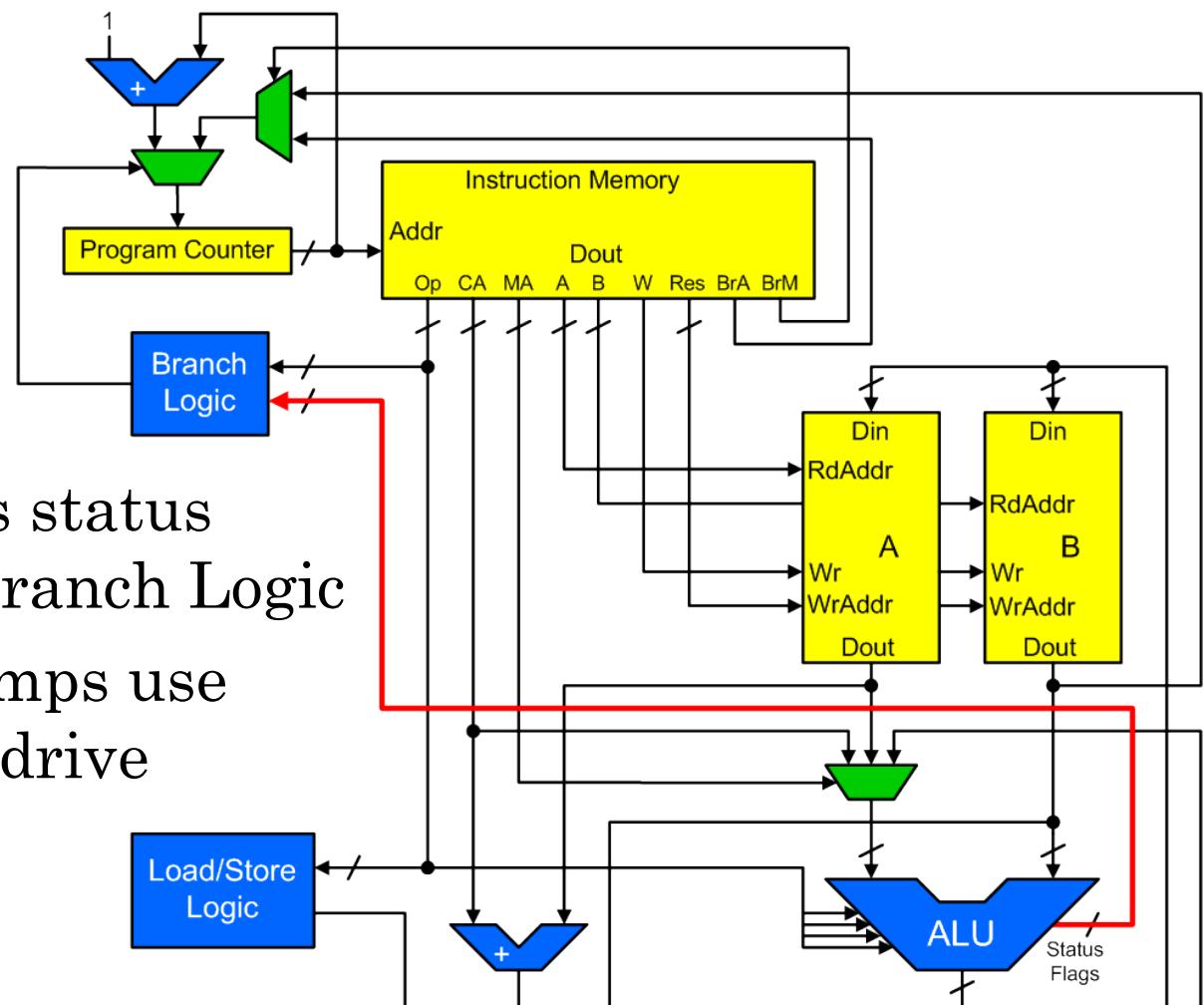
# BRANCH LOGIC, CONDITIONAL JUMPS

- Previous branching logic:



# BRANCH LOGIC, CONDITIONAL JUMPS (2)

- More powerful branching logic:



# IA32 CONDITIONAL OPERATIONS

- Status bits in **eflags** register:
  - CF = carry flag (1 indicates unsigned overflow)
  - SF = sign flag (1 = result is negative)
  - OF = overflow flag (1 indicates signed overflow)
  - ZF = zero flag (1 = result is zero)
- Conditional jump instructions use these flags to control program flow
- All arithmetic and logical operations set these flags
  - Good to know how these instructions affect above flags!
- Can also update these flags with **cmp**, **test**
  - **cmp Src2, Src1**
    - Updates flags as for Src1 – Src2 (i.e. **sub Src2, Src1**)
  - **test Src2, Src1**
    - Updates flags as for Src1 & Src2 (i.e. **and Src2, Src1**)
  - Src1, Src2 are unchanged by comparison/test operation
  - Can specify size prefixes, as usual: **cmpl %ecx, \$0**

# IA32 CONDITIONAL JUMPS

- Conditional jumps can only use a label
  - Can't specify an indirect conditional jump
- Some operations:
  - **je Label** “Jump if equal” ( $ZF = 1$ )
  - **jne Label** “Jump if not equal” ( $ZF = 0$ )
    - **sub Src2, Src1** produces zero result if  $Src1 == Src2$
    - **cmp Src2, Src1** sets zero-flag in this case
  - **js Label** “Jump if sign” ( $SF = 1$ )
  - **jns Label** “Jump if not sign” ( $SF = 0$ )
    - Jump if answer is negative ( $SF = 1$ ) or nonnegative ( $SF = 0$ )
  - **jc/jnc Label** “Jump if [not] carry”
    - Unsigned overflow tests
  - **jo/jno Label** “Jump if [not] overflow”
    - Signed overflow tests

# IA32 SIGNED CONDITIONAL-JUMPS

- **jg Label** “Jump if greater” (signed  $>$ )
  - **jnle** is synonym – “Jump if not less or equal”
  - All comparison opcodes have synonyms like this
- Also:
  - **jge** Jump if greater or equal
  - **jl** Jump if less
  - **jle** Jump if less or equal
- These look at sign flag, overflow flag, zero flag
  - Remember: OF = signed overflow, CF = unsigned overflow
  - ZF indicates if  $\text{Src1} == \text{Src2}$  ( $\text{Src2} - \text{Src1} == 0$ )
  - SF + OF indicate whether  $\text{Src2} - \text{Src1} > 0$  or  $< 0$  (when nonzero)
    - Logic is slightly involved; see CS:APP §3.6.2 for details

# IA32 UNSIGNED CONDITIONAL-JUMPS

- Unsigned comparisons are similar:
  - **ja Label**      “Jump if above” (unsigned >)
    - **jnbe** is synonym – “Jump if not below or equal”
  - Also:
    - **jae**      Jump if above or equal
    - **jb**      Jump if below
    - **jbe**      Jump if below or equal
  - These look at carry flag and zero flag
    - CF indicates whether unsigned overflow occurred from Src2 – Src1
    - If Src2 – Src1 generates unsigned overflow, (CF = 1) then Src2 < Src1
    - Again, ZF indicates if Src1 == Src2

# IA32 CONDITIONAL-SET INSTRUCTIONS

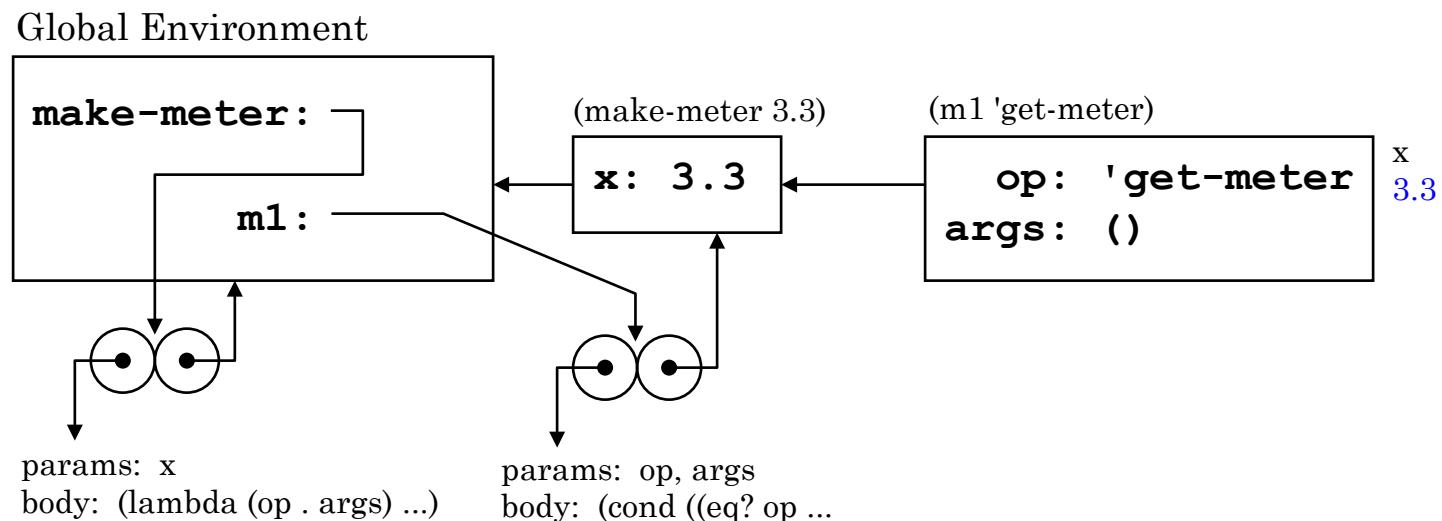
- Also a variety of conditional-set instructions
- Examples:
  - **sete Dst** “Set if equal”
    - Stores ZF into 8-bit target Dst
    - Result is 0 or 1
    - Synonym: **setz** “Set if zero”
  - Others:
    - **sets/setns Dst** “Set if sign” / “Set if not sign”
    - **setg Dst** “Set if greater” (signed >)
    - **setl Dst** “Set if less” (signed <)
    - **seta Dst** “Set if above” (unsigned >)
    - **setb Dst** “Set if below” (unsigned <)
    - etc. (same as for conditional-jump instructions)
- All instructions modify a single 8-bit destination

# BUT WAIT, THERE'S MORE!

- Really only scratched the surface of IA32
  - Covered a lot of what you will see in CS24...
  - ...but there's a *lot* more where that came from!
- The book reading for Week 2 covers several more instructions, and goes into greater detail
  - Chapter 3 – 3.7
- If you see an instruction you don't recognize, look it up in the IA32 manuals (provided on Moodle)
  - If it still doesn't make sense, ask Donnie or a TA ☺

# MORE ADVANCED LANGUAGE FEATURES

- Last time, introduced higher-level abstractions
  - Subroutines, the stack, stack frames, frame pointers
- Many different languages, calling conventions, computational models to choose from!
  - e.g. Scheme environment model allows functions to be created and passed around dynamically
  - When an environment or function is no longer used, it is garbage collected automatically



# C FUNCTIONS

- Start with a simple abstraction: C functions
  - Relatively simple computational model
  - No trapped frames, no lambdas, no garbage collection
- Pretty easy to implement with IA32 assembly
- To implement subroutines (tasks from last time):
  - Need a way to pass arguments and return values between caller and subroutine
  - Need a way to transfer control from caller to subroutine, then return back to caller
  - Need to isolate subroutine's state from caller's state

# EXAMPLE C PROGRAM

- A simple accumulator:
- Uses a global variable to store current value
- Functions to update accumulator, or reset it
- Main function to exercise the accumulator
- Three kinds of variables
  - Global variables
  - Function arguments
  - Local variables

```
int value;

int accum(int n) {
 value += n;
 return value;
}

int reset() {
 int old = value;
 value = 0;
 return old;
}

int main() {
 int i, n;

 reset();
 for (i = 0; i < 10; i++) {
 n = rand() % 1000;
 printf("n = %d\taccum = %d\n",
 n, accum(n));
 }
 return 0;
}
```

## EXAMPLE C PROGRAM (2)

- Three kinds of variables
  - Global variables
  - Function arguments
  - Local variables
- C computational model:
  - (approximately...)
  - A global environment at the top level
  - When a function is called, a new environment is created to hold args, local variables
  - All functions in the file can access the contents of the global environment

```
int value;

int accum(int n) {
 value += n;
 return value;
}

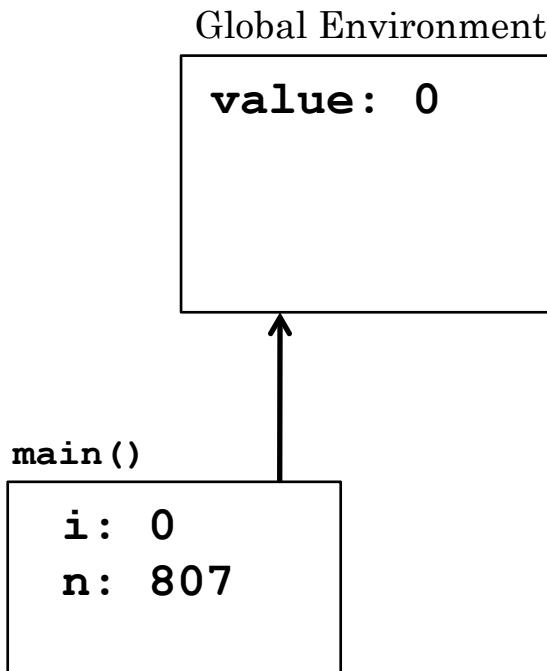
int reset() {
 int old = value;
 value = 0;
 return old;
}

int main() {
 int i, n;

 reset();
 for (i = 0; i < 10; i++) {
 n = rand() % 1000;
 printf("n = %d\taccum = %d\n",
 n, accum(n));
 }
 return 0;
}
```

# EXAMPLE C PROGRAM (3)

- After `reset()` call:
  - Also, `rand()` has returned 807



```
int value;

int accum(int n) {
 value += n;
 return value;
}

int reset() {
 int old = value;
 value = 0;
 return old;
}

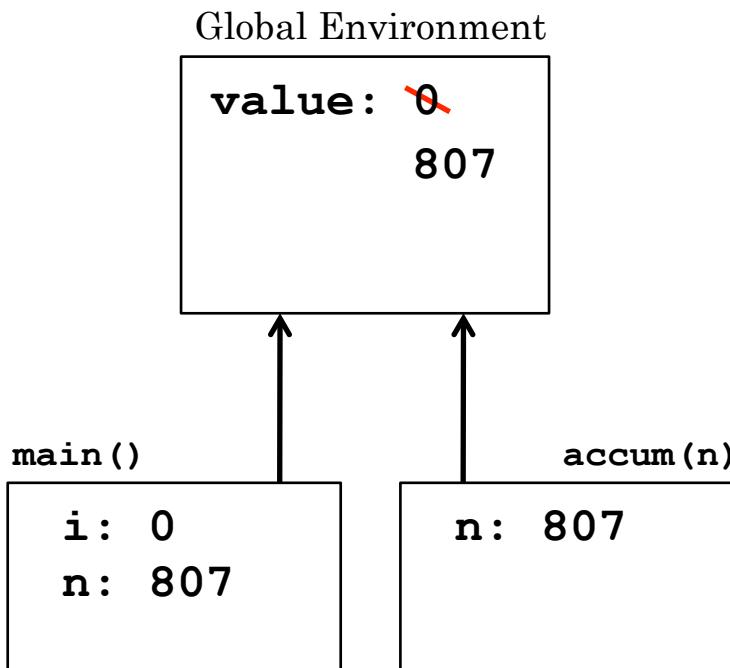
int main() {
 int i, n;

 reset();
 for (i = 0; i < 10; i++) {
 n = rand() % 1000;
 printf("n = %d\taccum = %d\n",
 n, accum(n));
 }
 return 0;
}
```

# EXAMPLE C PROGRAM (4)

- **accum(807)** call:

- Function invocation has its own local environment specifying n = 807



```
int value;

int accum(int n) {
 value += n;
 return value;
}

int reset() {
 int old = value;
 value = 0;
 return old;
}

int main() {
 int i, n;

 reset();
 for (i = 0; i < 10; i++) {
 n = rand() % 1000;
 printf("n = %d\taccum = %d\n",
 n, accum(n));
 }
 return 0;
}
```

# REPRESENTING C MODEL

## ○ Global variables

- Store at specific location
- Reference via absolute address

## ○ Function arguments

- Store on stack
- Pushed by caller before invoking subroutine
- IA32: frame pointer *plus* some offset

## ○ Local variables

- Also store on stack
- Subroutine manages space for these variables
- IA32: frame pointer *minus* some offset

```
int value;

int accum(int n) {
 value += n;
 return value;
}

int reset() {
 int old = value;
 value = 0;
 return old;
}

int main() {
 int i, n;

 reset();
 for (i = 0; i < 10; i++) {
 n = rand() % 1000;
 printf("n = %d\taccum = %d\n",
 n, accum(n));
 }
 return 0;
}
```

# IA32 SUBROUTINE CALLS

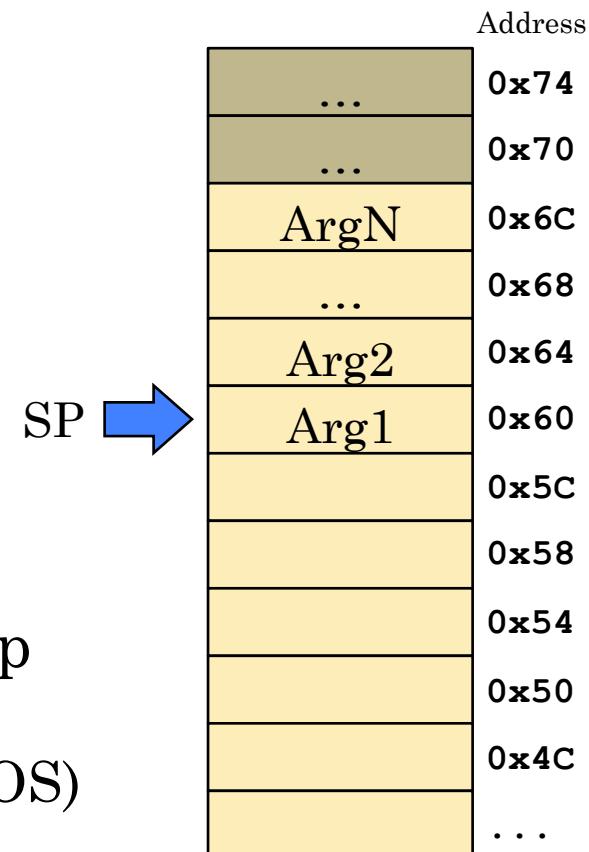
- IA32 provides specific features for subroutines
- Registers:
  - **esp** = stack pointer
    - Stack grows “downward” in memory
    - **push** decrements **esp**, then stores value to **(%esp)**
    - **pop** retrieves value at **(%esp)**, then increments **esp**
  - **ebp** = base pointer
    - IA32 name for frame pointer
- Instructions:
  - **call Addr**
    - Pushes **eip** onto stack (**eip** references *next* instruction)
    - Sets **eip** = Addr
  - **ret**
    - Pops top of stack into **eip**

# IA32 SUBROUTINE CALLS AND `gcc`

- Many different ways to organize stack frames!
- A calling convention is a particular way of passing information to/from a subroutine
- The *cdecl* convention is frequently used on x86 for C subroutines
- Both the procedure caller and the callee have to coordinate the operation!
  - Shared resources: the stack, the register file
- Calling convention specifies:
  - Who sets up which parts of the call
  - What needs to be saved, and by whom
  - How to return values back to the caller
  - Who cleans up which parts of the call

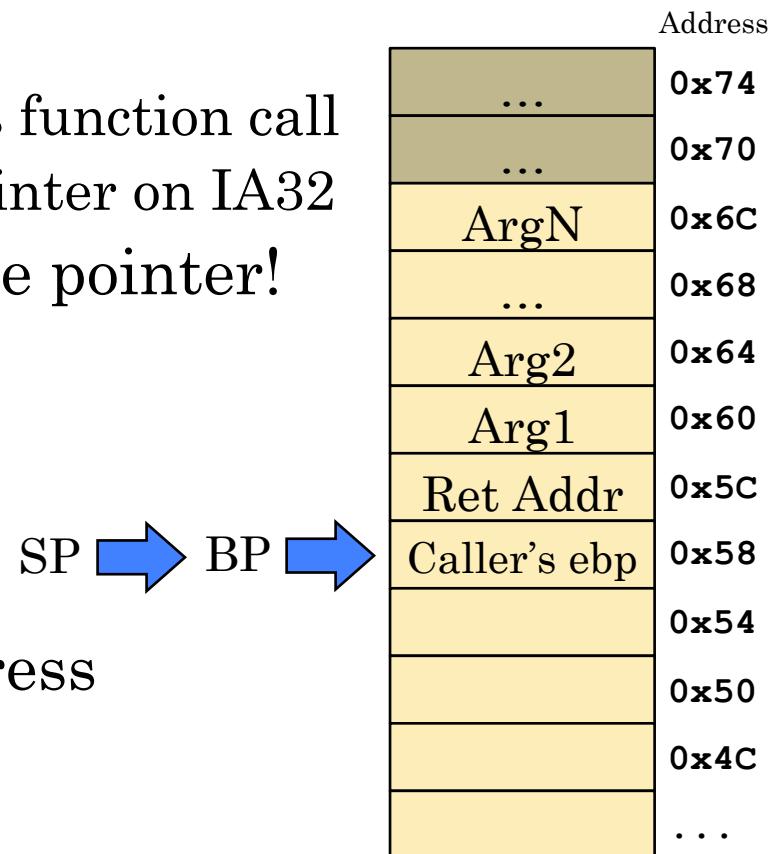
# CDECL: PASSING ARGUMENTS

- Caller is responsible for setting up arguments
- Arguments pushed onto stack in reverse order
  - Last argument is pushed first
  - 2<sup>nd</sup> argument pushed next-to-last
  - 1<sup>st</sup> argument is pushed last
- Two benefits:
  - Earlier arguments have a lower offset added to the frame pointer
  - If procedure is passed more args than it expects, it doesn't break the procedure's code
- Primitive values generally take up a doubleword (4 bytes) on stack
  - e.g. ints, floats, pointers (on 32-bit OS)



# CDECL: INVOKING THE PROCEDURE

- Caller uses **call** to invoke the procedure
  - Pushes **eip** of *next* instruction onto the stack
- First task of callee:
  - Set up frame pointer for this function call
  - **ebp** is used for the frame pointer on IA32
- Must preserve caller's frame pointer!
- Typical code:
  - **pushl %ebp**
  - **movl %esp, %ebp**
- Now:
  - **4(%ebp)** = Return address
  - **8(%ebp)** = Arg1 value
  - **12(%ebp)** = Arg2 value



# CDECL: SAVING REGISTERS

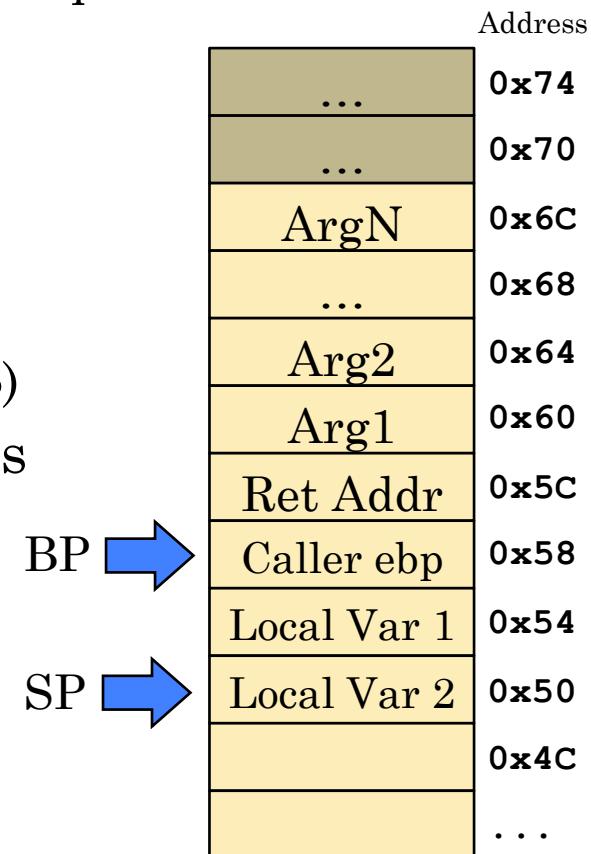
- “Callee must save **ebp** before it modifies it”
- A general issue:
  - The register file is a shared resource
  - Calling convention must specify how registers are managed
- Callee-save registers:
  - When callee returns, values *must* be same as when subroutine was invoked
  - **ebp**, **ebx**, **esi**, **edi** are callee-save registers
- Caller-save registers:
  - Callee may change these registers without saving them!
  - The *caller* must save these registers before the call, if the old values need to be preserved
  - **eax**, **ecx**, **edx** are caller-save registers

# CDECL: RETURNING RESULTS

- For now, only consider simple results
  - e.g. **int** or pointer
- In these cases, callee returns the result in **eax**
  - Set **eax** to result, restore **ebp**, then return to caller
- Who removes the arguments from the stack??
- In cdecl, the *caller* cleans up stack
  - Linux / GNU calling convention
  - e.g. can add a constant to **esp** to remove arguments
- In stdcall (Win32), the *callee* cleans up stack
  - Microsoft Visual C++ calling convention
  - IA32 includes version of **ret** that takes an argument
  - **ret n**
    - Sets **eip** to **(%esp)**, then pops **n** bytes off stack

# LOCAL VARIABLES

- Procedures sometimes need space for local variables
  - Compiler figures out how much space, from the source code
  - Sometimes allocates more than is strictly required
- Local variables reside just below the frame pointer
  - Accessed via `-off(%ebp)`
- Common pattern:
  - Allocate  $n$  bytes on stack for local vars
  - `subl $n, %esp` (or `addl $-n, %esp`)
- Example: allocate 8 bytes for local vars
  - `subl $8, %esp`
- Note: these memory locations are *not* initialized!
  - Contains whatever values were in that memory before the call...

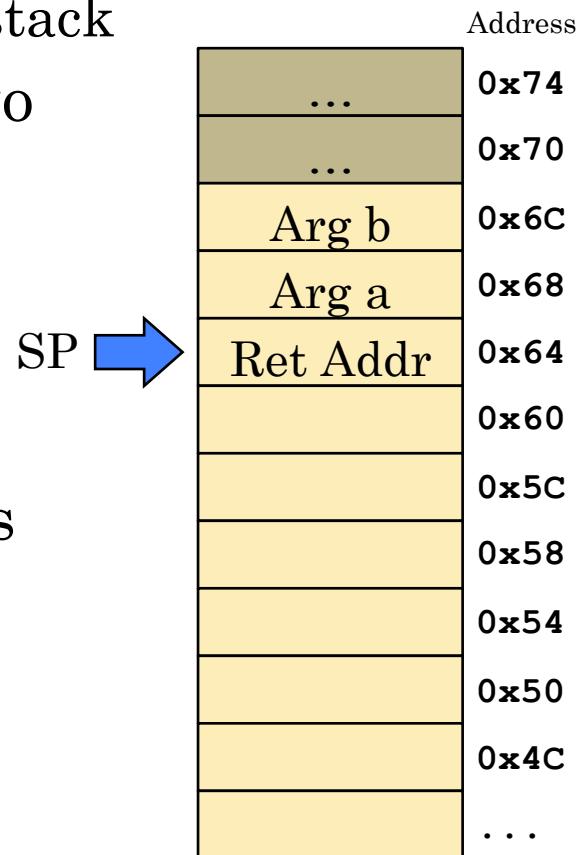


# CDECL AND FRAME POINTER

- Note: subroutines don't *always* use **%ebp**
  - **%ebp** is primarily used when a function manages its own local state on the stack
- Example: a function that adds two values, and returns the result

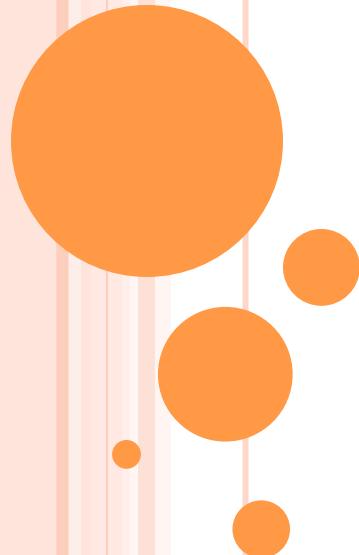
```
int add(int a, int b) {
 return a + b;
}
```

- Doesn't have any local variables!
- In this case, subroutine can access args **a** and **b** directly, via **%esp**:
  - **a** can be accessed via **4(%esp)**
  - **b** can be accessed via **8(%esp)**
  - Return address is at **(%esp)**



## NEXT TIME

- Look at how our simple C accumulator program is implemented in IA32 assembly language
  - Memory layout strategy for global variables, local variables, and arguments
  - `gcc` and the cdecl calling convention
- Begin to look at other C language features
  - C flow-control statements, and how they are translated into IA32 assembly



# CS24: INTRODUCTION TO COMPUTING SYSTEMS

Spring 2015  
Lecture 6

## LAST TIME: CDECL

- How to implement basic C abstractions in IA32?
  - C subroutines with arguments, local/global variables
- Began discussing the *cdecl* calling convention
  - Widely used on \*NIX systems running on x86
- Both the procedure caller and the callee have to coordinate the operation!
  - Shared resources: the stack, the register file
- Calling convention specifies:
  - Who sets up which parts of the call
  - What needs to be saved, and by whom
  - How to return values back to the caller
  - Who cleans up which parts of the call

# CDECL CHEAT SHEET (1)

- Caller pushes arguments in reverse order
- Caller uses **call** to invoke subroutine
- Callee pushes caller's **%ebp** onto stack, then sets **%ebp = %esp**

```
pushl %ebp
movl %esp, %ebp
```

- Arguments are at positive offsets from **%ebp**

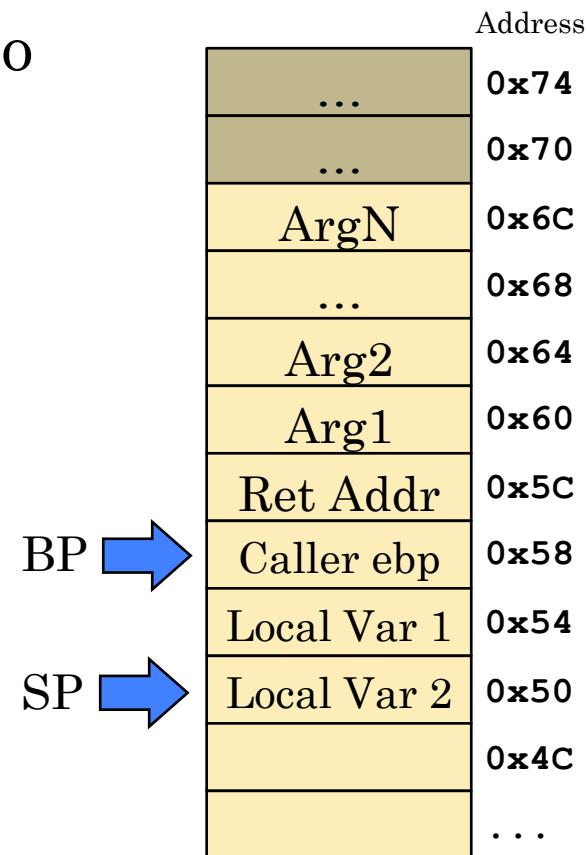
8 (%ebp) = Arg1

12 (%ebp) = Arg2

- Local variables at negative offsets from **%ebp**

-4 (%ebp) = Local Var 1

-8 (%ebp) = Local Var 2

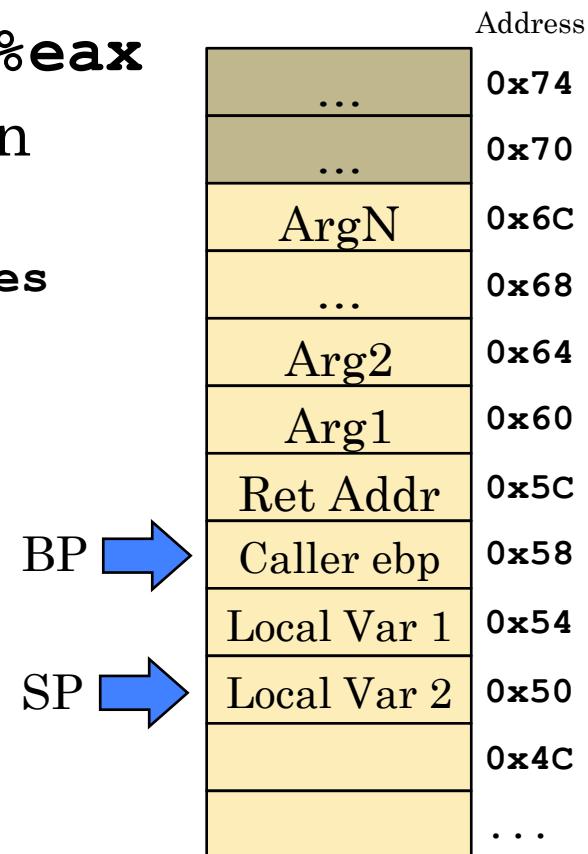


# CDECL CHEAT SHEET (2)

- Caller-save registers: **%eax**, **%ecx**, **%edx**
- Callee-save registers: **%ebp**, **%ebx**, **%esi**, **%edi**
- Callee saves return-value into **%eax**
- Callee restores stack state, then uses **ret** to return

```
Also discards local variables
movl %ebp, %esp
popl %ebp
ret
```

- Caller removes arguments from the stack
  - Either using **pop** instructions, or by adding a constant to **%esp**



# BACK TO OUR EXAMPLE C PROGRAM

- A simple accumulator:
- Uses a global variable to store current value
- Functions to update accumulator, or reset it
- Main function to exercise the accumulator
- *How is this program implemented in IA32?*

```
int value;

int accum(int n) {
 value += n;
 return value;
}

int reset() {
 int old = value;
 value = 0;
 return old;
}

int main() {
 int i, n;

 reset();
 for (i = 0; i < 10; i++) {
 n = rand() % 1000;
 printf("n = %d\taccum = %d\n",
 n, accum(n));
 }
 return 0;
}
```

# OUR EXAMPLE PROGRAM

- Can look at **gcc** assembly language output for our accumulator example
  - **gcc -O2 -S amain.c**
  - **-S** generates assembly output, not a binary file
    - Result is in **amain.s**
  - **-O2** applies some optimizations to generated code
    - Otherwise, assembly output includes some pretty silly code
- Results vary *widely* based on target platform!
  - We will look at Linux **gcc** output
  - (MacOS X output is *very* different... Ask if you want an explanation of what's going on. It's very cool!)

# GENERATED ASSEMBLY CODE

- Some of the output:
- Lines starting with . are assembler directives
  - e.g. **.text** tells assembler to generate machine code for instructions that follow
- Lines with a colon are labels
  - e.g. **accum**, **reset** are labels specifying start of our functions
- **.size** directive specifies the size of various symbols, in bytes
  - **accum** = address of function's start
  - . = current address
  - **.-accum** is size of fn. body

```
.file "amain.c"
.text
.p2align 4,,15
.globl accum
.type accum, @function
accum:
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %eax
addl value, %eax
popl %ebp
movl %eax, value
ret
.size accum, .-accum
.p2align 4,,15
.globl reset
.type reset, @function
reset:
pushl %ebp
movl value, %eax
...
```

# GLOBAL VARIABLES

- End of our output:
- **.size main, .-main**  
is end of **main()** function

```
...
popl %esi
popl %ebp
leal -4(%ecx), %esp
ret
.size main, .-main
.comm value,4,4
.ident "GCC: (GNU) ...
.section ...
```

- Global variable **value** specified with **.comm** directive
  - A “common symbol,” possibly shared across multiple files
  - Specifies name, size, optional alignment of variable
  - Address is assigned when assembling the code
  - The actual memory is uninitialized

# ACCUMULATOR CODE

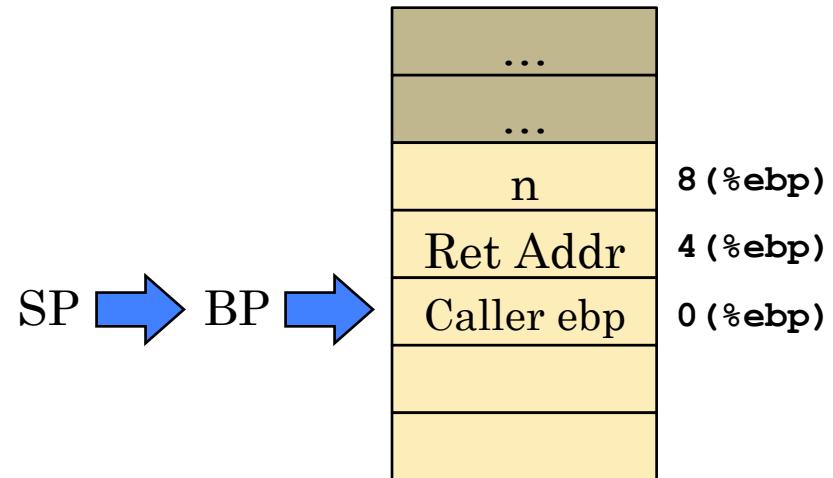
- Accumulator function:

```
int accum(int n) {
 value += n;
 return value;
}
```

- Translated into:

accum:

|                    |                       |
|--------------------|-----------------------|
| pushl %ebp         | # Set up stack frame  |
| movl %esp, %ebp    |                       |
| movl 8(%ebp), %eax | # Move n into eax     |
| addl value, %eax   | # eax += value        |
| popl %ebp          | # Restore caller ebp  |
| movl %eax, value   | # Store updated value |
| ret                |                       |



# RESET CODE

- Reset function:

```
int reset() {
 int old = value;
 value = 0;
 return old;
}
```

- Translated into:

```
reset:
 pushl %ebp
 movl value, %eax # Result goes into eax
 movl %esp, %ebp
 popl %ebp
 movl $0, value
 ret
```

- Clearly involves some unnecessary steps...
  - **ebp** isn't used at all! Could reduce down to 3 instructions.

# MAIN FUNCTION

- Main function code:

```
int main() {
 int i, n;

 reset();

 for (i = 0; i < 10; i++) {
 n = rand() % 1000;
 printf("n = %d\taccum = %d\n",
 n, accum(n));
 }

 return 0;
}
```

# MAIN FUNCTION (2)

- Main function code:

```
int main() {
 int i, n;

 reset();
 ...
}
```

- Assembly code:

```
main:
 leal 4(%esp), %ecx # Stack init: aligns stack with
 andl $-16, %esp # 16-byte boundary, then
 pushl -4(%ecx) # copies return-addr to TOS.
 pushl %ebp # Set up stack frame
 movl %esp, %ebp
 pushl %esi # Callee-save registers
 xorl %esi, %esi # %esi is i; sets i = 0.
 pushl %ebx
 pushl %ecx
 subl $12, %esp # Alloc space for fn. args
 call reset # Clear accumulator value
 ...
}
```

# MAIN FUNCTION (3)

- Main function code, cont.

```
for (i = 0; i < 10; i++) {
 n = rand() % 1000;
 printf("n = %d\taccum = %d\n",
 n, accum(n));
}
```

- Assembly code:

- **esi** is loop variable **i**
- **ebx** is **n**
- **.L6** is start of loop

- **rand()** **% 1000** implemented in a *very* unintuitive way...

- Integer division/modulus with a constant can be implemented as multiplication
- (See the book Hacker's Delight)

.L6:

```
call rand
movl $274877907, %edx
addl $1, %esi
movl %eax, %ecx
imull %edx
movl %ecx, %eax
sarl $31, %eax
movl %ecx, %ebx
sarl $6, %edx
subl %eax, %edx
imull $1000, %edx, %edx
subl %edx, %ebx
movl %ebx, (%esp)
call accum
movl %ebx, 4(%esp)
movl $.LC0, (%esp)
movl %eax, 8(%esp)
call printf
cmpl $10, %esi
jne .L6
...
```

# MAIN FUNCTION (4)

- Main function code, cont.

```
for (i = 0; i < 10; i++) {
 n = rand() % 1000;
 printf("n = %d\taccum = %d\n",
 n, accum(n));
}
```

- Calls to **accum(n)**, **printf(...)**

- Note that **gcc** doesn't explicitly push arguments onto stack!

- Also doesn't pop off stack when done

- This is a compiler optimization

- Why do the pushes and pops, when it can be faked? ☺
- **gcc** allocates extra space on the stack to speed up these calls

.L6:

```
call rand
movl $274877907, %edx
addl $1, %esi
movl %eax, %ecx
imull %edx
movl %ecx, %eax
sarl $31, %eax
movl %ecx, %ebx
sarl $6, %edx
subl %eax, %edx
imull $1000, %edx, %edx
subl %edx, %ebx
movl %ebx, (%esp)
call accum
movl %ebx, 4(%esp)
movl $.LC0, (%esp)
movl %eax, 8(%esp)
call printf
cmpl $10, %esi
jne .L6
...
```

# MAIN FUNCTION (5)

- Main function code, cont.

```
for (i = 0; i < 10; i++) {
 n = rand() % 1000;
 printf("n = %d\taccum = %d\n",
 n, accum(n));
}
```

- Also need a string constant to pass to **printf()**
- Before **main()** code:

```
.LC0:
.string "n = %d\taccum = %d\n"
```

- **as** copies this data to the output binary file
- Address of data is **.LC0**

```
.L6:
call rand
movl $274877907, %edx
addl $1, %esi
movl %eax, %ecx
imull %edx
movl %ecx, %eax
sarl $31, %eax
movl %ecx, %ebx
sarl $6, %edx
subl %eax, %edx
imull $1000, %edx, %edx
subl %edx, %ebx
movl %ebx, (%esp)
call accum
movl %ebx, 4(%esp)
movl $.LC0, (%esp)
movl %eax, 8(%esp)
call printf
cmpl $10, %esi
jne .L6
...
```

# CALLING CONVENTION AND RECURSION

- The cdecl calling convention:
  - Each function call has its own region of the stack
    - Caller pushes arguments onto stack, then calls the callee
    - Callee saves caller's frame pointer, then sets up its own frame pointer
    - Callee stores its local variables after the frame pointer
    - When callee returns to caller, stack is restored to prev state
- This calling convention easily supports recursion
- A procedure can call itself:
  - Each recursive invocation of the procedure will have its own stack space, as long as the conventions are followed!
- You get to explore this more on Assignment 2! ☺

# C FLOW-CONTROL STATEMENTS

- C provides a variety of flow-control statements
  - **if** statements

```
if (test-expr)
 then-statement
else
 else-statement
```
  - **while** loops, **for** loops, **do** loops

```
while (test-expr)
 body-statement
```
- Conceptually straightforward to use in your C programs
- How are these normally translated to IA32 assembly language?
  - Helps us better understand what the compiler generates
  - Also helps us know how to write them in IA32 ourselves!

# C FLOW-CONTROL STATEMENTS (2)

- C flow-control statements implemented in IA32 using a combination of conditional and unconditional jumps
- Example: **if** statements

```
if (test-expr)
 then-statement ;
else
 else-statement ;
```

- A common translation:

```
t = test-expr ;
if (t)
 goto true_branch;
else-statement ;
goto done;
true_branch:
then-statement ;
done:
```

- Compiler frequently optimizes/rearranges this flow

# DO-WHILE LOOPS

- o **do**-loops not used as frequently in programs, but very easy to implement in assembly language

- Requires minimum number of branching operations

**do**

*body-statement*

**while** (*test-expr*);

- o A simple translation:

**loop:**

*body-statement* ;

**t** = *test-expr* ;

**if** (**t**)

**goto loop**;

# WHILE LOOPS

- **while**-loops are much more common, but more involved at the assembly-language level

```
while (test-expr)
 body-statement ;
```

- One translation:

```
loop:
 t = test-expr ;
 if (!t)
 goto done ;
 body-statement ;
 goto loop ;
done :
```

- Problem: This code is slow to execute.
- Branching has a *big* performance impact
  - Affects instruction caching and pipelining

## WHILE LOOPS (2)

- A faster implementation “peels off” the first test

```
while (test-expr)
 body-statement ;
```

- Equivalent to:

```
if (!test-expr)
 goto done ;
do
 body-statement ;
while (test-expr)
done :
```

- Translating this to assembly code yields a loop body containing only one branching operation

# WHILE LOOPS (3)

- Our original while-loop:

```
while (test-expr)
 body-statement ;
```

- Completing the translation:

```
t = test-expr ;
if (!t)
 goto done;

loop:
 body-statement ;
 t = test-expr ;
 if (t)
 goto loop;

done:
```

# FOR LOOPS

- **for**-loops are more sophisticated **while** loops:

```
for (init-expr ; test-expr ; update-expr)
 body-statement ;
```

- Equivalent to:

```
init-expr ;
while (test-expr) {
 body-statement ;
 update-expr ;
}
```

- We know how to translate these components into assembly language

- Transform **while** loop into **do-while** loop
- Insert additional **for**-loop operations into appropriate places

## FOR LOOPS (2)

- Implementing **for** loops:

```
for (init-expr ; test-expr ; update-expr)
 body-statement ;
```

- Translate into:

```
init-expr ;
t = test-expr ;
if (!t)
 goto done;

loop:
 body-statement ;
 update-expr ;
 t = test-expr ;
 if (t)
 goto loop;

done:
```

# MORE ADVANCED PROGRAMS...

- We can now map basic C programs into IA32 assembly language
  - ...including programs that use recursion
- What about this problem?
  - Write a function that takes an argument  $n$
  - Return a collection of all prime numbers  $\leq n$
  - e.g. `int * find_primes(int n)`
- Don't yet have the tools to implement this!
  - Requires a variable amount of memory
  - Memory lifetime must extend beyond a single function call

# DYNAMIC ALLOCATION AND HEAP

- When programs need a variable amount of memory, they can allocate it from the heap
  - A large, resizable pool of memory for programs to use
  - Programs can request blocks of memory from the heap
  - When finished, programs release blocks back to the heap
  - This is a *run-time facility* for programs to utilize
- For C programs, standard functions to support heap:
  - Allocate a block of memory using **malloc()**
    - **void \* malloc(size\_t size)**
    - Returns pointer to block of memory with specified size, or **NULL** if **size** bytes are not available
    - **size\_t** is an unsigned integer data type
  - Release a block of memory using **free()**
    - **void free(void \*ptr)**
    - Specified memory block is returned to heap

# HEAP AND STACK

- Programs and stack usage:
  - Stack space automatically reclaimed when function returns
    - Stack values can last for up to the lifetime of a procedure call
  - Procedures are specifically encoded to use the stack
    - Explicit accesses relative to base pointer **ebp**
    - Adjustments to stack pointer to allocate/release space
  - Stack space used by a procedure doesn't vary substantially during its execution
    - Set of local variables within each code block is fixed
    - Set of arguments passed to a procedure is also fixed
- Programs and heap usage:
  - Memory required often depends on the input values
    - Can vary quite dramatically!
  - Programs must *explicitly* allocate and release blocks of memory on the heap

# SIMPLE EXAMPLE: VECTOR ADDITION

- Variation on an example from lecture 3:

```
int * vector_add(int a[], int b[], int length) {
 int *result;
 int i;

 result = malloc(length * sizeof(int));
 for (i = 0; i < length; i++)
 result[i] = a[i] + b[i];

 return result;
}
```

- Now the procedure dynamically allocates a result vector from heap, then sums inputs into this memory
- Now that we understand IA32 more deeply, let's explore how to implement this function

# POINTERS AND ARRAYS

- Our vector-add function:

```
int * vector_add(int a[], int b[], int length) {
 int *result;
 int i;

 result = (int *) malloc(length * sizeof(int));
 for (i = 0; i < length; i++)
 result[i] = a[i] + b[i];

 return result;
}
```

- Clear that pointers and arrays are closely related
  - Declare **result** as **int\***
  - Access it as an array, just like **a** and **b**

# ARRAYS IN C

- C arrays are collections of elements, all of same data type
  - Array elements are contiguous in memory
  - Elements are accessed by indexing into the array
- For an array declaration: **T A[N]**
  - **T** is the data type
  - **A** is the array's variable name
  - **N** is the number of elements
- C allocates a contiguous region of memory for the array
  - Allocates **N × sizeof(T)** bytes for the array
  - **sizeof(type)** is a standard C operator that returns the size of the specified data type, in bytes
    - e.g. **sizeof(int) = 4** for **gcc** on IA32
  - **sizeof(type)** is resolved to a value at compile-time
- **A** holds a pointer to the start of the array
  - Can use **A** to access various elements of the array

## ARRAYS IN C (2)

- For an array declaration:  $T \ A[N]$ 
  - $T$  is the data type
  - $A$  is the array's variable name
  - $N$  is the number of elements
- $A$  holds a pointer to the start of the array
  - Can use  $A$  to access various elements of the array
- What address does array-element  $i$  reside at?
  - $A$  points to first element in array
  - Each element is **sizeof**( $T$ ) bytes in size
  - Element  $i$  resides at address  $A + \text{sizeof}(T) * i$
- This is what the C array-index operator `[]` does
  - $A[i]$  computes index of element  $i$ , then reads/writes the element

# POINTERS AND ARRAYS IN C

- C also supports pointer arithmetic
  - Similar idea to array indexing
- For a C pointer variable: **T \*p**
- Adding 1 to **p** advances it one *element*, not one byte
  - e.g. for **int \*p**, **p + 1** actually advances **p** by 4 bytes
- Adding/subtracting an offset *N* from a pointer to **T** will move forward or backward *N* elements
- Implication:
  - **A[i]** is identical to saying **\* (A + i)**
  - **A** is a pointer to first element in array
  - **A + i** moves forward **i** elements (*not i bytes!*)
  - **A + i** is a *pointer* to element **i**, so dereference to get to the actual element **A[i]**

# VARIATION ON THEME

- Our original function:

```
int * vector_add(int a[], int b[], int length) {
 int *result;
 int i;

 result = (int *) malloc(length * sizeof(int));
 for (i = 0; i < length; i++)
 result[i] = a[i] + b[i];

 return result;
}
```

- Could also write something crazy like this:

```
int * vector_add(int *a, int *b, int length) {
 int *result, *elem;

 result = (int *) malloc(length * sizeof(int));
 for (elem = result; length != 0; length--) {
 *elem = *a + *b;
 elem++; a++; b++;
 }
 return result;
}
```

- Optimizing compilers may generate code like this
- Take advantage of C's equivalence between arrays and pointers

# VECTOR-ADD AND IA32

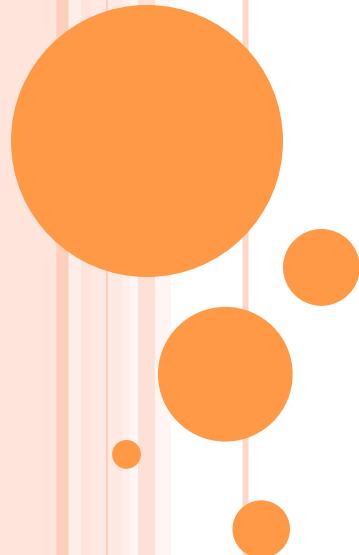
- How do we implement this function in IA32?

```
int * vector_add(int *a, int *b, int length) {
 int *result;
 int i;

 result = (int *) malloc(length * sizeof(int));
 for (i = 0; i < length; i++)
 result[i] = a[i] + b[i];

 return result;
}
```

- Next time, will go through the entire process of implementing this, from scratch. ☺



# CS24: INTRODUCTION TO COMPUTING SYSTEMS

Spring 2015  
Lecture 7

# LAST TIME

- Dynamic memory allocation and the heap:
  - A *run-time facility* that satisfies multiple needs:
    - Programs can use widely varying, possibly large amounts of memory for computations
    - Allocated memory is available beyond a single function call
  - Allocate with: `void * malloc(size_t size)`
  - Release with: `void free(void *ptr)`
- C arrays:  $T \ A[N]$ 
  - Allocates  $N \times \text{sizeof}(T)$  bytes for the array
  - Element  $i$  resides at address  $\mathbf{A} + \text{sizeof}(T) * i$
  - $\mathbf{A}[i]$  is equivalent to  $*(\mathbf{A} + i)$ 
    - Adding  $i$  to a pointer moves forward  $i$  elements, not  $i$  bytes
    - C infers the element size from the pointer's type

# TODAY: IMPLEMENTING VECTOR-ADD

- How to implement this function in IA32?

```
int * vector_add(int *a, int *b, int length) {
 int *result;
 int i;

 result = (int *) malloc(length * sizeof(int));
 for (i = 0; i < length; i++)
 result[i] = a[i] + b[i];

 return result;
}
```

- Note: This example works on Linux.
  - e.g. MacOS X introduces many additional complexities
    - Prepends an underscore \_ on public symbol names
    - Must use relative/indirect references to external symbols
- If on a confusing platform, use **gcc -S** on test code to figure out what's needed

# VECTOR-ADD ARGUMENTS

- How to implement this function in IA32?

```
int * vector_add(int *a, int *b, int length) {
 int *result;
 int i;

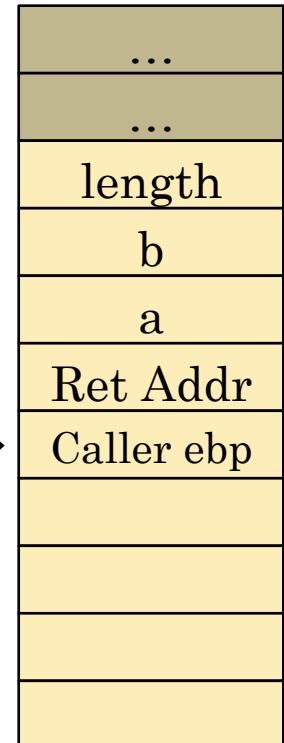
 result = (int *) malloc(length * sizeof(int));
 for (i = 0; i < length; i++)
 result[i] = a[i] + b[i];

 return result;
}
```

- Using cdecl calling convention:

- **a, b, length** pushed by caller; *ebp + offset* locations
- Arguments are pushed last-to-first...
- **a** stored at: 8(%ebp)
- **b** stored at: 12(%ebp)
- **length** stored at: 16(%ebp)

ebp →



# IMPLEMENTING VECTOR-ADD (1)

- Initial code:

- Put in placeholders where implementation is unknown

```
vector_add:
 # a = 8(%ebp)
 # b = 12(%ebp)
 # length = 16(%ebp)
 pushl %ebp # Save caller ebp
 movl %esp, %ebp # Set up stack frame ptr

 # TODO: Save callee-save registers we alter

 # TODO: Allocate space for result with malloc

 # TODO: Implement vector-sum loop

 # TODO: Restore callee-save registers we alter

 # Clean up stack before returning to caller
 movl %ebp, %esp # Remove local vars
 popl %ebp # Restore caller ebp
 ret
```

# CALLING MALLOC

- Next task is to call **malloc()** with array size

```
int * vector_add(int *a, int *b, int length) {
 int *result;
 int i;

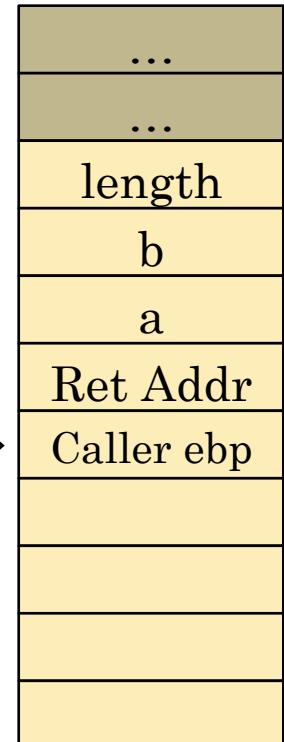
 result = (int *) malloc(length * sizeof(int));
 for (i = 0; i < length; i++)
 result[i] = a[i] + b[i];

 return result;
}
```

- **malloc()** also uses cdecl calling convention

- Push **length \* 4** onto stack
- Call **malloc()** subroutine
- Memory address will be returned in **eax**

- We want to return this memory, so leave **malloc()**'s result in **eax** and return it to our caller as well



# IMPLEMENTING VECTOR-ADD (2)

- Continuing our code:

`vector_add:`

```
a = 8(%ebp)
b = 12(%ebp)
length = 16(%ebp)
...
Allocate space for result with malloc
movl 16(%ebp), %ecx
shll $2, %ecx # ecx = length * 4
pushl %ecx
call malloc # allocate space for result
popl %ecx # cdecl: caller removes args

TODO: Implement vector-sum loop
```

- We return this memory to caller; leave pointer in **eax**

# IMPLEMENTING OUR LOOP

- Next, implement our loop!

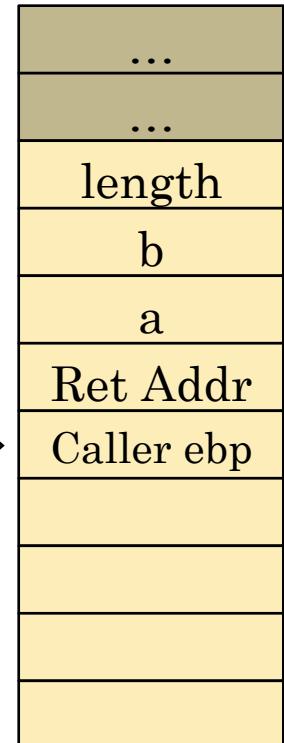
```
int * vector_add(int *a, int *b, int length) {
 int *result;
 int i;

 result = (int *) malloc(length * sizeof(int));
 for (i = 0; i < length; i++)
 result[i] = a[i] + b[i];

 return result;
}
```

- Need a loop variable
  - Use **esi**. Compare directly to **length**, 16 (%**ebp**)
- Also need to access element  $i$  of arrays **a** and **b**
  - Use scaled, indexed memory addressing
  - $(R_b, R_i, s)$  syntax accesses  $M[R_b + R_i \times s]$
  - Need to store start-addresses of **a**, **b** into registers too

ebp →



# IMPLEMENTING VECTOR-ADD (3)

- Continuing our code:

```
vector_add:
```

```
 # a = 8(%ebp)
```

```
 # b = 12(%ebp)
```

```
 # length = 16(%ebp)
```

```
 ...
```

```
 # Implement vector-sum loop
```

```
 movl 8(%ebp), %ebx # Start of a
```

```
 movl 12(%ebp), %ecx # Start of b
```

```
 movl $0, %esi # Loop variable i
```

- Now we need to implement our **for**-loop

- for**-loop = **while**-loop + initialization and update

- Turn **while**-loop into **do**-loop by factoring out first test

# IMPLEMENTING VECTOR-ADD (4)

- Continuing our code:

```
...
Implement vector-sum loop
movl 8(%ebp), %ebx # Start of a
movl 12(%ebp), %ecx # Start of b
movl $0, %esi # Loop variable i

cmpl 16(%ebp), %esi # First for-test
jge vadd_done # Is i >= length ?

vadd_loop:
 movl (%ebx, %esi, 4), %edx # edx = a[i]
 addl (%ecx, %esi, 4), %edx # edx += b[i]
 movl %edx, (%eax, %esi, 4) # r[i] = edx

 incl %esi # i++
 cmpl 16(%ebp), %esi # Subsequent for-tests
 jl vadd_loop # Is i < length ?

vadd_done:
```

# LOOP CONDITIONS

- AT&T syntax is really confusing for conditions
- C code: **for (i = 0; i < length; i++)**
- Assembly code:

```
...
 cmpl 16(%ebp), %esi # First for-test
 jge vadd_done # Is i >= length ?

vadd_loop:
 movl (%ebx, %esi, 4), %edx # edx = a[i]
 addl (%ecx, %esi, 4), %edx # edx += b[i]
 movl %edx, (%eax, %esi, 4) # r[i] = edx

 incl %esi # i++
 cmpl 16(%ebp), %esi # Subsequent for-tests
 jl vadd_loop # Is i < length ?

vadd_done:
```

- **cmpl Src2, Src1** sets flags as for  $\text{Src1} - \text{Src2}$ 
  - For  $A op B$ , generally want to specify **cmp B, A**
  - This way, the conditional jump's type mirrors the C code

# FINISHING UP THE IMPLEMENTATION

- Just need to save/restore our callee-save registers
- Code:

```
vector add:
 # a = 8(%ebp)
 # b = 12(%ebp)
 # length = 16(%ebp)
 pushl %ebp # Save caller ebp
 movl %esp, %ebp # Set up stack frame ptr

 # Save callee-save registers.
 pushl %ebx
 pushl %esi

 ... (rest of implementation)

 # Restore callee-save registers.
 popl %esi
 popl %ebx

 # Clean up stack before returning to caller
 movl %ebp, %esp # Remove local vars
 popl %ebp # Restore caller ebp
 ret
```

# USING OUR VECTOR-ADD CODE

- Save assembly code into `vector_add.s`
- To make function callable, need to put this at top:

```
Function to add two vectors together.
(bla bla bla)
.globl vector_add
vector_add:
 ...
```

- `.globl` makes the symbol visible to other functions
- Many other directives you can add, but this is minimal to make the function callable.
- Assemble our function into an object file:
  - `as -o vector_add.o vector_add.s`
  - Can also include debug info by adding `-g` to args

# USING OUR VECTOR-ADD CODE (2)

- A simple test program, `va_main.c`:

```
#include <stdio.h>
#include <stdlib.h>

/* Normally would declare this in file vector_add.h */
int * vector_add(int *a, int *b, int length);

int main() {
 int a[] = {1, 2, 3, 4, 5};
 int b[] = {6, 7, 8, 9, 10};
 int *res;
 int i;

 res = vector_add(a, b, 5);
 for (i = 0; i < 5; i++)
 printf("res[%d] = %d\n", i, res[i]);

 free(res);
 return 0;
}
```

## USING OUR VECTOR-ADD CODE (2)

- Compile `va_main.c`, link with `vector_add.o`

```
gcc -c va_main.c
```

```
gcc -o vadd va_main.o vector_add.o
```

- Run the program:

```
[user@host:~]> ./vadd
```

```
res[0] = 7
```

```
res[1] = 9
```

```
res[2] = 11
```

```
res[3] = 13
```

```
res[4] = 15
```

- Success! ☺

- And if not, `gdb` is always available for debugging...

# VECTOR-ADD FUNCTION

- Using scaled, indexed memory access mode, very easy to implement array indexing in IA32
  - Once array's base address and index are in registers, very easy to retrieve/store a particular element
- Can also call C runtime functions from assembly
  - Called **malloc()** function by adhering to calling conventions
- Returned a heap-allocated chunk of memory to our caller
  - Now our functions can return results larger than **eax**
  - Results also last longer than a single procedure call since they aren't allocated on the stack
  - The program has to free the memory when it's done...

# FINDING PRIMES

- Our other program to return primes:
  - Write a function that takes an argument  $n$
  - Returns a collection of all prime numbers  $\leq n$
  - e.g. `find_primes(int n)`
- Size of result depends on inputs
- Need to build up result as we go
- Might be better to use a linked-list structure:

```
typedef struct IntList {
 int value; /* Value for this node */
 struct IntList *next; /* Pointer to next node */
} IntList;
```

```
IntList * find_primes(int n);
```

## FINDING PRIMES (2)

- A simple algorithm to generate result:

```
for i = 2 to n do
 if is_prime(i) then
 Append new IntList node for i onto result.
done
return result
```

- Simple to implement, except possibly list part...

- To keep track of list pointers:

```
IntList *result = NULL;
IntList *last = NULL;
```

- Use **malloc()** to allocate new nodes for list

## FINDING PRIMES (3)

- To create and append new list nodes:

```
IntList *new;
...
if (is_prime(i)) {
 /* Allocate new node. */
 new = (IntList *) malloc(sizeof(IntList));
 new->value = i;
 new->next = NULL;

 /* Append new node onto existing list. */
 if (last != NULL)
 last->next = new;
 else
 result = new;

 /* Get ready to append next new node. */
 last = new;
}
```

# PRIME NUMBER LINKED-LIST

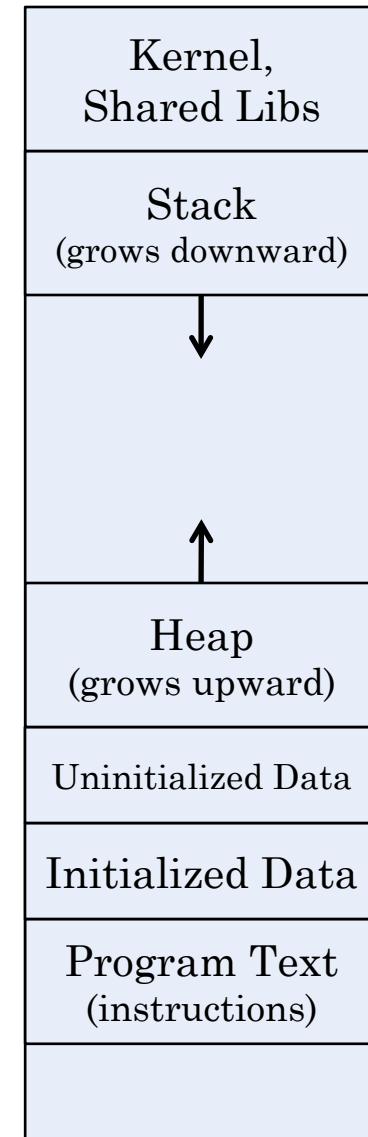
- There is no way to construct such a data structure on the stack!
  - Simply doesn't match the usage pattern for the stack
- Introduces another reason to use memory heap!
- Previous reasons:
  - Size of result depends on value of input(s)
  - Need result to live beyond lifetime of the procedure
- New reason:
  - Data structure simply cannot be represented on stack

# PROCESS MEMORY LAYOUT

- Where do the stack and heap actually live?
- Where does the program itself live?
- Every program is laid out in memory following a specific pattern
  - Memory regions devoted to stack, heap, instructions, constants, global variables, etc.
- Some of these regions vary in size from program to program
  - Instruction data depends on program size
  - Program may not have global variables
  - Compiler determines these details at compile-time
- Some regions depend on program behavior
  - e.g. heap usage or stack usage is a run-time behavior

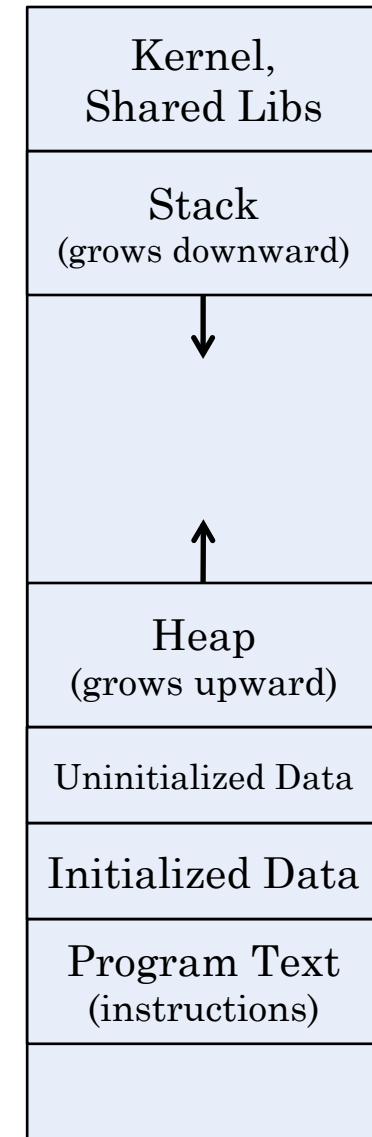
## PROCESS MEMORY LAYOUT (2)

- A common memory layout:
- Program text (instructions), globals, and constants are placed by compiler
  - Reside toward bottom of address space
- Stack resides near top of address space
  - Grows downward as space is consumed
- Heap resides above fixed-size program data
  - Grows upward as space is consumed
- Arrangement gives stack and heap maximum room to grow



# PROCESS MEMORY LAYOUT (3)

- Variable-size memory areas have soft and hard limits
  - When soft limit is hit, the region is extended, if possible
  - When hard limit is hit, the program is aborted by the operating system
- Book mentions **brk** value (§9.9)
  - “Break” address, where the program’s memory heap ends
  - **sbrk()** (“set break”) requests more heap space
- Also see **getrlimit()**/**setrlimit()** functions
  - Gets/sets limits on **brk**
  - Also limits on stack size, etc.



# STACK AND HEAP MANAGEMENT

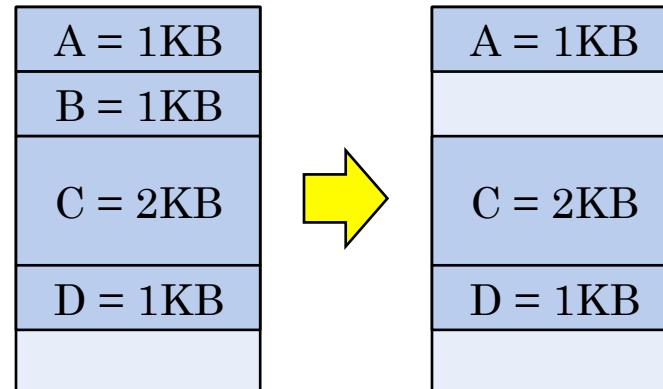
- Stack is relatively easy to manage
  - State consists of memory region + stack pointer
- Heap is *much* more complex!
  - Programs allocate and free blocks of memory, depending on their own needs
  - Heap must keep track of what memory is currently used, and what memory is available for use
  - Data structures for managing heap memory are necessarily complex
- Heap memory is managed by a heap allocator
  - Receives and attempts to satisfy allocation requests from a program
  - Manages these data structures internal to the heap

# HEAP ALLOCATORS

- Two kinds of heap allocators
  - Explicit allocators require applications to manually release memory when finished
    - e.g. C `malloc()`/`free()`, C++ `new/delete`
  - Implicit allocators detect when a memory region is no longer used
    - Employ garbage collectors for finding unused regions
    - e.g. Python, Java, Scheme, ...
- Today: begin looking at explicit heap allocators
- You will get to write one in HW3! ☺

# ALLOCATION AND FRAGMENTATION

- Allocators must handle any sequence of allocation requests
  - May affect ability to satisfy requests
- Example: heap with 6KB total space
  - A = allocate 1KB
  - B = allocate 1KB
  - C = allocate 2KB
  - D = allocate 1KB
  - Free B
- Try to allocate 2KB?
  - 2KB of memory is *available*, but it's not contiguous!
- Heap memory can become fragmented from use
  - Allocator must minimize occurrence of fragmentation...
  - ...but this also depends heavily on program's behavior!

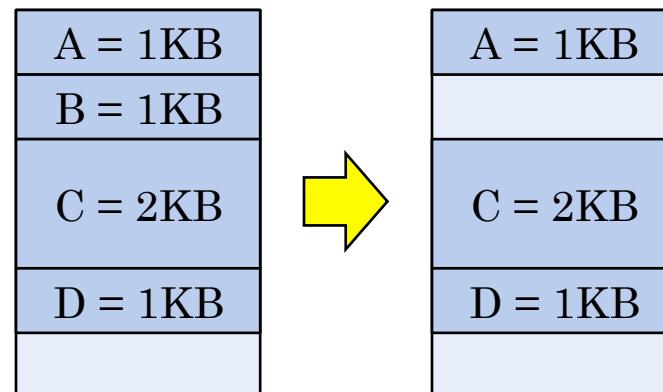


# ALLOCATION AND FRAGMENTATION (2)

- Allocators cannot modify a memory block once in use
  - e.g. to move it to another part of memory

- Our example:

- A = allocate 1KB
- B = allocate 1KB
- C = allocate 2KB
- D = allocate 1KB
- Free B
- Try to allocate 2KB?

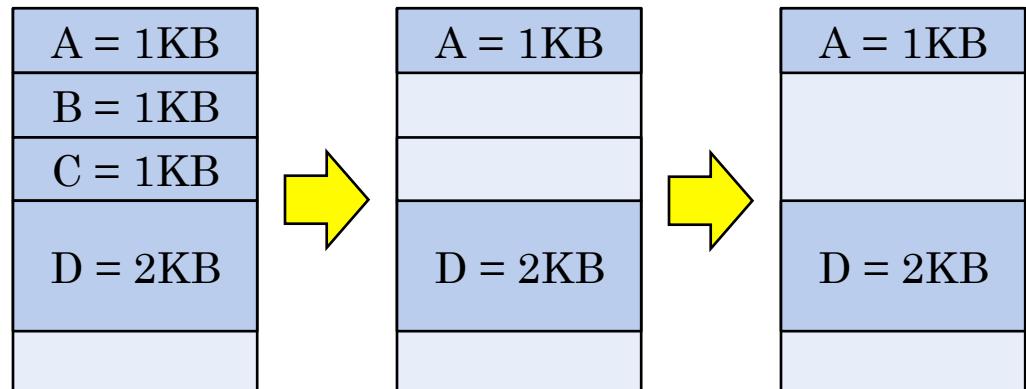


- Cannot modify allocated blocks to compact free space
  - Program has pointers into the allocated region
  - Allocator simply cannot find and update those pointers!
- Allocators can only manipulate free blocks
  - Program isn't using the free blocks, so it doesn't care...

# COALESCING FREE BLOCKS

- Another sequence of allocations:

- A = allocate 1KB
- B = allocate 1KB
- C = allocate 1KB
- D = allocate 2KB
- Free C
- Free B
- Try to allocate 2KB?

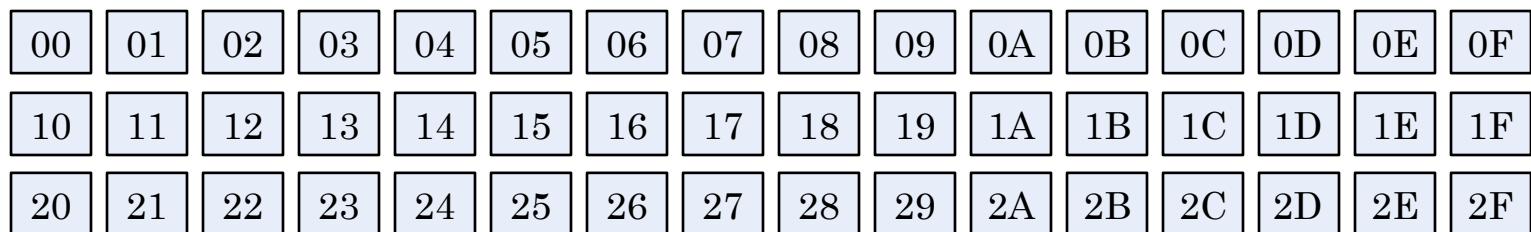


- Allocator needs to coalesce adjacent free blocks

- Failure to do so leads to false fragmentation
- Suitable free memory is available, but allocator can't find a region of the proper size to satisfy request
- Ideally, coalescing free blocks will be *fast*

# DATA ALIGNMENT AND ALLOCATORS

- Allocators often must care about data alignment
- So far, we have presented memory as an array of individually addressable bytes



- The processor may have a much larger data bus, e.g.  $w = 32$  bits
  - Processor can read or write 32 bits of data at once
- To take full advantage of this, memory must also be able to read or write the same size data value

## DATA ALIGNMENT AND ALLOCATORS (2)

- For e.g. 32-bit word size, memory looks like this:

|    |    |    |    |
|----|----|----|----|
| 00 | 01 | 02 | 03 |
| 04 | 05 | 06 | 07 |
| 08 | 09 | 0A | 0B |

- Each addressable cell actually holds 4 bytes
  - Not possible to address individual bytes in each cell; must read or write the entire cell
  - (Memories often allow you to specify a mask saying which bytes in the cell you actually want to use...)
- The processor is able to present an abstraction that individual bytes are addressable...
    - ...but, the CPU will still be interacting with memory in 32-bit words, not 8-bit bytes

# DATA ALIGNMENT AND ALLOCATORS (3)

- For e.g. 32-bit word size, memory looks like this:

|     |    |    |    |    |     |    |    |    |    |     |    |    |    |    |     |    |    |    |    |
|-----|----|----|----|----|-----|----|----|----|----|-----|----|----|----|----|-----|----|----|----|----|
| 00: | 00 | 01 | 02 | 03 | 01: | 04 | 05 | 06 | 07 | 02: | 08 | 09 | 0A | 0B | 03: | 0C | 0D | 0E | 0F |
| 04: | 10 | 11 | 12 | 13 | 05: | 14 | 15 | 16 | 17 | 06: | 18 | 19 | 1A | 1B | 07: | 1C | 1D | 1E | 1F |
| 08: | 20 | 21 | 22 | 23 | 09: | 24 | 25 | 26 | 27 | 0A: | 28 | 29 | 2A | 2B | 0B: | 2C | 2D | 2E | 2F |

- (Boxes contain byte-addresses; cell address is on left)
- CPU presents an abstraction that each byte is individually addressable...
- CPU maps the program's memory accesses to their actual addresses in physical memory
- Example: Write value 0xFF to byte-address 0x06
  - CPU writes 0x0000FF00 to memory address 1, and tells memory that only the 3<sup>rd</sup> byte should be written

# DATA ALIGNMENT AND ALLOCATORS (4)

- For e.g. 32-bit word size, memory looks like this:

|     |    |    |    |    |     |    |    |    |    |     |    |    |    |    |     |    |    |    |    |
|-----|----|----|----|----|-----|----|----|----|----|-----|----|----|----|----|-----|----|----|----|----|
| 00: | 00 | 01 | 02 | 03 | 01: | 04 | 05 | 06 | 07 | 02: | 08 | 09 | 0A | 0B | 03: | 0C | 0D | 0E | 0F |
| 04: | 10 | 11 | 12 | 13 | 05: | 14 | 15 | 16 | 17 | 06: | 18 | 19 | 1A | 1B | 07: | 1C | 1D | 1E | 1F |
| 08: | 20 | 21 | 22 | 23 | 09: | 24 | 25 | 26 | 27 | 0A: | 28 | 29 | 2A | 2B | 0B: | 2C | 2D | 2E | 2F |

- What happens when a program tries to access a word that spans multiple memory cells?
  - e.g. read word starting at byte-address 7
- The CPU must perform two reads, then assemble the result into a single word
  - Read memory cell at address 1, keep only the 4<sup>th</sup> byte
  - Read memory cell at address 2, keep bottom 3 bytes
  - Assemble these into a single word

# DATA ALIGNMENT AND ALLOCATORS (5)

- For e.g. 32-bit word size, memory looks like this:

|     |    |    |    |    |     |    |    |    |    |     |    |    |    |    |     |    |    |    |    |
|-----|----|----|----|----|-----|----|----|----|----|-----|----|----|----|----|-----|----|----|----|----|
| 00: | 00 | 01 | 02 | 03 | 01: | 04 | 05 | 06 | 07 | 02: | 08 | 09 | 0A | 0B | 03: | 0C | 0D | 0E | 0F |
| 04: | 10 | 11 | 12 | 13 | 05: | 14 | 15 | 16 | 17 | 06: | 18 | 19 | 1A | 1B | 07: | 1C | 1D | 1E | 1F |
| 08: | 20 | 21 | 22 | 23 | 09: | 24 | 25 | 26 | 27 | 0A: | 28 | 29 | 2A | 2B | 0B: | 2C | 2D | 2E | 2F |

- This obviously complicates the CPU logic...
- Some CPUs simply disallow non-word-aligned memory accesses
  - If a program tries to perform such an access, the CPU reports an error
  - (IA32 supports non-word-aligned memory accesses...)
- The compiler and the memory allocator must be aware of these data-alignment issues.*

# DATA ALIGNMENT AND ALLOCATORS (6)

- For e.g. 32-bit word size, memory looks like this:

|     |    |    |    |    |     |    |    |    |    |     |    |    |    |    |     |    |    |    |    |
|-----|----|----|----|----|-----|----|----|----|----|-----|----|----|----|----|-----|----|----|----|----|
| 00: | 00 | 01 | 02 | 03 | 01: | 04 | 05 | 06 | 07 | 02: | 08 | 09 | 0A | 0B | 03: | 0C | 0D | 0E | 0F |
| 04: | 10 | 11 | 12 | 13 | 05: | 14 | 15 | 16 | 17 | 06: | 18 | 19 | 1A | 1B | 07: | 1C | 1D | 1E | 1F |
| 08: | 20 | 21 | 22 | 23 | 09: | 24 | 25 | 26 | 27 | 0A: | 28 | 29 | 2A | 2B | 0B: | 2C | 2D | 2E | 2F |

- Solution: the compiler positions all values so that they always start at word boundaries
  - e.g. on a 32-bit system, only store values at addresses that are evenly divisible by 4 bytes
- Programmers generally don't have to think about these word-alignment issues...
  - ...but sometimes you can make your program *much* faster by knowing about them...

# DATA ALIGNMENT AND ALLOCATORS (7)

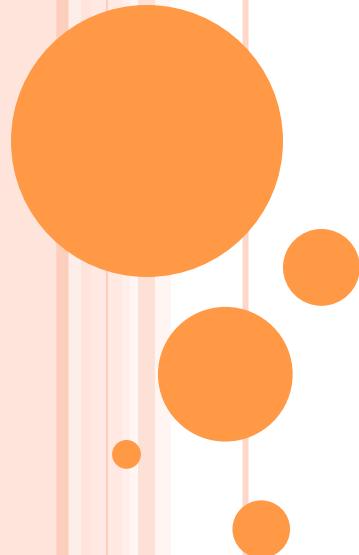
- For e.g. 32-bit word size, memory looks like this:

|     |    |    |    |    |     |    |    |    |    |     |    |    |    |    |     |    |    |    |    |
|-----|----|----|----|----|-----|----|----|----|----|-----|----|----|----|----|-----|----|----|----|----|
| 00: | 00 | 01 | 02 | 03 | 01: | 04 | 05 | 06 | 07 | 02: | 08 | 09 | 0A | 0B | 03: | 0C | 0D | 0E | 0F |
| 04: | 10 | 11 | 12 | 13 | 05: | 14 | 15 | 16 | 17 | 06: | 18 | 19 | 1A | 1B | 07: | 1C | 1D | 1E | 1F |
| 08: | 20 | 21 | 22 | 23 | 09: | 24 | 25 | 26 | 27 | 0A: | 28 | 29 | 2A | 2B | 0B: | 2C | 2D | 2E | 2F |

- Similarly, allocators generally only hand out chunks of memory that are word-aligned
  - Request size is also rounded up to next word-aligned boundary
- Example: an allocator that keeps regions aligned on 8-byte boundaries, and a request for 53 bytes
  - Allocator returns block with address 0x00417E58 (bottom 3 bits of the address are 0), size is 56 bytes

## NEXT TIME

- More about how explicit allocators work
  - How do allocators track which memory regions are free and which are used?
  - How do allocators deal with fragmentation?
  - What approaches and data structures are used?

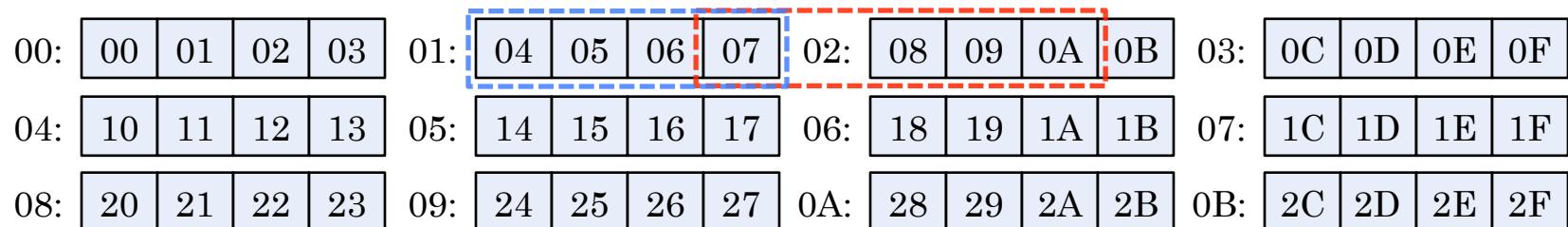


# CS24: INTRODUCTION TO COMPUTING SYSTEMS

Spring 2014  
Lecture 8

# LAST TIME

- Began examining explicit heap allocators
  - The program is responsible for releasing memory when it's no longer needed
- Allocator must deal with several challenges:
  - Avoiding or minimizing memory fragmentation
  - Coalescing adjacent blocks of free memory
  - Dealing with data alignment issues



# HEAP ALLOCATOR INTERFACE

- Common interface exposed by explicit allocators:

`void * malloc(size_t size)`

- Allocates a block of memory of [at least] `size` bytes
- Returns a pointer to start of the block, or `NULL` if `size` bytes are not available

`void free(void *ptr)`

- Releases a block of memory back to the heap

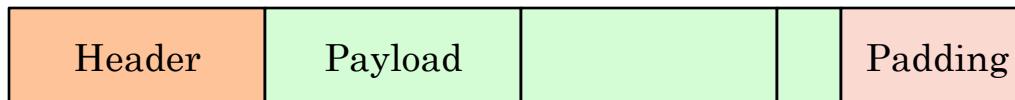
- Sometimes other operations as well:

`void * realloc(void *ptr, size_t size)`

- Attempts to change the size of an existing allocation
- If reallocation succeeds, original contents are copied to the new region, and original allocation is freed
- If reallocation fails, returns `NULL` and original allocation is left unchanged

# REPRESENTING MEMORY BLOCKS

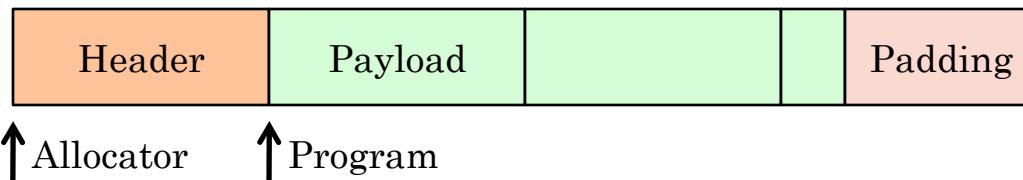
- Common way to represent blocks of memory on heap:



- Header specifies:
  - Total block size in bytes, including all parts
  - “Allocated” / “free” flag
  - Since memory block is usually word-aligned, the block-size value won’t actually use all bits
    - e.g. for aligning blocks on 8 bytes, bottom 3 bits of size will be 0
    - Can use bottom-most bit(s) to store allocated/free flag
- Payload: area that the program gets to use
  - Payload’s size is what the program requested
- Padding: any space necessary to make the block word-aligned (if necessary or desirable for platform)

# REPRESENTING MEMORY BLOCKS (2)

- Heap memory blocks:



- When a program requests memory:

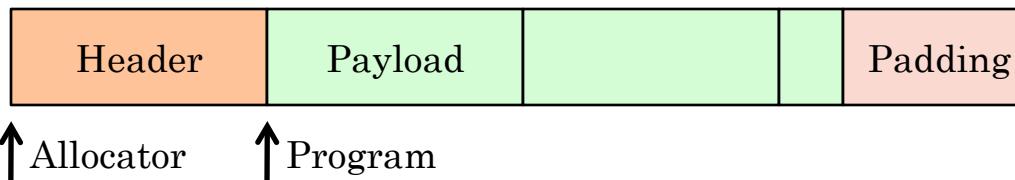
- Allocator works with block structure, creates/updates a block header to satisfy request
- Allocator returns a pointer to start of the *payload*, not start of the *header*

- Abstraction:

- Caller *doesn't care* how allocator manages the heap
- Just wants some memory to use for their program!

# REPRESENTING MEMORY BLOCKS (3)

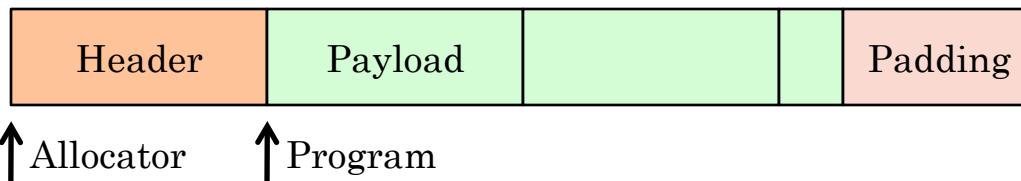
- Heap memory blocks:



- What can happen if program accidentally writes outside of the payload region?
  - e.g. due to a bug, the program writes past the end of its payload, or perhaps before the start
- In these cases, the heap can become corrupted
  - Can no longer keep track of heap's allocation state
- These bugs usually manifest at the *next* allocation or deallocation operation
  - This is when the allocator tries to use its state...

# REPRESENTING MEMORY BLOCKS (4)

- Heap memory blocks:



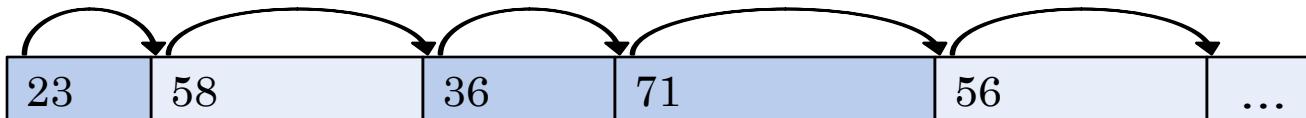
- When program frees memory:

- Program passes its pointer to start of the payload back to the allocator
- Allocator must adjust pointer to gain access to the header

# IMPLICIT FREE LIST

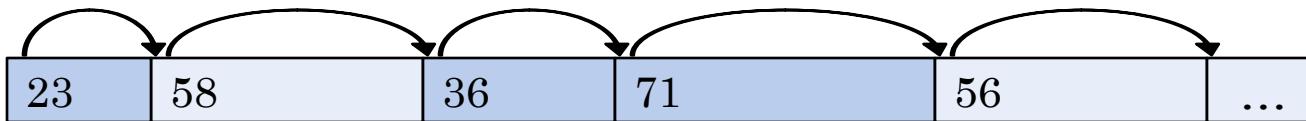
- Heap memory blocks form an implicit free list
  - Can determine start of next memory block by looking at size in header of current block
- When allocation request is made, allocator searches thru list of blocks to find a free block that can satisfy the request
- Several strategies for finding a suitable free block
  - First fit – start at beginning of list, stop when first suitable free block is found
  - Next fit – similar to first-fit, but remember where the last suitable free block was found, and start next search there
  - Best fit – check *all* free blocks; choose the smallest free block that can satisfy the request

 = used       = free

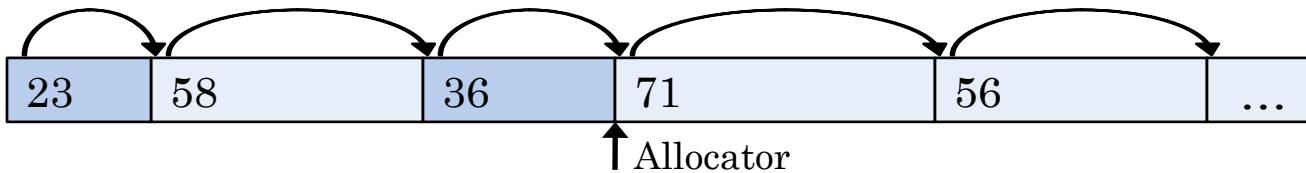


# COALESCING BLOCKS

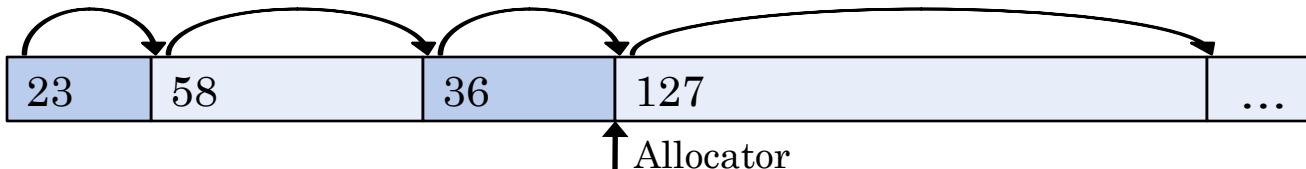
- When a block is freed, may need to coalesce it with adjacent free blocks



- Example: 4<sup>th</sup> block is freed by application

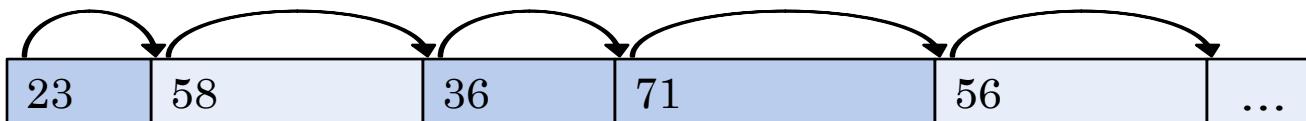


- Allocator checks if next block is also free
- If so, coalesce into a larger free block

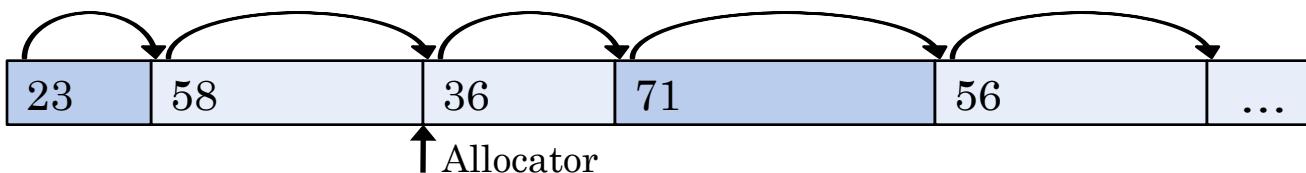


# COALESCING PREVIOUS BLOCKS

- This is only fast when coalescing with *next* block



- Example: 3<sup>rd</sup> block is freed by application



- How can we efficiently find the *previous* free block?!
- A simple solution: boundary tags (Knuth)
  - Give each block a footer as well as a header
    - (Footer is identical to header)
  - Can easily find prev. block size, and whether it's free

# EXPLICIT FREE LISTS

- Implicit free list approach is simple, but not fast
  - Allocation requires a scan of *all* blocks, whether allocated or free
  - No point in looking at allocated blocks!
- Two observations:
  - Really only need to keep the *free* blocks in a list, since these are the ones we check during allocation
  - Since free blocks aren't used by the program, could even store free-list pointers within the block payloads
- For example:
  - Use a linked list to chain together free blocks
- Called an explicit free list approach
  - Explicitly arranging free blocks into a data structure

# ORGANIZING FREE BLOCKS

- Many strategies for organizing/selecting free blocks
- Example 1: Maintain free-list in *LIFO order*
  - Newly freed blocks always go at front of free-list
  - Use first-fit policy for assigning new blocks
  - Even though free-list is in LIFO order, can use boundary tags on memory blocks to support constant-time coalescing with neighboring blocks
  - Very fast approach, but more susceptible to memory fragmentation
- Example 2: Maintain free-list in *address order*
  - Keep free blocks sorted by increasing address in free-list
  - Use first-fit policy for assigning new blocks
  - Slower due to linear-time insertion sort when freeing
  - Better memory utilization than LIFO-order free list

# ORGANIZING FREE BLOCKS (2)

- Other strategies maintain multiple free-lists
- Example: Segregated fits strategy
  - Each free-list is assigned a size class
    - e.g. {0-1024}, {1025-2048}, {2049-3072}, etc.
    - Each list contains free blocks of that size class
  - When allocating memory:
    - First-fit strategy, starting with appropriate size class
    - If no block is found, go to next larger size class
    - If still no block found, request more memory from OS
    - Once available block is found, may optionally split block and put free part into free-list for corresponding size class
  - When freeing memory:
    - Coalesce with adjacent free blocks, then put result into free list of appropriate size class
- This approach frequently used by standard allocators
  - Fast: allocation searches target blocks of appropriate size
  - Efficient: approaches memory usage of best-fit strategies

# DYNAMIC MEMORY ALLOCATION

- Like branching instructions, a heap facility greatly expands the programs we can write
- Heap management is a hard problem to solve!
  - Don't force every program to solve it separately...
  - Provide a run-time facility that implements this capability
  - Programs can leverage this facility as needed
- Understanding how heaps are implemented will help you use them more effectively. For example:
  - Code that performs allocations and deallocations of varying sizes can suffer from memory fragmentation issues
  - Code that always allocates/deallocates blocks of the same size won't suffer from fragmentation
  - Allocating many small objects will tend to waste lots of time and memory, due to bookkeeping overhead

## STACK VS. HEAP: COMPARISON

- Stacks are much faster and simpler than heaps
  - Bookkeeping complexity/overhead of heaps is why we don't allocate each local variable on the heap!
  - Much faster to keep local variables and other temporaries on stack
- Heaps provide different capabilities than stacks
  - Enables different usage patterns, but also has additional costs
- Important to understand these distinctions so you use the right tool for the job

# IMPORTANT OPTIMIZATION PRINCIPLES

- Two major optimization principles in systems:
- **Make the common case fast.**
  - Determine the most common behaviors of programs, and optimize hardware to execute these cases fast.
  - (Will explore this principle in 2<sup>nd</sup> half of the term!)
- **Make the fast case common.**
  - If hardware is good at certain cases, arrange your computations to make them as frequent as possible!

# DATA ALIGNMENT AND OPTIMIZATION

- See this very clearly with data alignment:
  - “Make the fast case common.”
  - Compiler and assembler adjust the layout of your code to align it with word boundaries for the CPU
- Accumulator code from lecture 5:

```
int value;

int accum(int n) {
 value += n;
 return value;
}

int reset() {
 int old = value;
 value = 0;
 return old;
}
...

.file "amain.c"
.text
.p2align 4,,15
.globl accum
.type accum, @function
accum:
 ... # code for accum()
 ret
 .size accum, .-accum
 .p2align 4,,15
.globl reset
.type reset, @function
reset:
 ... # code for reset()
```

- Same thing also happens with data structures

# HETEROGENEOUS DATA STRUCTURES IN C

- C represents heterogeneous data structures with **struct** declarations
  - **struct** members can be different data types, if desired
  - Each member has its own name
  - Members can be primitive types, pointers, arrays, other structs, etc.
- Example: a linked-list node

```
struct node {
 int number;
 char *string;
 struct node *next;
};
```

- Using our struct:

```
struct node n;
n.number = 42; /* Refer to members of struct. */
n.string = "answer";
n.next = NULL;
```

# C STRUCTURES IN IA32

- Compiler computes important details for each struct:

- Relative offset of each member from start of struct
- Size of each member's data type

- When C code accesses struct members:

- Compiler adds computed offsets to starting address of struct to access specific members

```
void init(struct node *n) { init:
 n->number = 42;
 n->string = "answer";
 n->next = NULL;
}
```

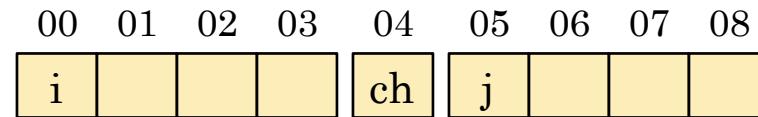
| Member       | Offset | Size    |
|--------------|--------|---------|
| int number   | 0      | 4 bytes |
| char *string | 4      | 4 bytes |
| node *next   | 8      | 4 bytes |

```
 pushl %ebp
 movl %esp, %ebp
 movl 8(%ebp), %eax # eax = n
 movl $42, (%eax) # n->number
 movl $.LC0, 4(%eax) # n->string
 movl $0, 8(%eax) # n->next
 popl %ebp
 ret
```

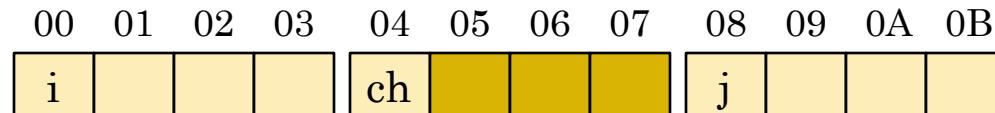
# STRUCTURES AND DATA ALIGNMENT

- Previous example had all 4-byte members
  - No data alignment issues
- Can use members that are less than a word

```
struct s1 {
 int i;
 char ch;
 int j;
};
```



- If compiler uses only one byte for **ch**, can't properly align **j** with word boundaries
- Compiler *pads* the struct to properly align all members

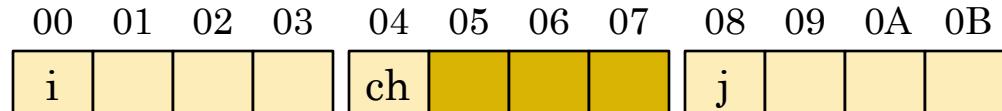


- **ch** is followed by 3 unused bytes, to properly align **j**

# STRUCTURES AND DATA ALIGNMENT (2)

- In this example:

```
struct s1 {
 int i;
 char ch;
 int j;
};
```



- Accesses to **ch** use byte-width operations

```
void init(struct s1 *rec) {
 rec->i = 1234;
 rec->ch = 'a';
 rec->j = 5678;
}
```

- **sizeof(s1)** reports 12 bytes, not 9 bytes

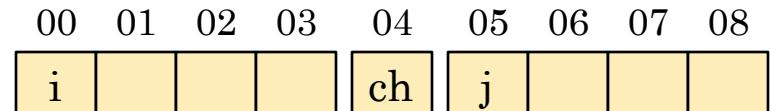
```
init:
 pushl %ebp
 movl %esp, %ebp
 movl 8(%ebp), %eax # eax = rec
 movl $1234, (%eax) # rec->i
 movb $97, 4(%eax) # rec->ch
 movl $5678, 8(%eax) # rec->j
 popl %ebp
 ret
```

# STRUCTURES AND DATA ALIGNMENT (3)

- Some compilers support packing these structures into minimal space necessary (non-standard!)

- e.g. **gcc** supports a **\_\_packed\_\_** attribute

```
struct s1 {
 int i;
 char ch;
 int j;
} __attribute__((__packed__));
```



```
void init(struct s1 *rec) {
 rec->i = 1234;
 rec->ch = 'a';
 rec->j = 5678;
}
```

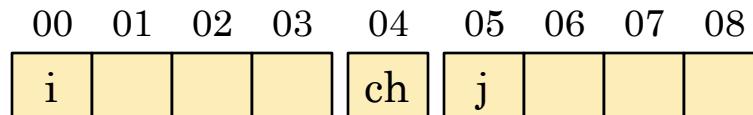
- `sizeof(s1)` also reports 9 bytes now

```
init:
 pushl %ebp
 movl %esp, %ebp
 movl 8(%ebp), %eax # eax = rec
 movl $1234, (%eax) # rec->i
 movb $97, 4(%eax) # rec->ch
 movl $5678, 5(%eax) # rec->j
 popl %ebp
 ret
```

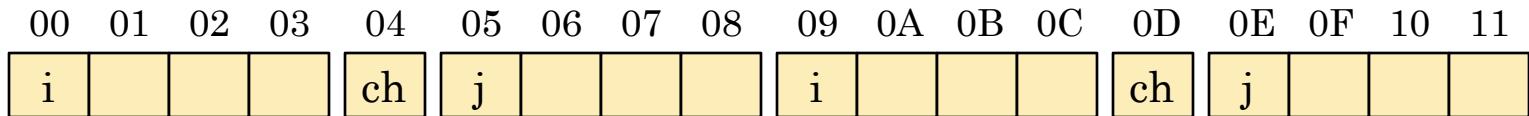
# STRUCTURES AND DATA ALIGNMENT (4)

- Packed version of structure will be significantly slower to work with in memory

- e.g. need multiple reads to access value of **j**



- If working with an array of **s1** values, *many* member accesses will not be word-aligned!

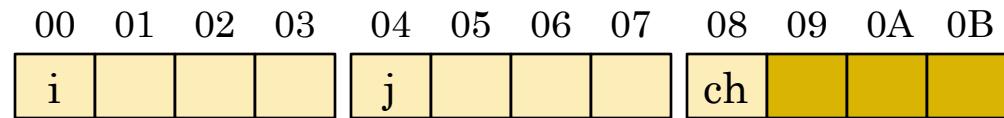


- Primarily useful for easily breaking apart packed values – e.g. network packets, disk/file structures
  - In memory, work with properly word-aligned struct
  - For IO, use packed version of struct to easily convert to/from a byte-sequence

# STRUCTURES AND DATA ALIGNMENT (5)

- Even if small member is at end of structure, still need to pad to word boundary

```
struct s2 {
 int i;
 int j;
 char ch;
};
```



- With local variables, don't want to affect other variables' word alignment

```
void foo(int i, int j) {
 struct s2; /* -12(%ebp) */
 int d; /* -16(%ebp) */
 ...
}
```

- If using in an array, want every element to be word-aligned

```
struct s2 values[100];
```

- Note:** 25% of space is wasted on padding to word boundaries!  
Important to consider in data structure design.

# OPTIMIZATION AND DATA ALIGNMENT

- Optimization: “Make the fast case common.”
  - Computers are fastest when accessing data aligned on word boundaries...
  - ...so the compiler lays out instructions and data to be aligned on word boundaries.
- Wastes a certain amount of space, but yields a substantial performance improvement

# C UNIONS

- C also provides unions:
  - Unlike structs, all union members occupy the same memory location

```
union value {
 int int_val;
 char *str_val;
 float float_val;
};
```
  - All members are assigned same offset by the compiler
  - Size of union is size of largest member
- Very useful for representing different, mutually-exclusive kinds of values in a single structure
- Also can be bug-prone, since C type-system can't keep you from misinterpreting a value!

| Member          | Offset | Size    |
|-----------------|--------|---------|
| int int_val     | 0      | 4 bytes |
| char *str_val   | 0      | 4 bytes |
| float float_val | 0      | 4 bytes |

## C UNIONS (2)

- Unions normally used in concert with a tag field
  - Unlike structs, union members all occupy the same memory location

```
struct value {
 enum ValType type;
 union {
 int int_val;
 char *str_val;
 float float_val;
 };
};
```

- Code can use **type** to determine what union-member to access

```
if (v->type == IntValue)
 set_result(a * v->int_val);
```

# C LANGUAGE RUN-TIME FACILITIES

- C is a relatively low-level language
  - Virtually all C abstractions translate easily to assembly language
    - Primitive data types, procedures, arrays and composite data types, flow-control statements
  - Not many run-time facilities for C programmers
- Example: array bounds checking
  - C does not stop you from indexing past an array's bounds!

```
int a;
int r[4];
int b;
...
r[4] = 12345; /* Compiles! */
r[-1] = 67890; /* Also compiles! */
```

- May affect **a** and/or **b**, depending on relative placement of the variables by the compiler

# C PROGRAMS AND ARRAY BOUNDS

- Lack of array bounds-checking can cause problems
- Buffer overflows:

- Program includes a **char** buffer for receiving input data

```
/* Buffer for reading in lines of input data. */
char buf[100];

...
gets(buf); /* Standard C function in stdio.h */
```

- An example implementation of **gets()**:

```
char * gets(char *s) {
 int i = 0;
 int ch = getchar(); /* Get char from console */
 while (ch != EOF && ch != '\n') {
 s[i] = ch; ch = getchar(); i++;
 }
 s[i] = 0; /* Zero-terminate the string. */
 return (ch == EOF && !feof(stdin)) ? NULL : s;
}
```

# C PROGRAMS AND ARRAY BOUNDS (2)

- **gets()** has no way of knowing how large its buffer is!

```
char * gets(char *s) {
 int i = 0;
 int ch = getchar();
 while (ch != EOF && ch != '\n') {
 s[i] = ch; ch = getchar(); i++;
 }
 s[i] = 0; /* Zero-terminate the string. */
 return (ch == EOF && !feof(stdin)) ? NULL : s;
}
```

- **gets()** is a common source of buffer overflows!

- **NEVER EVER** use **gets()** in your C programs!

- Much better to use:

```
char * fgets(char *s, int size, FILE *stream)
```

- Takes the buffer's size as an argument
- Function makes sure to stay within the buffer

# BUFFER OVERFLOW EXPLOITS

- Buffer overflows don't just cause your program to crash!
- They can frequently be leveraged to compromise system security
- Denial of service:
  - Cause the server program to crash unexpectedly when fed bad input
  - Example: Ping of Death
    - IP packets > 64KB in size are illegal
    - Many OSes had a packet buffer size of exactly 64KB
    - When sent a packet larger than 64KB, target machine would crash, hang, or otherwise freak out

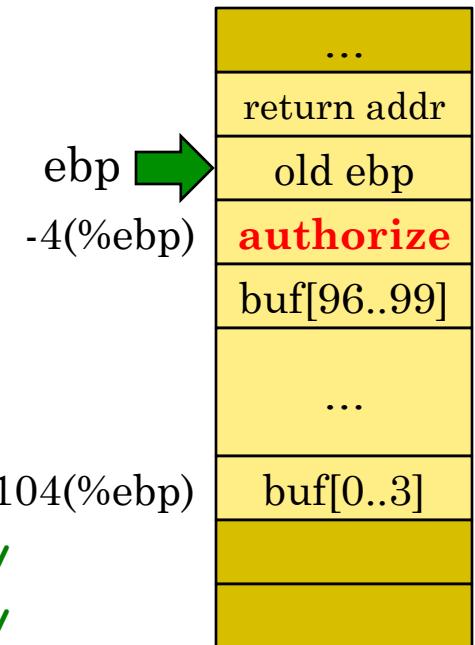
# BUFFER OVERFLOW EXPLOITS (2)

- Modify program state:

- Example server code:

```
void handle_request() {
 int authorize = 0;
 char buf[100];
 ... /* Read request into buffer */
 ... /* Verify user, etc. */

 if (authorize) {
 ... /* Request is allowed! Do it. */
 }
}
```



- By overflowing the input buffer, can modify value of other local variables.

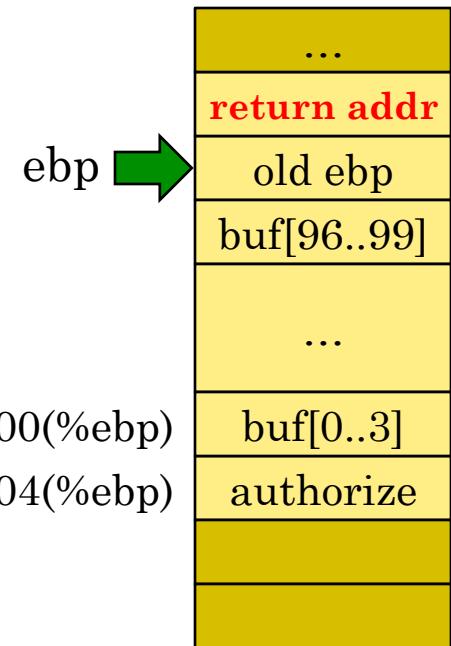
# BUFFER OVERFLOW EXPLOITS (3)

- Executing arbitrary code:

```
void handle_request() {
 char buf[100];
 int authorize = 0;

 ... /* Read request into buffer */ -100(%ebp)
 ... /* Verify user, etc. */ -104(%ebp)

 if (authorize) {
 ... /* Request is allowed! Do it. */
 }
}
```



- Instead of modifying state, exploit aims to change the actual *return address* on stack
  - Input includes malicious code loaded into the input buffer
  - Set return-address to jump into the buffer

# BUFFER OVERFLOW EXPLOITS (4)

- These examples store the buffer on the stack
- Heap-based buffer overflows are just as feasible
  - Overwrite jump-tables, function pointers, other code
- Details of these attacks depend very much on system details! For example:
  - Which direction the stack grows
  - Stack layout of function(s) being exploited
    - What local variables are present, and what effect they have
    - Where the return-address is stored
  - Heap layout of program being exploited
  - ...many other details of server being compromised...
- Nonetheless, a very large percentage of exploits use buffer overflows to compromise the system.

# AVOIDING BUFFER OVERFLOWS

- A simple solution to buffer overflows?
  - Build array bounds-checking into your language
  - All array indexes are verified before access occurs
  - Invalid indexes are flagged with some kind of error
- To support this, need to store more information about arrays
  - Add metadata to the array representation
- Example:

```
struct array_t {
 int length; /* Number of elements */
 struct value_t values[];
};
```

- Arrays include length information in their run-time representation

# ARRAY BOUNDS-CHECKING

- New array representation:

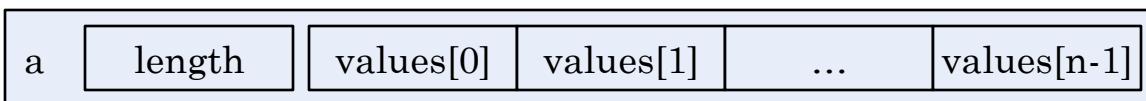
```
struct array_t {
 int length; /* Number of elements */
 struct value_t values[];
};
```

- Last member of a struct can be an array with no size
- Supports variable-size arrays in structs

```
array_t *a = (array_t *)
 malloc(sizeof(array_t) + n * sizeof(value_t));
a->length = n;
```

- **values** is a pointer to start of variable-size array

- Memory layout:



## ARRAY BOUNDS-CHECKING (2)

- Can expose array length as a member for programs to reference

```
for (int i = 0; i < a.length; i++) {
 compute(a[i]);
}
```

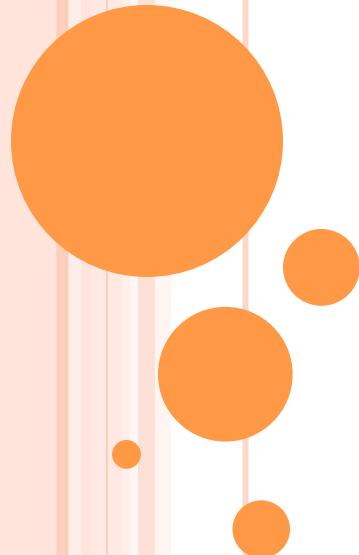
- In language implementation, all array-indexing operations are bounds-checked against length
- Still need to further constrain our language!
- Close off other potential holes:
  - No more pointer manipulation and pointer arithmetic
  - Present a simplified, opaque “reference” abstraction for programmers to use
  - Need a clean way to report errors when they occur

# MEMORY LEAKS

- Another common issue: memory leaks
- Explicit allocators require program to inform allocator when memory is no longer used
- If the program fails to do this properly:
  - Program no longer has a pointer to allocated memory
  - But, allocator thinks memory block is still in use!
  - Cannot reclaim memory until program terminates
- *Implicit allocators* assume the responsibility for reclaiming unused memory
  - Garbage collection: reclaiming unused heap storage
  - Program no longer has to free memory itself
  - Memory-leak issues largely disappear

# LOOKING FORWARD

- Languages like C don't make it very easy to write correct and bug-free programs!
  - Much easier than assembly language, but can easily have buffer overflows, memory leaks, exploits...
- Take another step up the abstraction hierarchy:
  - Programming languages that provide more powerful abstractions can mitigate many of these problems
- Will focus on three very common features:
  - Memory management: references, implicit allocators
  - Object-oriented programming and polymorphism
  - Another flow-control mechanism: exception handling



# CS24: INTRODUCTION TO COMPUTING SYSTEMS

Spring 2015  
Lecture 9

# LAST TIME

- Finished covering most of C's abstraction capabilities
  - Dynamic memory allocation on heap, structs, unions
- Began to see something very disturbing:
  - It's *very easy* to write incorrect or unsafe programs in C!
- Unchecked array accesses and buffer overflows:
  - Allow an attacker to crash a program, modify its data in unintended ways, or even execute arbitrary code!
- Other memory management problems as well:
  - Programs don't free memory when they are done with it
  - Programs allocate memory and then access beyond its end
  - Programs access memory after they have freed it
- Idea: *Most people don't actually need all this power!*
  - Provide a simplified memory management abstraction that makes it much easier to implement correct programs

# HIGHER-LEVEL LANGUAGE FACILITIES

- Starting to enter realm of higher-level languages
- Much safer programming models:
  - Easier to write correct programs
  - Fewer potential security holes!
- Much greater abstracting capabilities
  - Greater modularity, encapsulation, code reuse
- Also requires much larger run-time support for various facilities
  - Usually slower than C/C++ programs, but much safer!

# HIGHER-LEVEL LANGUAGE FEATURES (2)

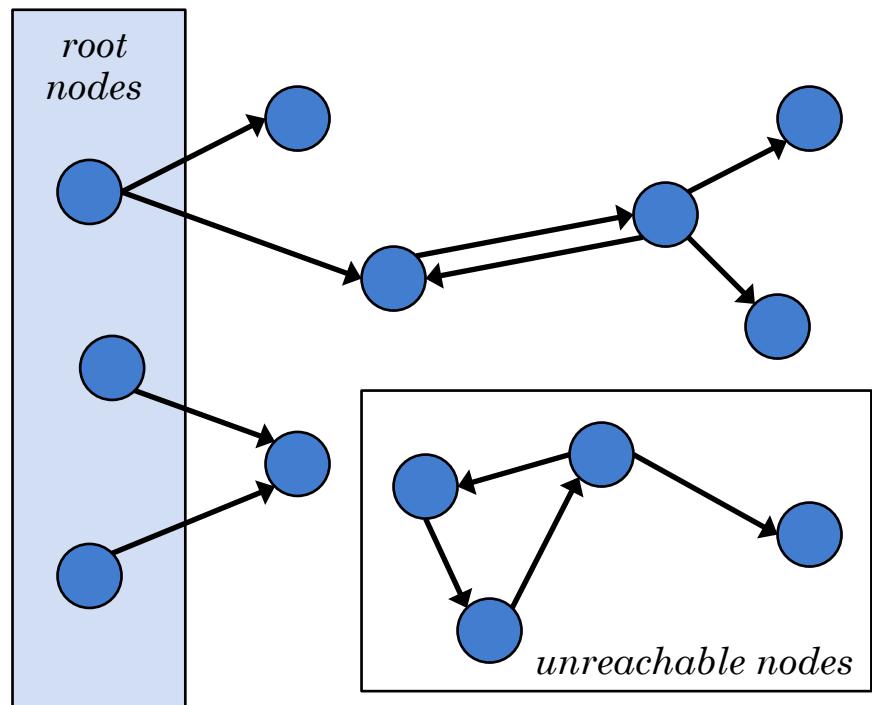
- In next few lectures, will explore three language features:
  - Implicit allocators and garbage collection
  - The object-oriented programming model
  - Exception handling
- Specifically, how to map these features to C, IA32
  - Taking another step up the abstraction hierarchy...
- Examples will draw from Java language features
  - A higher level language than C, with C-like syntax
  - (Not a *huge* step up the abstraction hierarchy...)
  - Includes all of the above language features
  - Itself implemented in C/C++ and assembly language

# IMPLICIT HEAP ALLOCATORS

- Explicit heap allocators rely on the program to release memory when no longer in use
  - ...*but programs are notoriously bad at this...*
- A first step towards better programs:
  - Implicit allocators assume the responsibility for identifying when a program is finished with memory
  - Employ a process called garbage collection to identify when a memory block is no longer used by a program
- Use of an implicit allocator eliminates *many* memory management issues for programs
  - Additional overhead for performing garbage collection
  - (A few other issues as well, all relatively minor)

# REACHABILITY GRAPH

- Garbage collectors are often built on the concept of a reachability graph
- Some allocated nodes are root nodes
  - Referenced from global environment, or stored on the stack
- All nodes reachable from the root nodes are live
- Unreachable nodes are garbage
  - May be reclaimed by allocator and reused for subsequent allocations
- How to determine this reachability graph?



# IMPLICIT ALLOCATORS AND POINTERS

- What about performing garbage collection in a language like C or C++?
- With such languages, garbage collection can be *very* difficult
- Unfortunately, C/C++ allows:
  - Pointers into the middle of memory blocks
  - Pointers into the middle of structs and arrays
  - Recasting pointers into pointers of other types
  - Recasting pointers into integers and vice-versa
  - All kinds of crazy pointer arithmetic!
- Makes it *very* difficult to build an accurate reachability graph in a C/C++ run-time system

# CONSERVATIVE GARBAGE COLLECTORS

- In languages like C and C++:
- Garbage collector attempts to identify any value that “looks like” a pointer
  - References a memory location that is currently valid
  - Assume this is a pointer to memory that is in use
- If the value isn’t actually a pointer?
  - Only real drawback is that the garbage-collector thinks memory is in use, when it really isn’t in use
  - (Hopefully) doesn’t happen often enough to be a problem
- Conservative garbage collection:
  - All reachable blocks are identified as reachable
  - Some unreachable blocks are also identified as reachable
  - **Not all garbage is reclaimed.** But we can live with that.

# PRECISE GARBAGE COLLECTORS

- Another approach is to *strictly control* how programs can use and manipulate pointers
  - Specify rules on casting pointers, and disallow casting to/from non-pointer types
  - Completely forbid pointer arithmetic!
- Allows precise garbage collection
  - Garbage collector can determine the reachability graph *exactly*, for all memory blocks in the heap
  - All garbage can be reclaimed by the allocator
- Given other issues with pointer manipulation (e.g. buffer overflows), *makes tons of sense* to constrain pointers this way!!!

# JAVA REFERENCES

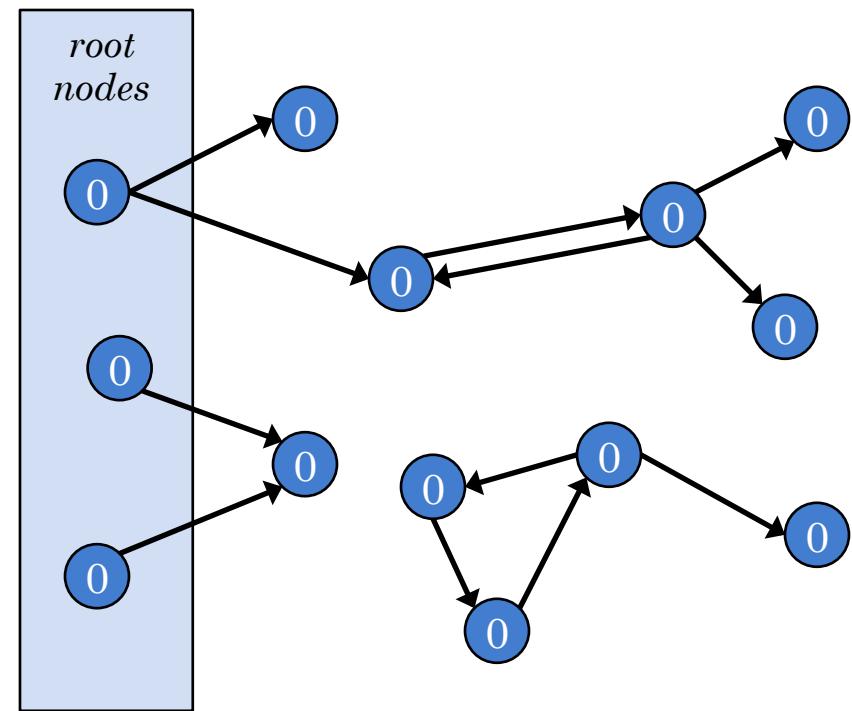
- Like many higher-level languages, Java includes references
- A “reference” is simply a means for looking up and accessing a particular object
  - A pointer is a very primitive kind of reference: it contains the exact memory address of the object
- Introduce a higher-level reference abstraction:
  - The reference is opaque to the program!
  - Programs can no longer directly access or manipulate the memory address associated with a reference
  - (The reference’s type can also only be manipulated in very specific, controlled ways.)
  - Gives the run-time system much greater flexibility in managing memory (e.g. moving allocated blocks)

# INDIRECTION

- **Indirection** is a very important technique used in computer system design
  - The ability to reference something using a name or other value that represents the *actual* value
- Have already seen this technique multiple times
  - e.g. “branch to register,” labels as jump-targets in code, using pointers to access values, jump tables, ...
- References allow programs to *indirectly* access objects, while the runtime directly accesses them
- “All problems in computer science can be solved by another level of indirection.”
  - David Wheeler, inventor of the subroutine

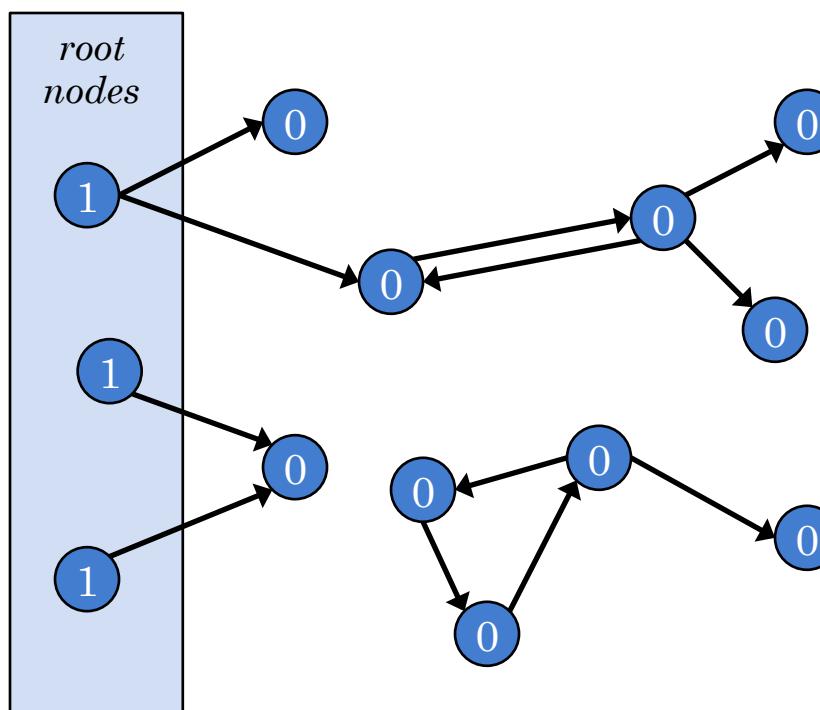
# GARBAGE COLLECTION ALGORITHMS

- Variety of algorithms used for garbage collection
- Simplest algorithm is called “mark and sweep”
- Every object has a flag associated with it
- Initially: all flags are 0
- Two phases:
  - First phase involves traversing entire object graph, marking all reachable objects
  - Second phase involves removing all unreachable objects



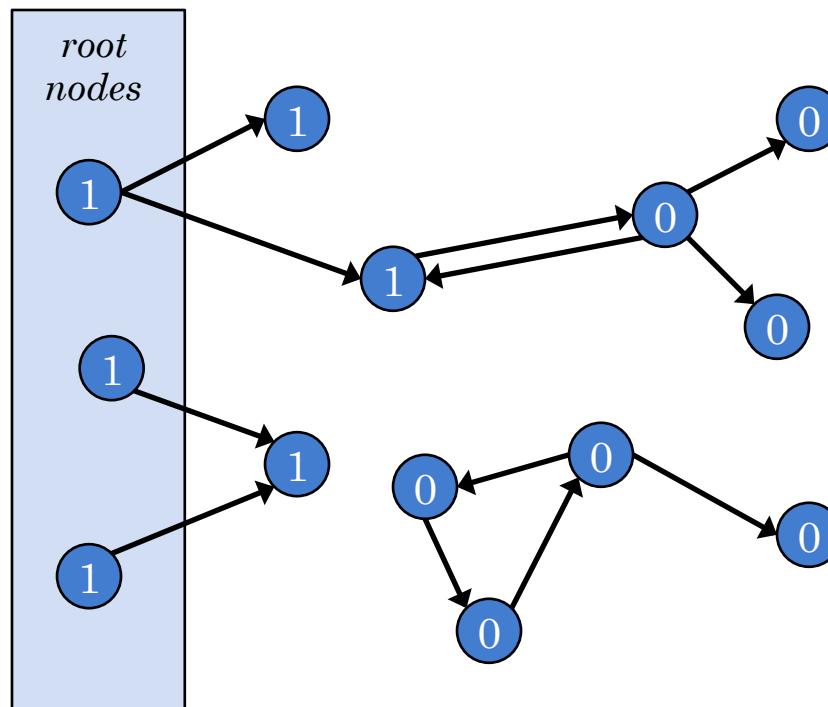
## MARK-AND-SWEEP (1)

- Start by marking root nodes as reachable
- (Note: would implement marking phase as depth-first traversal, not breadth-first...)



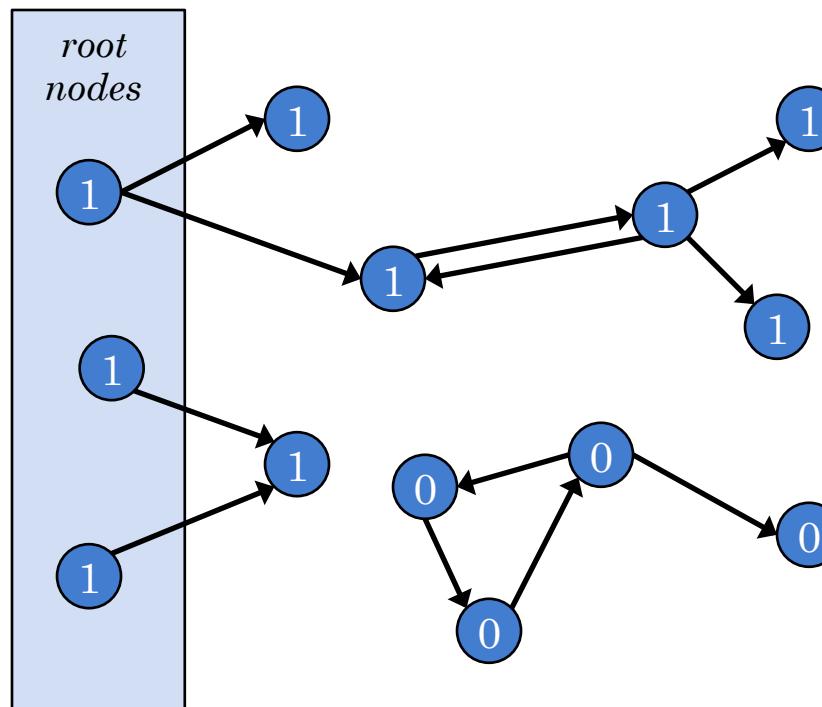
## MARK-AND-SWEEP (2)

- Next, nodes reachable from root nodes



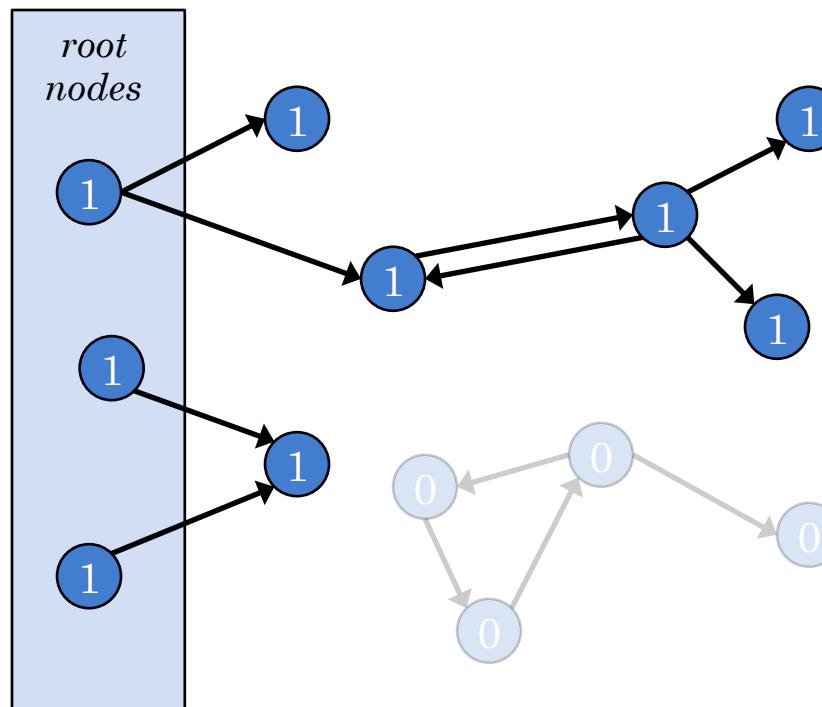
## MARK-AND-SWEEP (3)

- Continue until all reachable nodes are marked.
- Now, any node with a 0 is unreachable, and may be reclaimed.



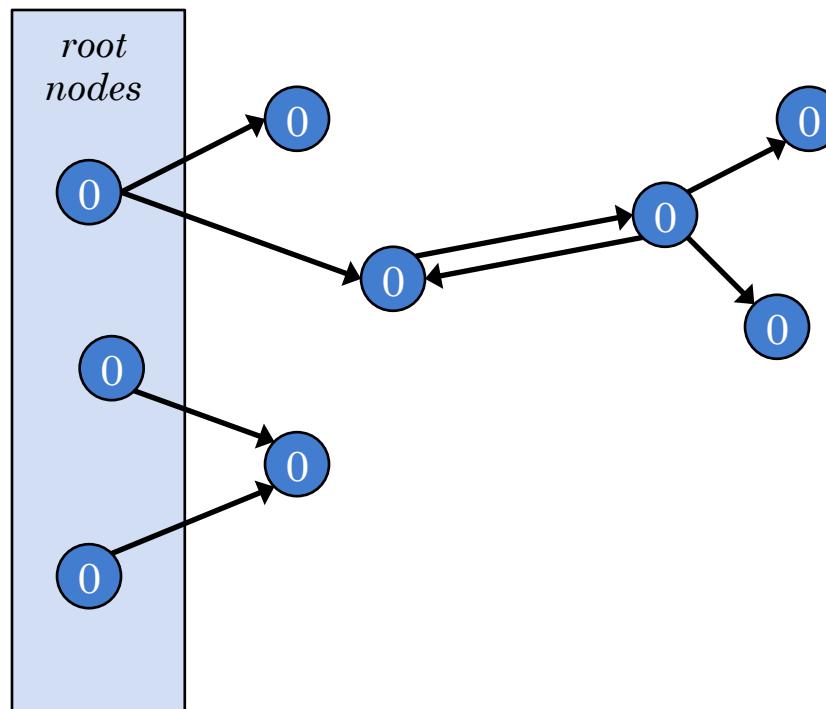
## MARK-AND-SWEEP (4)

- Second phase:
  - If object is unreachable, reclaim it.
  - (If object is reachable, reset flag to 0 for next time.)



## MARK-AND-SWEEP (5)

- Final result:



# MARK-AND-SWEEP CHARACTERISTICS

- This GC algorithm has several drawbacks
- Most important one:
  - To ensure that all garbage is identified, the program cannot run while the garbage collector is working!
  - Even on a multicore system, program and GC cannot run concurrently
- This is called a “stop-the-world” garbage collector
  - Entire program must be suspended while garbage collection is performed
- Clearly unacceptable for applications where response-time is critical
  - e.g. real-time applications, interactive applications

# GARBAGE COLLECTOR CHARACTERISTICS

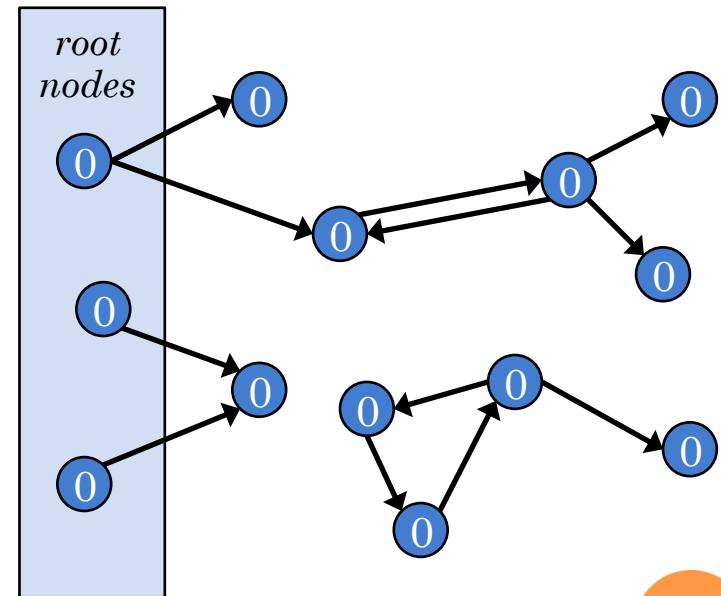
- Several kinds of garbage collector characteristics
- Stop-the-world GC vs. concurrent GC
  - Stop-the-world garbage collectors must suspend the program while performing garbage collection
  - Concurrent garbage collectors work in the background, while the program continues to run
    - Algorithm may also require “stop-the-world” phases, but they are kept as short as possible
    - Frequently, the “stop-the-world” phases are parallelized to minimize wait-time
- Serial GC vs. Parallel GC
  - Serial garbage collectors are not able to use multiple processors during GC phases
  - Parallel collectors are able to employ multiple processors to speed up collection phases

## GC CHARACTERISTICS (2)

- Compacting GC vs. non-compacting GC
  - Non-compacting collectors do nothing with the remaining live objects
    - This is the only real option for languages that allow programs to use explicit pointers into memory
    - Memory fragmentation can become a big problem!
  - Compacting garbage collectors move remaining live objects together, maximizing size of free space
  - Only possible to compact memory if language doesn't expose explicit pointers to programs!
    - Another reason to only expose opaque references

## MARK-AND-SWEEP CHARACTERISTICS (2)

- In what situations is mark-and-sweep garbage collection the least expensive?
  - (Compacting mark-and-sweep, in particular?)
- If most objects are not reclaimed, mark-and-sweep will be relatively inexpensive
  - Mark phase always touches *all* objects, regardless...
  - Sweep phase will have to do less work if most objects don't need reclaimed
  - Particularly true if GC must also compact memory!



# OTHER GC STRATEGIES

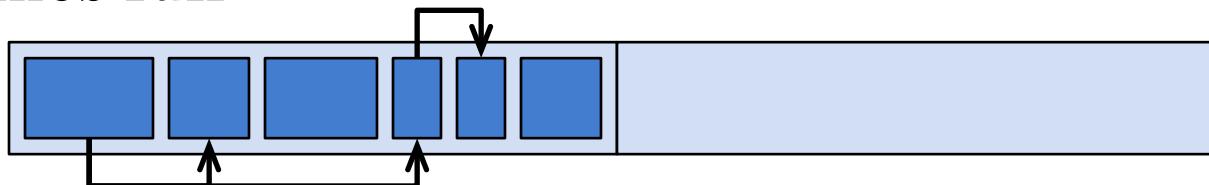
- Mark-and-sweep garbage collection must do a lot of work if many objects are deleted during sweep:
  - Must deallocate each garbage object individually (this overhead definitely adds up)
  - Must compact live objects to avoid fragmentation
- Mark-and-sweep prefers long-living objects ☺
- Another strategy: Copying garbage collectors
  - Instead of compacting live objects together, live objects are all copied (“evacuated”) to another, contiguous region of memory
  - Compaction is performed automatically!
  - All dead objects can be deallocated in one operation!

# STOP-AND-COPY GARBAGE COLLECTORS

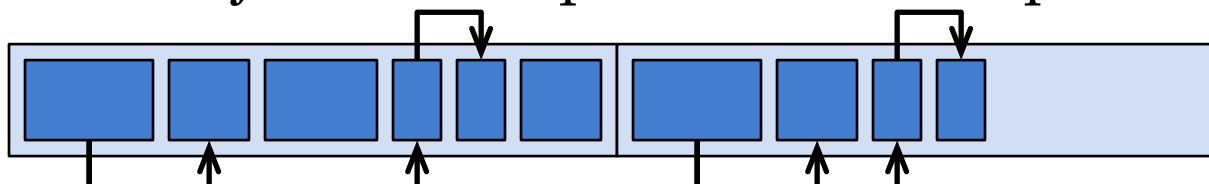
- Stop-and-copy garbage collectors divide memory into two regions: “from-space” and “to-space”
- New objects are allocated in the to-space
- When to-space becomes full, it becomes the “from-space,” and vice versa:
  - Starting with root objects, all reachable objects are copied from from-space into to-space
  - At end, entire from-space can be reclaimed at once
- Program resumes execution, using new to-space
- Automatically compacts live objects, but also effectively halves memory available to programs

# STOP-AND-COPY EXAMPLE

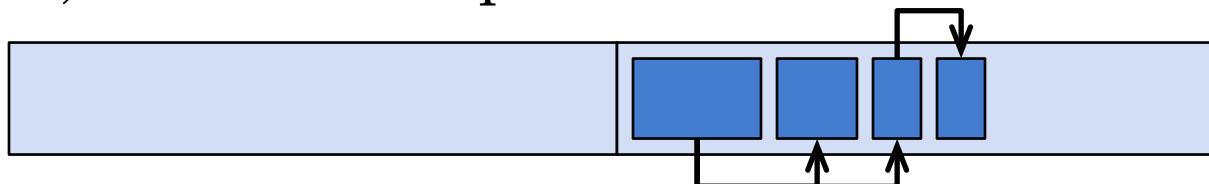
- Memory divided into from-space and to-space
- New objects are allocated in the to-space, until it becomes full



- Program is stopped; starting with root objects, reachable objects are copied to new to-space

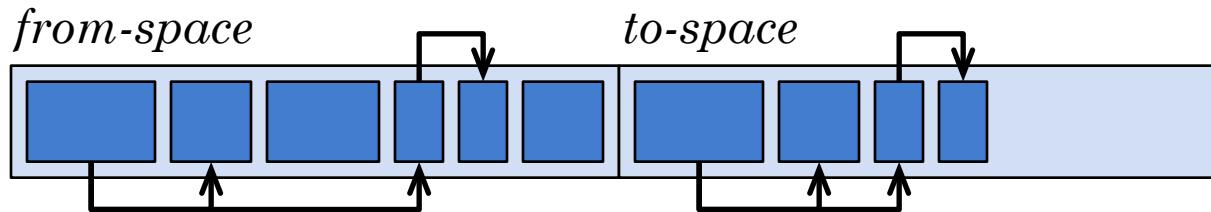


- At end, entire from-space is reclaimed



# STOP-AND-COPY GC

- Stop-and-copy garbage collection is fast when:
  - Not a lot of objects live through the GC phase!



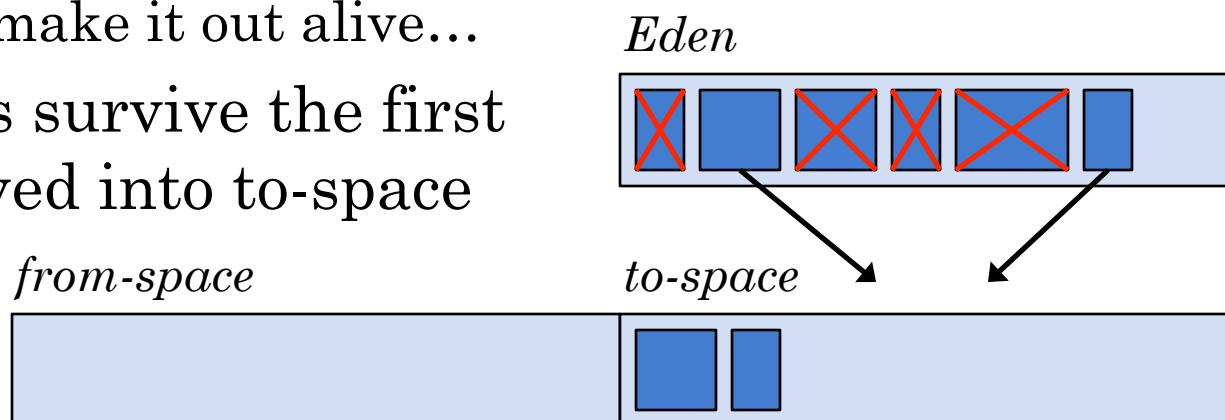
- The fewer objects you have to copy, the better.
- Stop-and-copy prefers short-lived objects ☺
- Observation: Different garbage collection algorithms are best for different situations...
- Questions:
  - Do all heap-allocated objects behave “the same” with respect to garbage collection?
  - Are there differences that we can take advantage of?

# GENERATIONAL GARBAGE COLLECTION

- Some empirical observations about programs:
- Most objects in a program are very short-lived
  - e.g. used for local variables, intermediate results, etc.
  - “Most objects die young.”
- Longer-lived objects generally do not reference shorter-lived ones
- These ideas called the *generational hypothesis*
  - (Also “infant mortality,” but that’s just macabre.)
- Idea:
  - Design a garbage collector that takes advantage of this behavioral characteristic of OO program state
  - Called generational garbage collection
  - Optimization: “Make the common case fast.”

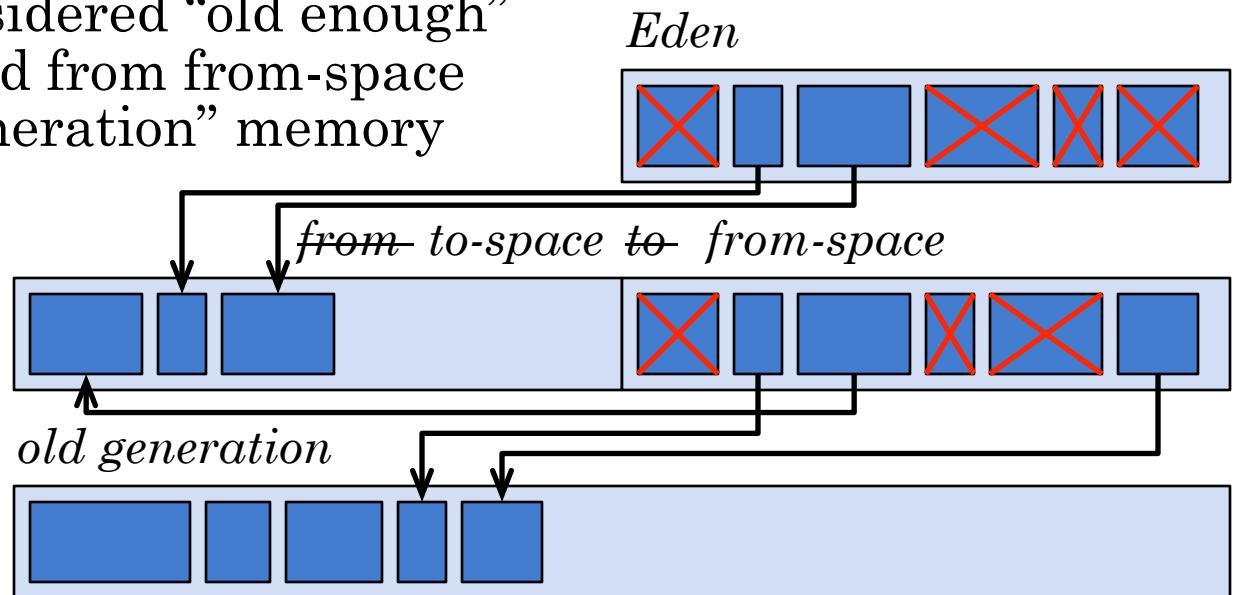
# SUN HOT-SPOT GENERATIONAL GC

- Sun Java VMs implement generational garbage collection
- Divides objects into “young objects,” “old objects”
- Young objects kept in a separate memory area
  - Uses stop-and-copy GC, since most will not live long
  - Three memory areas: Eden, and two survivor spaces
- Newest objects are allocated in Eden
  - Most never make it out alive...
- If new objects survive the first GC pass, moved into to-space



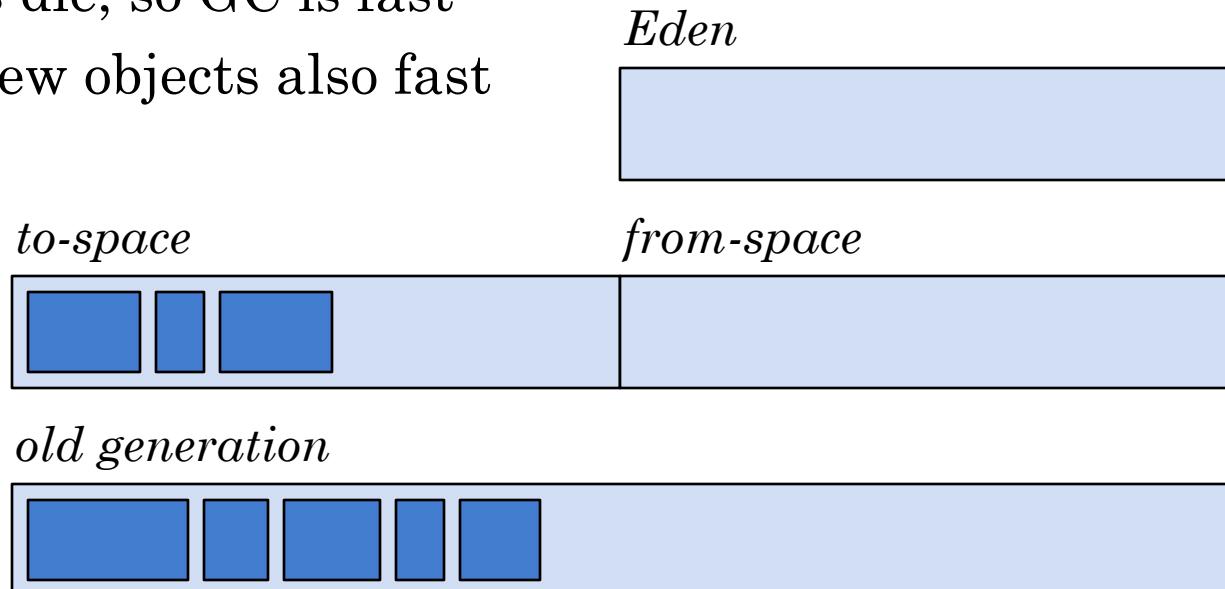
## SUN HOT-SPOT GENERATIONAL GC (2)

- Survivor spaces for slightly older “young objects”
  - Give young objects “additional chances to die” before they are considered “old objects”
- As before, when to-space fills up, turn into from-space, and perform stop-and-copy GC
- An important difference:
  - Objects considered “old enough” are promoted from from-space into “old generation” memory



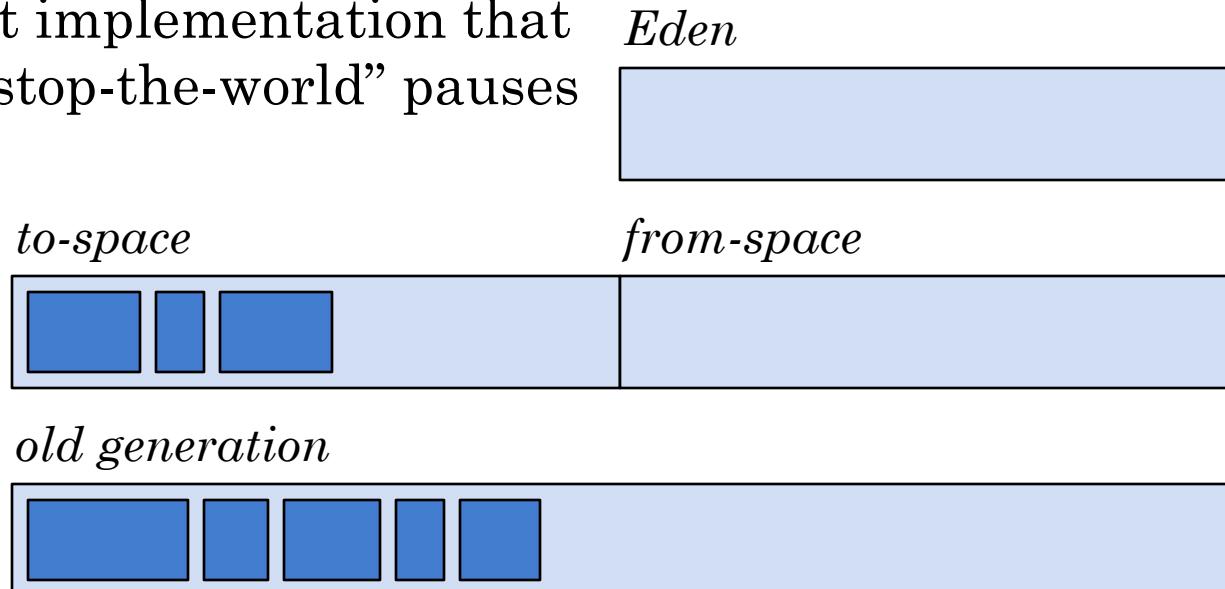
# SUN HOT-SPOT GENERATIONAL GC (3)

- After young-object garbage collection, both Eden and the from-space are now empty
  - Objects have either been reclaimed, or copied to another memory area
- Two benefits of stop-and-copy GC here:
  - Most objects die, so GC is fast
  - Allocating new objects also fast



## SUN HOT-SPOT GENERATIONAL GC (4)

- Finally, old generation also needs periodic GC
- Most of these objects are expected to survive...
  - Stop-and-copy garbage collection is not appropriate!
- Old generation is managed with a compacting mark-and-sweep algorithm
  - A concurrent implementation that minimizes “stop-the-world” pauses



# GENERATIONAL GARBAGE COLLECTION

- Generational garbage collection is very complex!
- Takes advantage of two important details:
  - Different garbage collection algorithms are good in different situations
  - Program state tends to fall into two major categories: young, short-lived objects, and old, long-lived objects
- Provides a much more effective garbage collection system than the individual GC algorithms could possibly provide on their own!

# GENERAL-PURPOSE SOLUTIONS

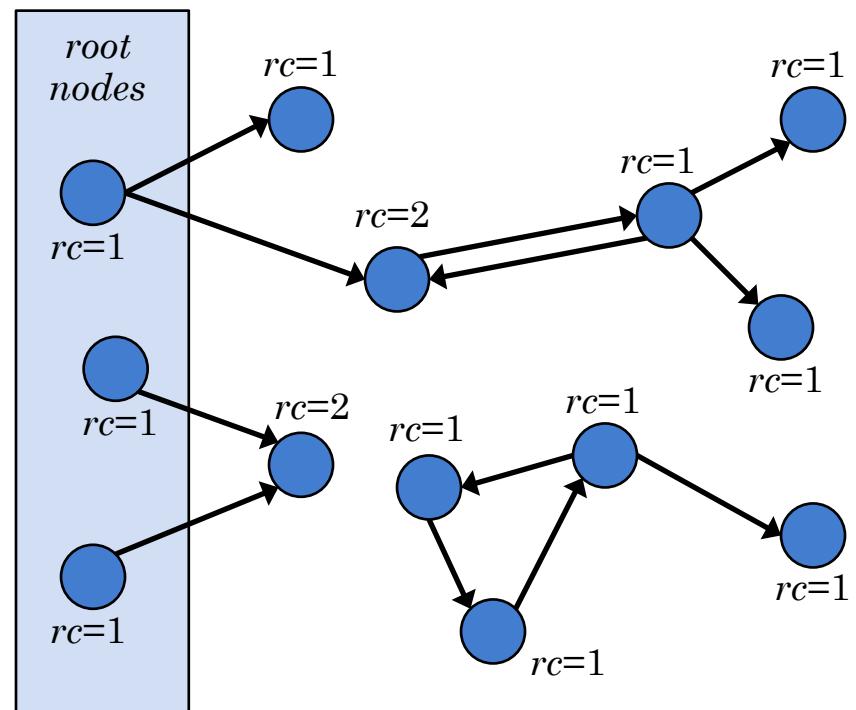
- Another very important system-design pattern: creating general solutions from specialized ones
- Can frequently solve a problem in multiple ways
  - e.g. mark-and-sweep GC vs. stop-and-copy GC
  - Each solution works well in different situations
- We want our computers to be general-purpose...
- We want our operating systems to support a wide range of program behaviors and usage scenarios
- The most powerful, generic solutions frequently blend multiple techniques in a very elegant way
  - Generational garbage collection is a great example!
  - This is a common theme in computer system design

# REFERENCE COUNTING

- Some implicit allocators are based on reference counting, instead of on a reachability graph
- Each object keeps a reference count
  - A simple integer count of how many other objects reference the object
- When code first references an object, its reference-count is automatically incremented
- When code finishes working with the object, its reference count is automatically decremented
- When an object's reference count hits zero, it is automatically reclaimed
  - Instead of periodic, complicated garbage-collection sweeps, objects are immediately reclaimed

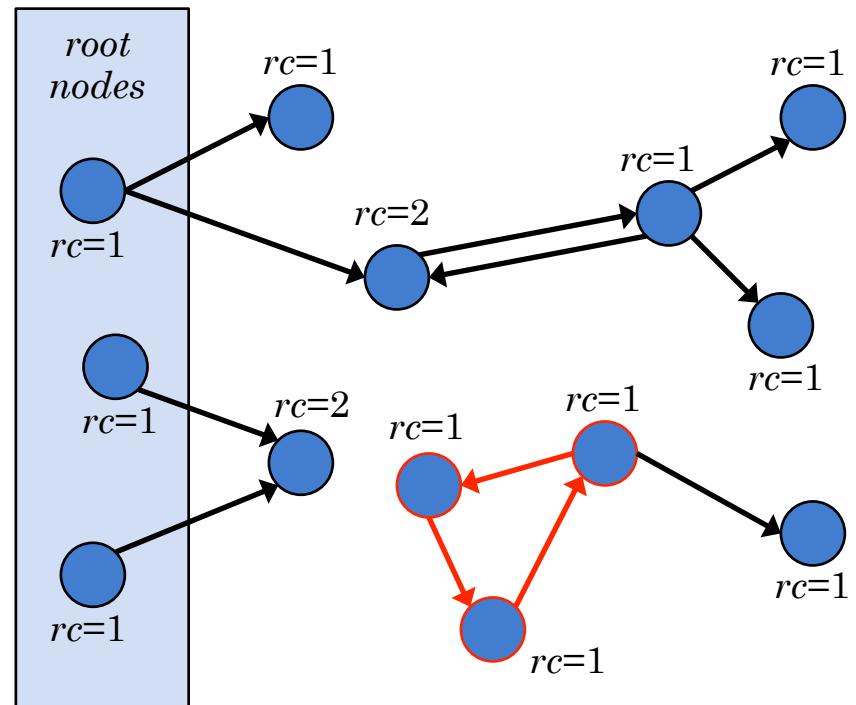
# REFERENCE COUNTING AND CYCLES

- Major drawback of reference counting:
  - Cannot properly release cycles of objects!
- Our earlier example:
  - All objects have a nonzero reference-count...
  - ...but some of the objects are unreachable!
- Despite this limitation, many systems still use reference-counting for automatically freeing objects
  - e.g. the Python runtime



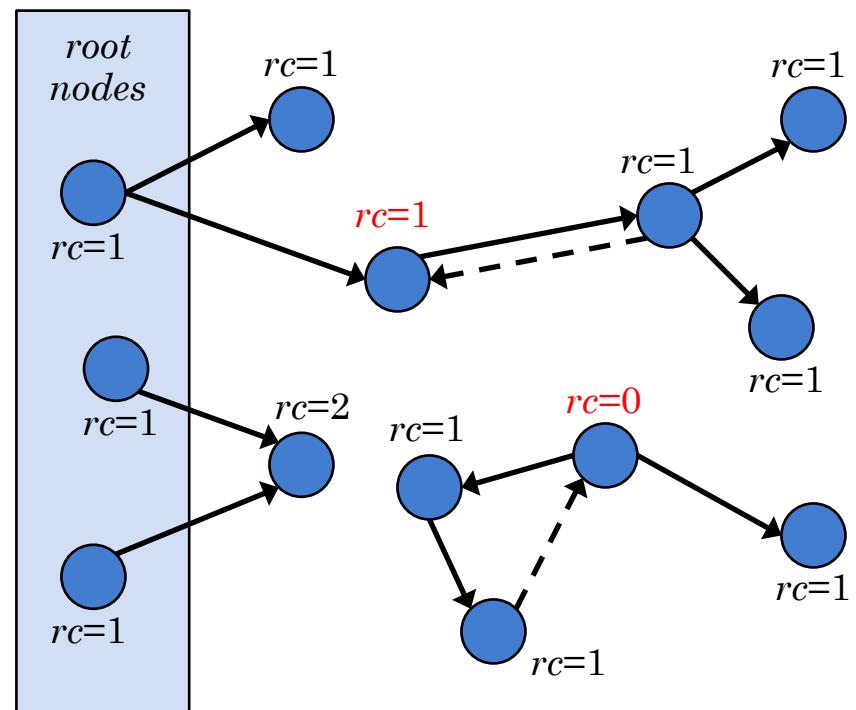
# REFERENCE COUNTING AND CYCLES (2)

- Several techniques for dealing with cycles
- Cycle detection algorithms
  - Use more standard reachability-graph techniques to detect and reclaim cycles
  - Since it's only needed for a subset of objects, won't have a heavy impact
  - Approaches usually focus on identifying objects that *could be* part of a cycle, and starting from there
- More complex allocator, but keeps the coder's life simpler!



# REFERENCE COUNTING AND CYCLES (3)

- Another approach: weak references
  - Simply don't allow cycles in reference graph
  - When objects need to refer to each other, one object uses a weak reference
    - Doesn't increment target object's reference count
    - Target of a weak reference may go away unexpectedly
- Properly breaks cycles...
- Programmer must design the program to use weak references properly
  - Can be *very* difficult to get this correct!

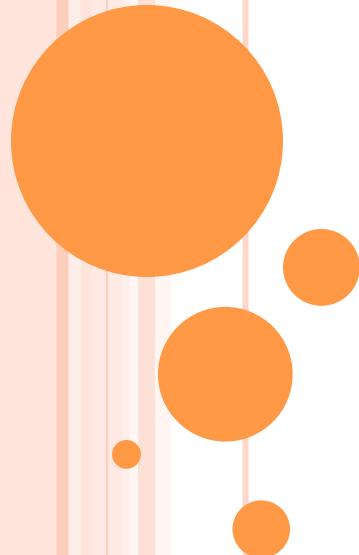


# REF-COUNTING, GARBAGE COLLECTION

- Other drawbacks with reference counting
  - Cost of incrementing and decrementing reference-counts really adds up
  - Languages with garbage collection don't incur this overhead
  - Some ways to optimize away reference-count updates
- Nonetheless, still quite a common approach
  - Easy to implement!

# SUMMARY

- Implicit allocators allow us to eliminate *many* memory management issues
  - Provide a simplified abstraction to programs
- Programs allocate memory, but implicit allocator uses garbage collection to determine when to free
  - e.g. mark and sweep, stop and copy
  - More advanced allocators blend these techniques to create very powerful, general-purpose approaches, e.g. generational garbage collection
- Many languages replace pointers with references
  - Program indirectly references objects, while runtime can still directly access and manipulate them
  - Eliminates many other kinds of serious security bugs!



# CS24: INTRODUCTION TO COMPUTING SYSTEMS

Spring 2015  
Lecture 10

## PREVIOUSLY: ARRAY ACCESS

- C doesn't provide a very safe programming environment
- Previous example: array bounds checking

```
int a;
int r[4];
int b;

...
r[4] = 12345; /* Compiles! */
r[-1] = 67890; /* Also compiles! */
```

- Depending on variable placement, could affect:
  - **a** and/or **b**
  - Caller's **ebp**, return address on stack, etc.
- Or, perhaps nothing at all!

# CHECKED ARRAY INDEXING

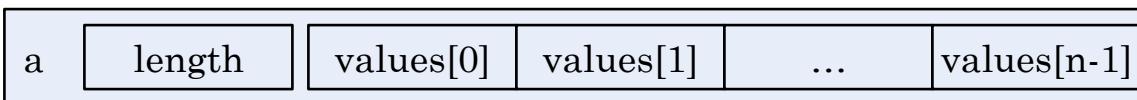
- Could add metadata to arrays

```
struct array_t {
 int length; /* Number of elements */
 struct value_t values[];
};
```

- Arrays include length information in their run-time representation
- Last member of a struct can be an array with no size
- To initialize a new array:

```
array_t *a = (array_t *)
 malloc(sizeof(array_t) + n*sizeof(value_t));
a->length = n;
```

- **values** is a pointer to start of variable-size array



# ARRAY BOUNDS-CHECKING (2)

- Arrays are now a more intelligent data type:

```
for (int i = 0; i < a.length; i++) {
 compute(a[i]);
}
```

- A composite type containing multiple related values
- Ideally, **length** would be read-only, and every indexing operation would be verified against **length**
- If only our type could also expose specific behaviors...
  - Operations that can be performed on these values
  - e.g. expose length via a function, or check indexes in an access function

# OBJECT ORIENTED PROGRAMMING

- Idea:
  - Group together related data values into a single unit
  - Provide functions that operate on these data values
  - This state and its associated behavior is an object
- A class is a definition or blueprint of what appears within objects of that type
- Encapsulation:
  - Disallow direct access to the state values of an object
  - Provide accessors and mutators that control *when* and *how* state is modified
- Abstraction:
  - Class provides simplified representation of what it models
  - Compose simpler objects together to represent assemblies
    - e.g. a Car has an Engine, a Transmission, Pedals, a SteeringWheel, Instruments, etc.

# OBJECT ORIENTED PROGRAMMING (2)

- Idea:
  - Object oriented programming paradigm makes it easier to create large software systems
  - Promotes modularity and encapsulation of state
  - Provides sophisticated modeling and abstraction capabilities for programs to use
- (Not everyone believes that OOP is best way to provide these features...)
- Many different object-oriented languages now!
  - C++, Java, C#, Scala, Python, Ruby, JavaScript, Perl, PHP, ...
- Today: focus on some OO features found in Java

# OBJECT ORIENTED PROGRAMMING: JAVA

- Java presents a specific object-oriented programming model
- Includes some kinds of variables we recognize:
  - Global variables
  - Function arguments
  - Local variables
- Object-oriented model also introduces:
  - Class variables
  - Instance variables

```
public class RGBColor {
 public static RGBColor RED =
 new RGBColor(1.0, 0.0, 0.0);

 private float red, green, blue;

 public RGBColor(...) { ... }

 public void setRed(float v) {
 red = v;
 }
 ...

 public void fromHSV(float h,
 float s,
 float v) {
 float p = v * (1.0 - s);
 ...
 }
}
```

# JAVA TYPES AND ABSTRACTIONS (2)

- How do environments fit together to provide these kinds of state?
  - Global variables
  - Function arguments
  - Local variables
  - Class variables
  - Instance variables

```
public class RGBColor {
 public static RGBColor RED =
 new RGBColor(1.0, 0.0, 0.0);

 private float red, green, blue;

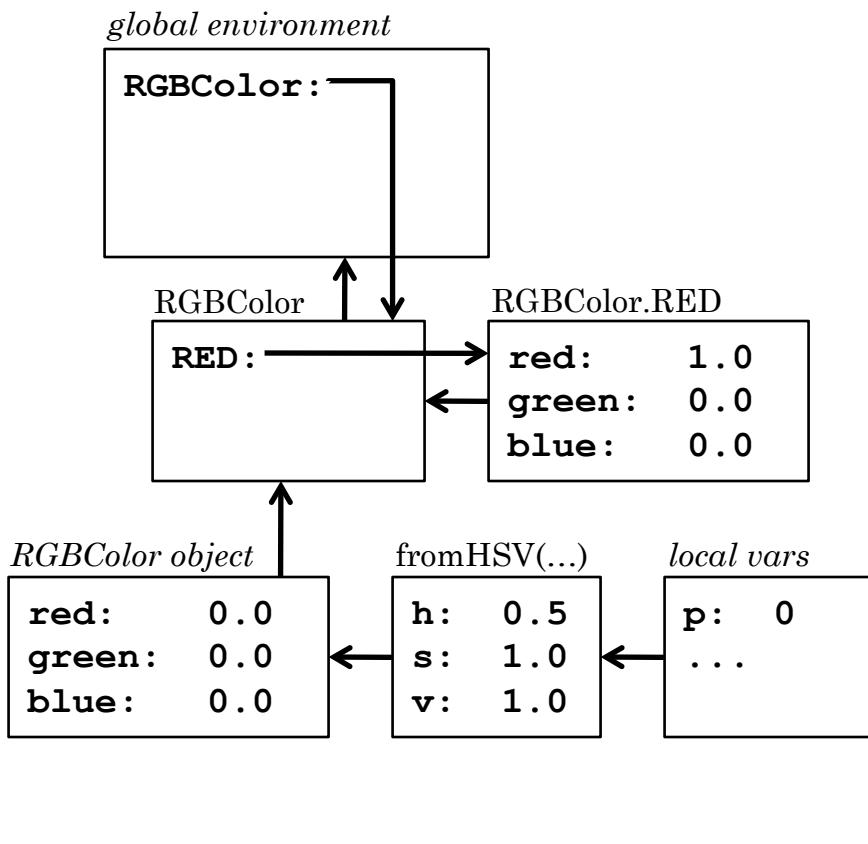
 public RGBColor(...) { ... }

 public void setRed(float v) {
 red = v;
 }
 ...

 public void fromHSV(float h,
 float s,
 float v) {
 float p = v * (1.0 - s);
 ...
 }
}
```

# JAVA TYPES AND ABSTRACTIONS (3)

- Example environment structure:



```
public class RGBColor {
 public static RGBColor RED =
 new RGBColor(1.0, 0.0, 0.0);

 private float red, green, blue;

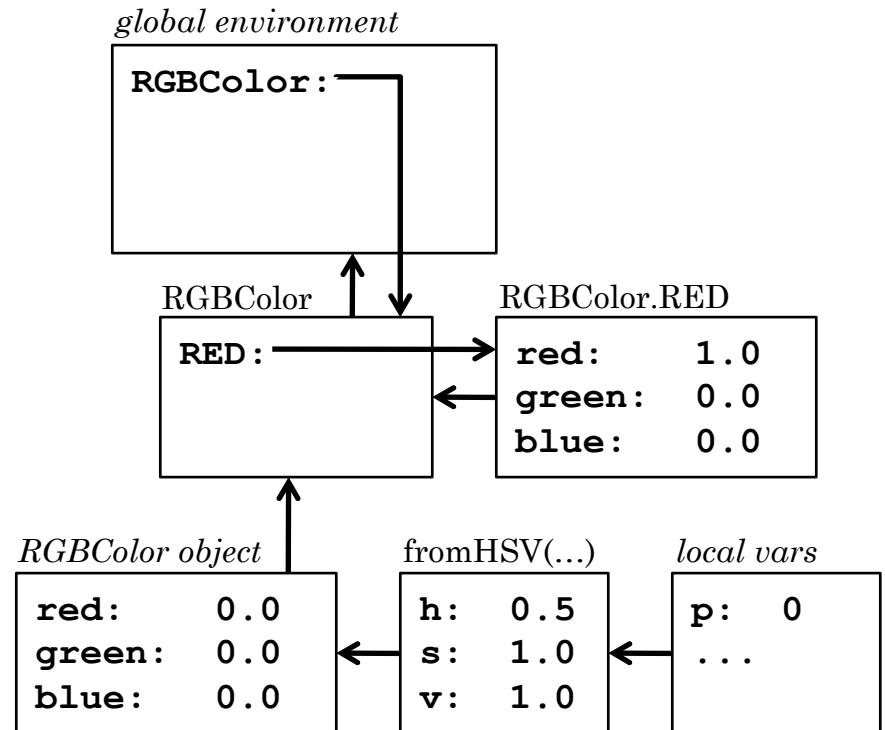
 public RGBColor(...) { ... }

 public void setRed(float v) {
 red = v;
 }
 ...

 public void fromHSV(float h,
 float s,
 float v) {
 float p = v * (1.0 - s);
 ...
 }
}
```

# JAVA TYPES AND ABSTRACTIONS (4)

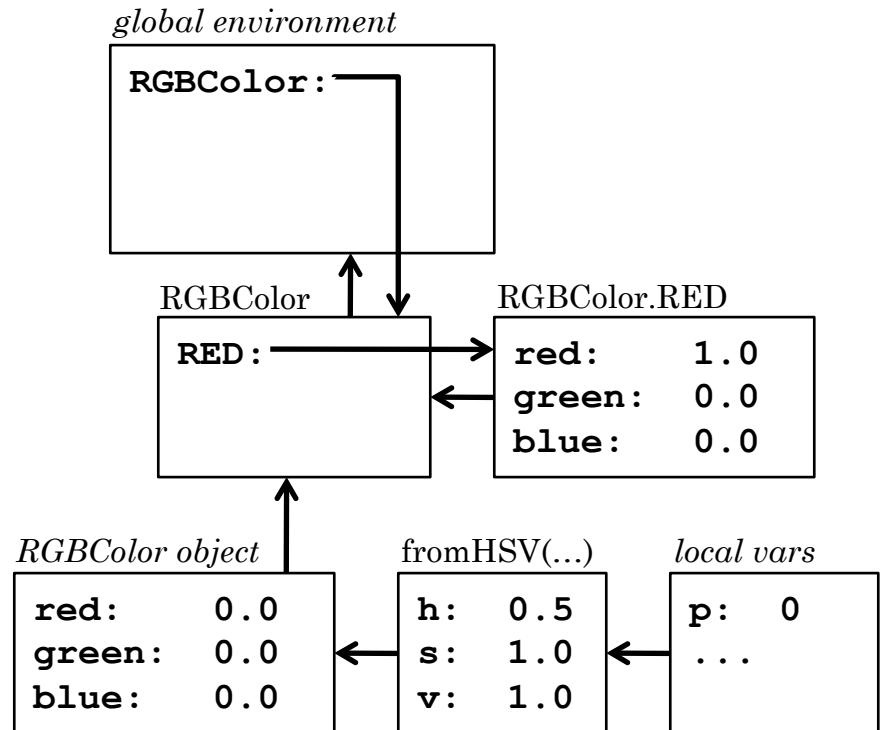
- Example environment structure:



- To support objects:
  - Need to introduce new frames to represent classes
  - Also need frames to represent specific instances of a class

# JAVA TYPES AND ABSTRACTIONS (5)

- Example environment structure:
- Can use memory heap to implement these frames
- By controlling what programs can do with references, can also provide precise garbage collection for frames
  - Clean up objects when no longer referenced by any frame
  - Can even remove class definitions when not referenced by any object (used by Java application servers for code-reloading)

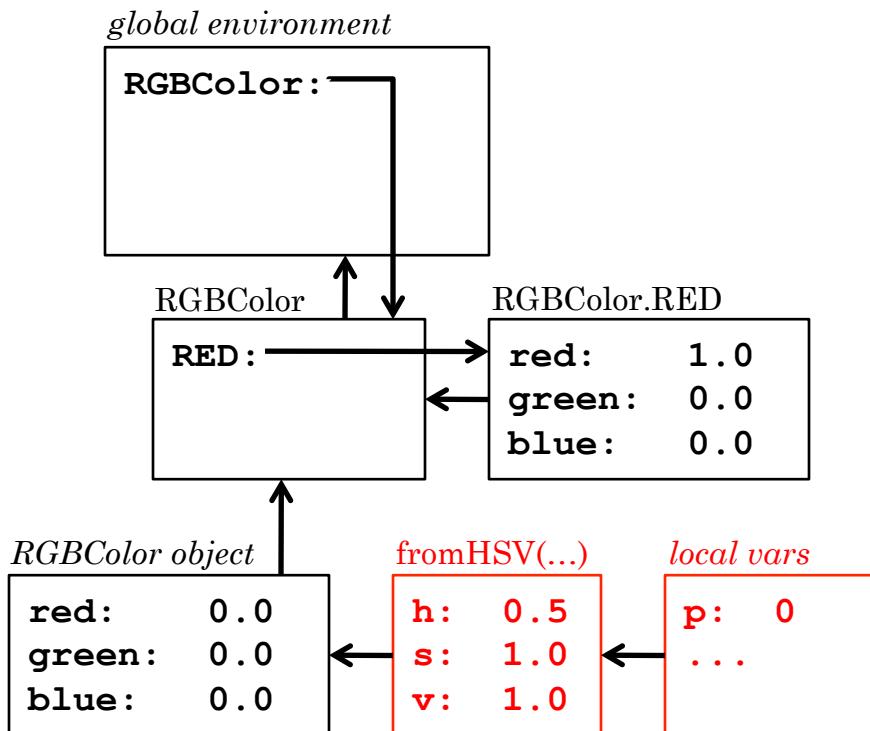


# IMPLEMENTING OOP IN C

- How might we implement this object-oriented programming model in C?
- A *very* rich topic... Definitely won't cover it all
- Start with basic object-oriented concepts
  - See how these translate into C-style concepts
- Build up the model until it includes most Java OOP capabilities
- Discussion will necessarily be at a high level
  - Many implementation details left out!!

# IMPLEMENTING OOP IN C (2)

- Can already implement some aspects of this model
  - Function calls, local variables
  - Implement these with a stack



```
public class RGBColor {
 public static RGBColor RED =
 new RGBColor(1.0, 0.0, 0.0);

 private float red, green, blue;

 public RGBColor(...) { ... }

 public void setRed(float v) {
 red = v;
 }
 ...

 public void fromHSV(float h,
 float s,
 float v) {
 float p = v * (1.0 - s);
 ...
 }
}
```

# IMPLEMENTING OOP IN C (3)

- How to store object data in C?

```
public class RGBColor {
 private float red, green, blue;
 ...
}
```

- C provides composite data types using **struct**
- Can use this to represent the data in our objects

```
struct RGBColor_Data {
 float red;
 float green;
 float blue;
};
```

- This **struct** loosely corresponds to a class declaration
- Variables of this type will represent individual objects
- Each **RGBColor** object will have its own frame for its state variables

# IMPLEMENTING OOP IN C (4)

- C representation of our object:

```
struct RGBColor_Data {
 float red;
 float green;
 float blue;
};
```

- Need a way to provide methods as well:

```
public class RGBColor {
 ...
 public float getRed() {
 return red;
 }
 public void setRed(float v) {
 red = v;
 }
}
```

- No explicit argument representing the object

```
c.setRed(0.5);
```

# METHODS AND `this`

- Need to introduce an implicit parameter into methods
  - `this`: a reference to the object the method is called on
- Object-oriented code:

```
public class RGBColor {
 ...
 public void setRed(float v) {
 red = v;
 }
 ...
}
```

- Translate into this equivalent C code:

```
RGBColor_setRed(RGBColor_Data *this, float v) {
 this->red = v;
}
```

## METHODS AND **this** (2)

- Instance methods include an implicit parameter **this**
  - Allows the object's code to refer to its own fields
- When a program calls a method on an object:
  - The underlying implementation transparently passes a reference to the called object, to the method code
- A common feature across all OO languages
  - Some languages explicitly specify this parameter
  - e.g. Python:

```
class RGBColor:
 def __init__(self, red, green, blue):
 self.red = red
 self.green = green
 self.blue = blue

 def get_red(self):
 return self.red

 ...
```

# METHODS CALLING METHODS

- Methods frequently call other methods

```
public float getGrayScale() {
 return 0.30 * getRed() +
 0.59 * getGreen() +
 0.11 * getBlue();
}
```

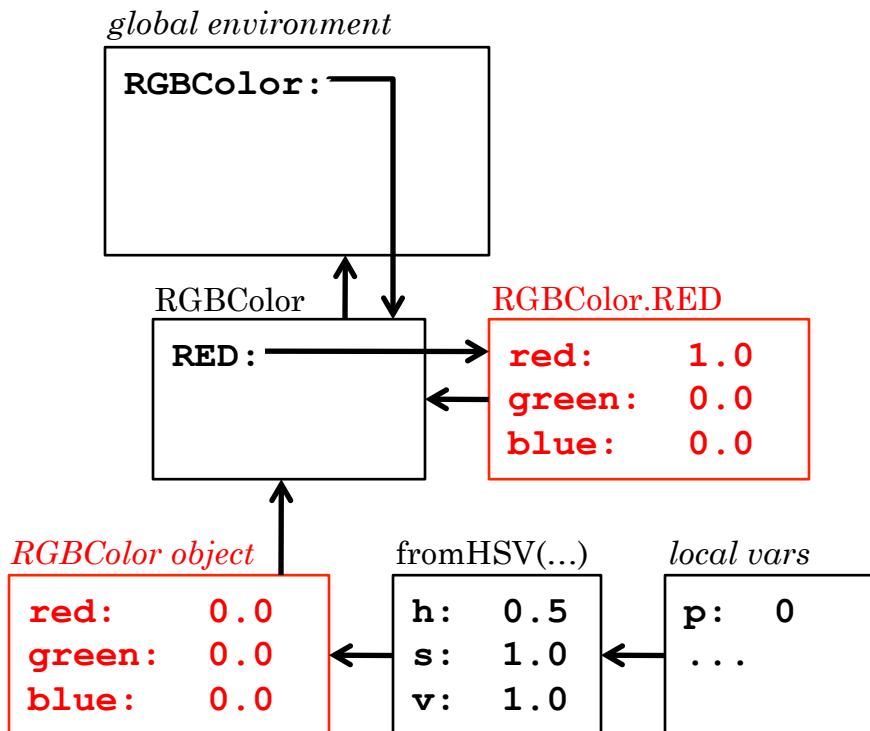
- Calls other methods on the same object
- Must pass the **this** reference to called methods

```
float RGBColor_getGrayScale(RGBColor_Data *this) {
 return 0.30 * RGBColor_getRed(this) +
 0.59 * RGBColor_getGreen(this) +
 0.11 * RGBColor_getBlue(this);
}
```

- Again, straightforward translation to support this

# OBJECT FRAMES

- This approach allows us to implement our object frames
  - Programs can manipulate independent objects of a class



```
public class RGBColor {
 public static RGBColor RED =
 new RGBColor(1.0, 0.0, 0.0);

 private float red, green, blue;

 public RGBColor(...) { ... }

 public void setRed(float v) {
 red = v;
 }
 ...

 public void fromHSV(float h,
 float s,
 float v) {
 float p = v * (1.0 - s);
 ...
 }
}
```

# FRAMES FOR CLASSES?

- Our example also has a class-level constant:

```
public class RGBColor {
 public static RGBColor RED =
 new RGBColor(1.0, 0.0, 0.0);

 ...
}
```

- Many OO languages call these static members
- Member isn't associated with a specific object
- Refer to member using the class name:  
`g.setColor(RGBColor.RED);`
- Clearly requires a frame at the class-level for such constants
- Object frames should also reference their class' frame
  - Specifies object's type, allow easy use of static members

# RGBCOLOR CLASS FRAME

- Simple frame for our **RGBColor** class:

```
struct RGBColor_Class {
 RGBColor_Data *RED; /* static member */
};
```

- Update definition of **RGBColor\_Data**:

```
struct RGBColor_Data {
 RGBColor_Class *class; /* type info */
 float red;
 float green;
 float blue;
};
```

- A new problem: how to initialize the static members?
- Classes define constructors to set up new objects...
- Similarly, init class info first time type is referenced

# RGBCOLOR CLASS FRAME

- Mechanism to initialize **RGBColor** class frame:

```
RGBColor_class_init(RGBColor_Class *class) {
 /* Initialize static member RED. */
 class->RED = malloc(sizeof(RGBColor_Data));
 RGBColor_init(class, class->RED, 1.0, 0.0, 0.0);
}
```

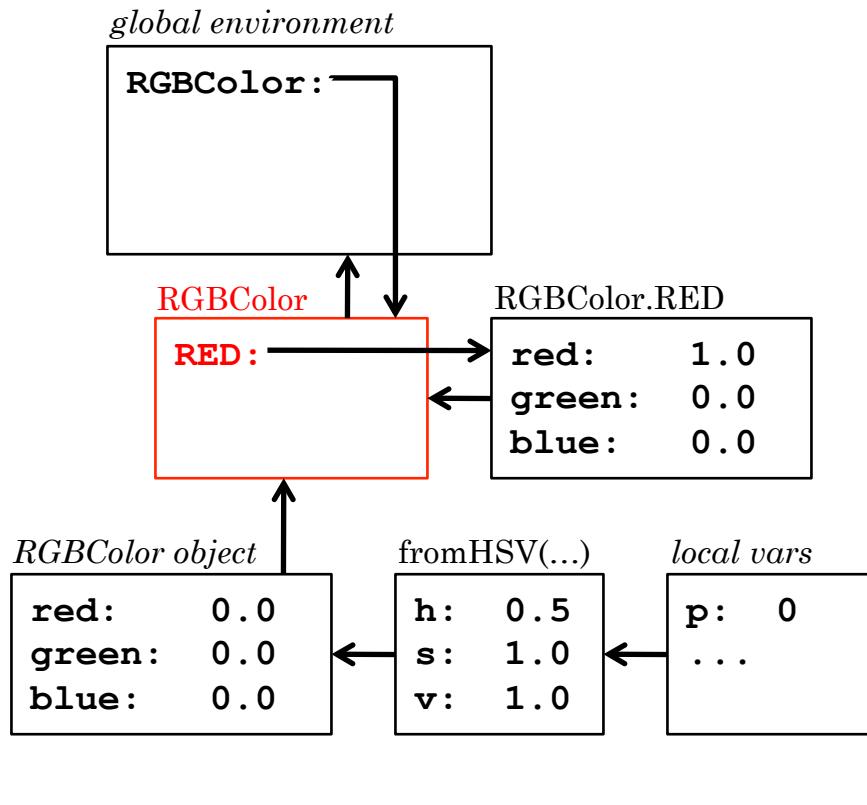
- *Global environment manages class frames, somehow... ☺*

- Simple **RGBColor** constructor translated to C:

```
RGBColor_init(RGBColor_Class *class,
 RGBColor_Data *this, float red,
 float green, float blue) {
 this->class = class;
 this->red = red;
 this->green = green;
 this->blue = blue;
}
```

# RGBCOLOR CLASS FRAME (2)

- Now we can do everything in our simplified object-oriented programming model!



```
public class RGBColor {
 public static RGBColor RED =
 new RGBColor(1.0, 0.0, 0.0);

 private float red, green, blue;

 public RGBColor(...) { ... }

 public void setRed(float v) {
 red = v;
 }
 ...

 public void fromHSV(float h,
 float s,
 float v) {
 float p = v * (1.0 - s);
 ...
 }
}
```

# MORE ADVANCED OOP CONCEPTS

- Object-oriented programming languages also support class inheritance and polymorphism
- Class inheritance:
  - Can construct hierarchies of classes
  - Parent classes represent more general-purpose types
  - Child classes are specializations of parent classes
    - Can extend functionality of parent classes with new fields and methods
    - Can override parent-class methods with specialized features
- Polymorphism:
  - Parent class specifies a common interface
  - Subclasses provide specialized implementations
  - Code written against the parent class behaves differently, depending on which subclass it is given

# CLASS INHERITANCE AND POLYMORPHISM

- Instead of a simple **RGBColor** class, provide a color class hierarchy
  - Parent class specifies methods that all subclasses will provide
  - toRGB()** produces an integer value usable by the graphics hardware
    - Bits 16-23 are red value
    - Bits 8-15 are green value
    - Bits 0-7 are blue value
- Implement two subclasses
  - RGBColor** subclass, using RGB color space
    - Red, green, blue color components
  - HSVColor** subclass, using HSV color space
    - Hue, saturation, value; effectively implements a color wheel

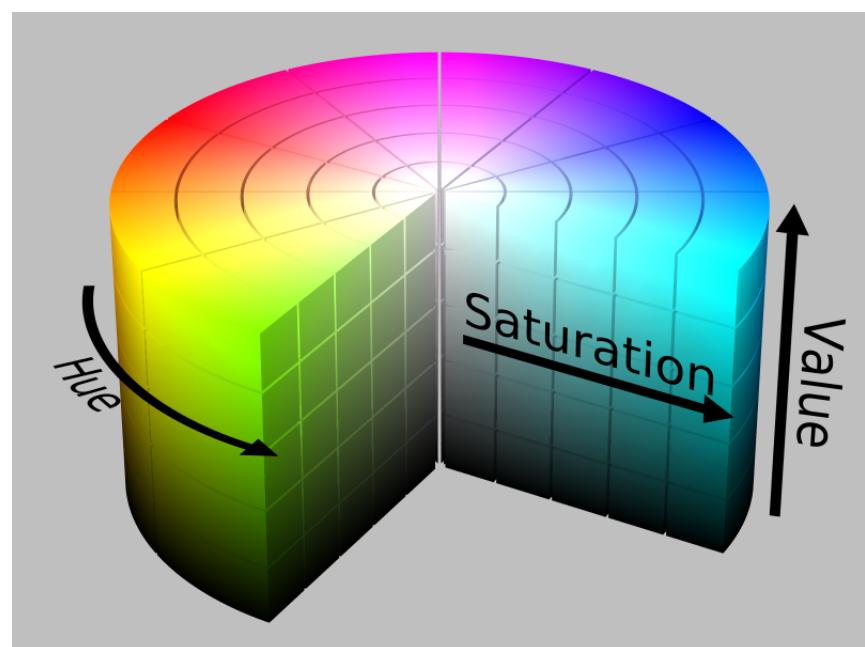
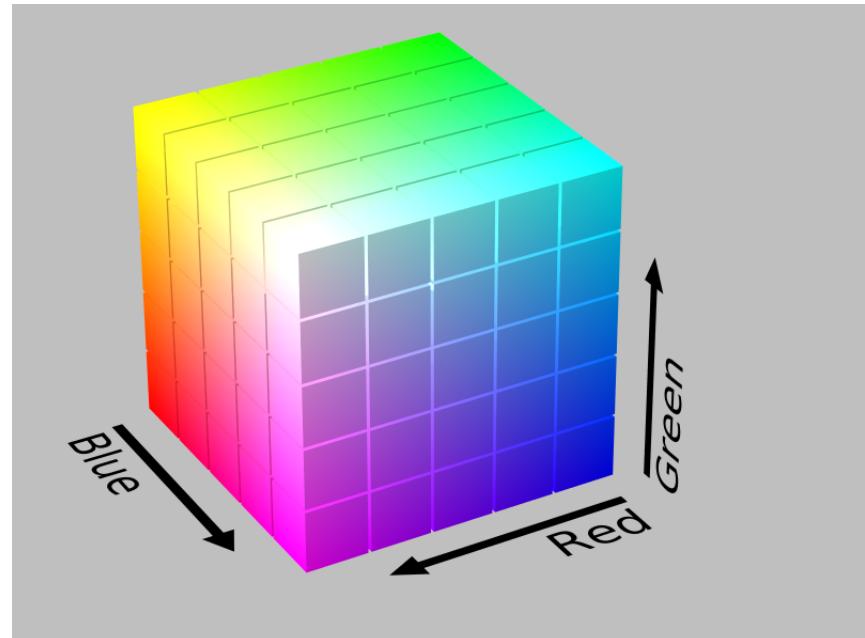
```
Color
int toRGB()
float getGrayScale()
```

Bits: 31 24 23 16 15 8 7 0



# COLOR SPACES

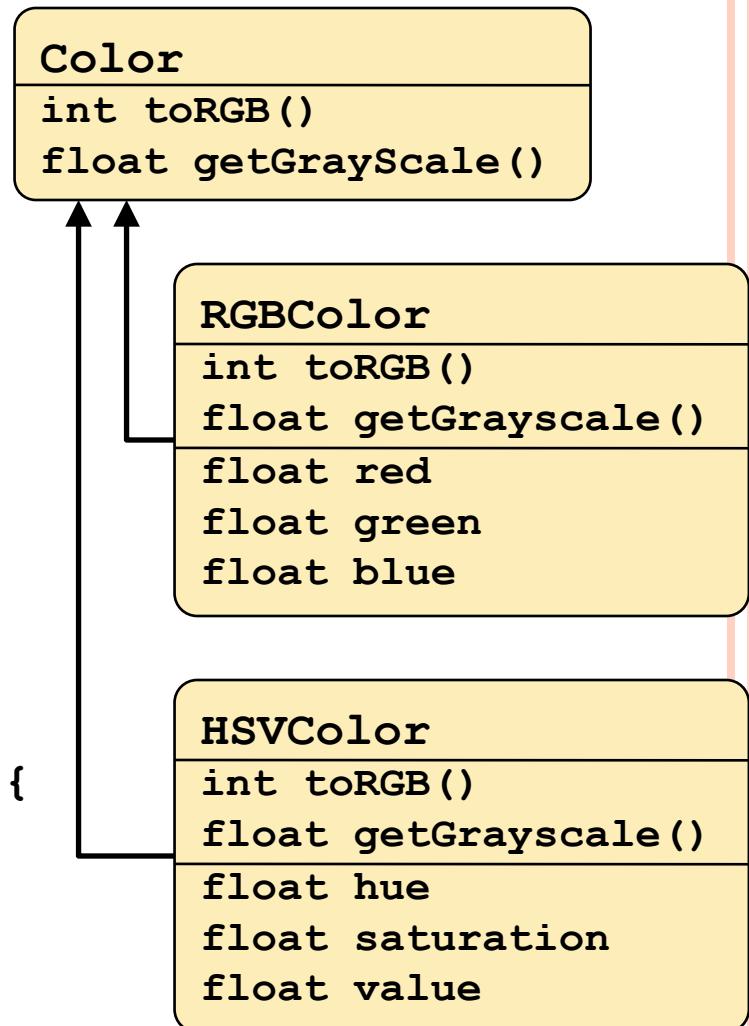
- RGB and HSV are different *color spaces*
  - Different ways of representing colors
- RGB mixes red, green, and blue components
  - Used by virtually all graphics hardware
- HSV combines hue, saturation, and value
  - Frequently used for color-choosers in UIs
  - Also used frequently in computer vision



# COLOR CLASS HIERARCHY

- Now, can implement functions that use the abstract base-class
  - `Shape.setColor(Color)`
  - `Graphics.setColor(Color)`
  - etc.
- Programs can use the type of color that makes sense for them
- Graphics code can use `toRGB()` method to set up for drawing:

```
public class Graphics {
 ...
 public void setColor(Color c) {
 device.setRGB(c.toRGB());
 }
}
```



# COLOR CLASS HIERARCHY (2)

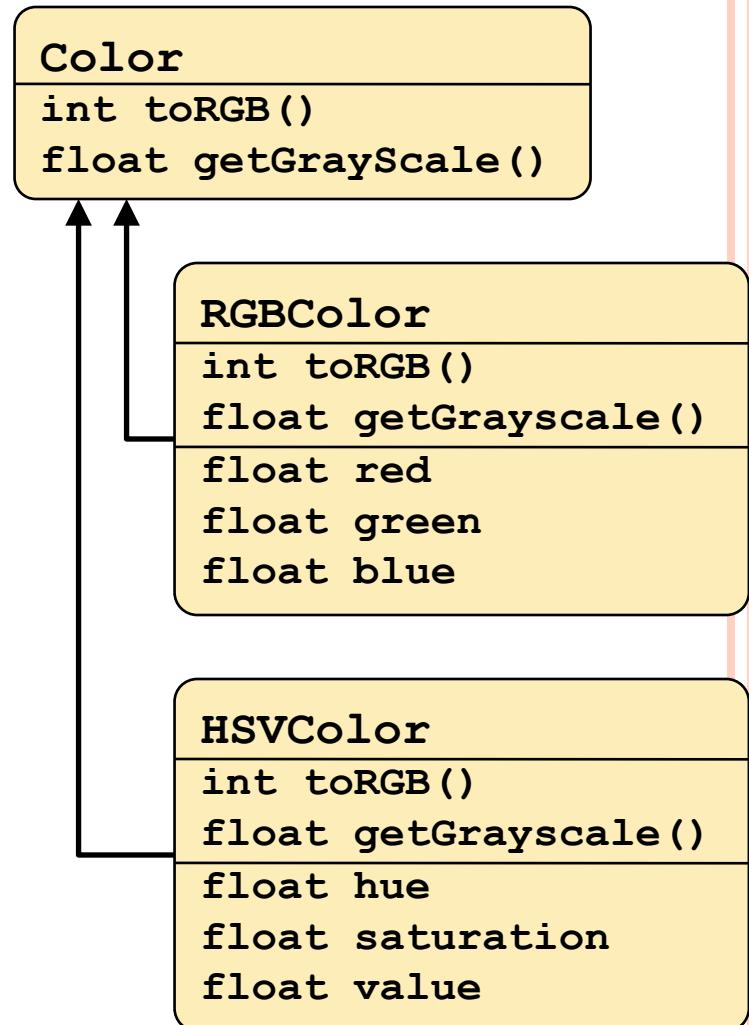
- Graphics code:

```
public class Graphics {
 ...
 public void setColor(Color c) {
 device.setRGB(c.toRGB());
 }
}
```

- **RGBColor** and **HSVColor** provide different versions of this function!

- **RGBColor** can simply pack up the RGB components into an **int**
- **HSVColor** must convert to RGB before returning the result

- **Graphics.setColor()** needs to call proper version of **toRGB()**, depending on type of argument!

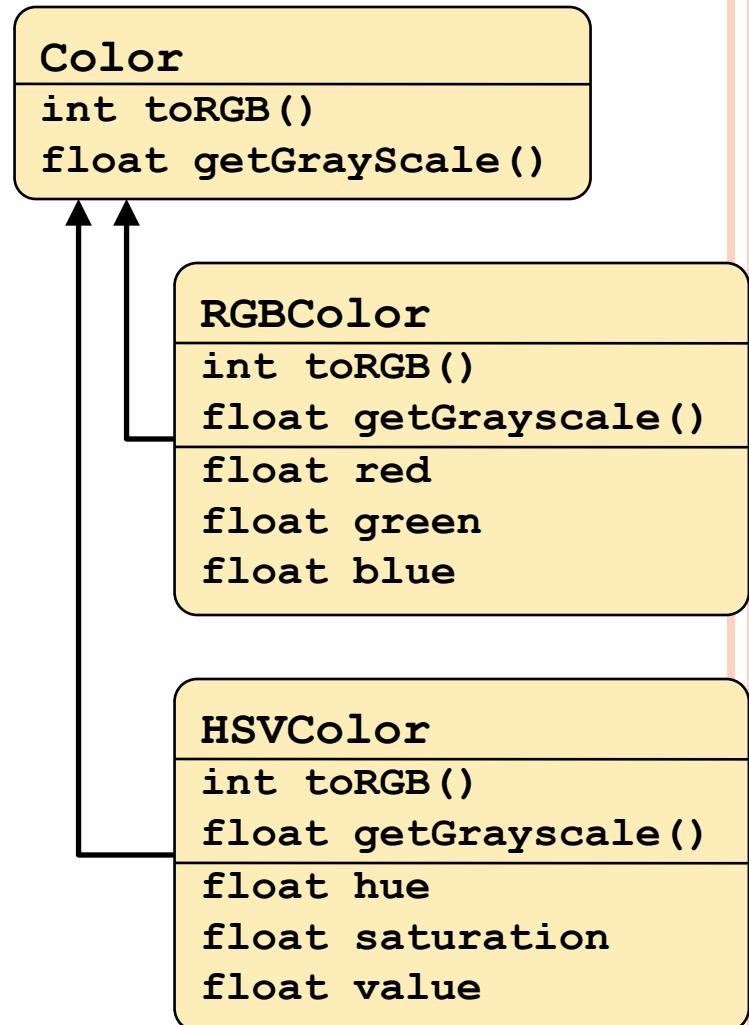


# VIRTUAL FUNCTIONS

- `toRGB()` and `getGrayScale()` are called virtual functions
- Subclasses have provided their own implementations...
- Code written against base-class must call appropriate version when passed a subclass object

```
public class Graphics {
 ...
 public void setColor(Color c) {
 device.setRGB(c.toRGB());
 }
}
```

- Somehow, objects must indicate which version of `toRGB()` to use



# CLASSES AND FUNCTION POINTERS

- Each object already has a reference to its class info...
- Simple solution:
  - Add details of which methods go with each type, into the class information
  - Can look up which method to call, using object's class-info
- The C language supports *function pointers*
  - Instead of a pointer to data, points to a function
  - A function-pointer **fp**, which points to a function that takes a **double** and returns a **double**:

```
double (*fp) (double);
```
  - Set **fp** to point to the **sin()** function:

```
fp = sin; /* Note: NO parentheses!! */
```
  - Call the **sin()** function through **fp**:

```
result = fp(x);
```

# COLOR CLASS DETAILS

- Base class representation:

```
struct Color_Class {
 int (*toRGB) (Color_Data *this);
 float (*getGrayScale) (Color_Data *this);
};
```

- Two function pointers, one for each virtual function

- Subclass type information is identical

- (These subclasses don't have static members...)

```
struct RGBColor_Class {
 int (*toRGB) (RGBColor_Data *this);
 float (*getGrayScale) (RGBColor_Data *this);
};
```

```
struct HSVColor_Class { ... /* same idea */ };
```

## COLOR CLASS DETAILS (2)

- Can model basic class-inheritance with C structs
- Declare “base-type” struct with certain members

```
struct Color_Data {
 Color_Class *class;
};
```

- “Sub-type” structs can add other members if needed, but must have same types of members at the start!

```
struct RGBColor_Data {
 RGBColor_Class *class;
 float red;
 float green;
 float blue;
};
```

- Then, can cast a base-type pointer to subtype pointer
  - The common members are at same offsets in both structs

# COLOR CLASS INITIALIZATION

- Now our class-initialization code becomes:

```
RGBColor_class_init(RGBColor_Class *class) {
 /* Initialize function pointers */
 class->toRGB = RGBColor_toRGB;
 class->getGrayScale = RGBColor_getGrayScale;

}

HSVColor_class_init(HSVColor_Class *class) {
 /* Initialize function pointers */
 class->toRGB = HSVColor_toRGB;
 class->getGrayScale = HSVColor_getGrayScale;
}
```

- Objects of each type can easily invoke the proper version of `Color.toRGB()` now

# COLOR-OBJECT DATA TYPES

- **RGBColor\_Data** definition is same as before:

```
struct RGBColor_Data {
 RGBColor_Class *class; /* type info */
 float red;
 float green;
 float blue;
};
```

- **HSVColor\_Data** definition:

```
struct HSVColor_Data {
 HSVColor_Class *class; /* type info */
 float hue;
 float saturation;
 float value;
};
```

# GRAPHICS CODE TRANSLATION

- Our Graphics code from before:

```
public class Graphics {
 ...
 public void setColor(Color c) {
 device.setRGB(c.toRGB());
 }
}
```

- Translate into C code:

```
void GraphicsSetColor(Graphics_Data *this,
 Color_Data *c) {
 Device_SetRGB(this->device, c->class->toRGB(c));
}
```

- If **RGBColor** passed in, **RGBColor\_toRGB()** is used
- If **HSVColor** passed in, **HSVColor\_toRGB()** is used

# GRAPHICS CODE TRANSLATION (2)

- Note the two different calling patterns:

```
void GraphicsSetColor(Graphics_Data *this,
 Color_Data *c) {
 Device_setRGB(this->device, c->class->toRGB(c));
} non-virtual method invocation virtual method invocation
```

- Non-virtual methods do not support polymorphism
  - The method is chosen at compile-time, and cannot change
    - Also called static dispatch
  - Doesn't require an extra lookup, so it's faster
- Virtual methods do support polymorphism:
  - Method is determined at run-time, from the object itself
    - Also called dynamic dispatch
  - *Essential* when methods are overridden by subclasses!
  - Slightly slower, due to the extra lookup

# OBJECT ORIENTED PROGRAMMING MODEL

- Our simple example now supports simple class hierarchies and polymorphism
- Conceptually straightforward to implement in C
  - Structs to represent data for objects and classes
  - Implement virtual functions by storing function-pointers in the class descriptions
  - Look up which virtual function to call at run-time,*directly from the object itself*
- Note 1: almost all Java methods are virtual
  - ...unlike C++, where member functions must explicitly be declared virtual

# OOP MODEL (2)

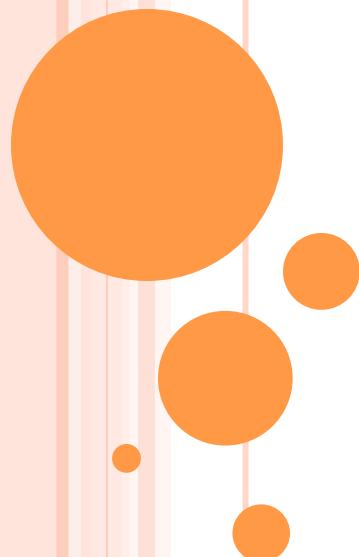
- Note 2:
  - Many OOP languages represent virtual function pointers with a *virtual-function pointer table* (a.k.a. vtable)
- Our representation:

```
struct color_class {
 int (*toRGB) (Color_Data *this);
 float (*getGrayScale) (Color_Data *this);
};
```

  - Our simple example includes more type information
- Frequently:
  - Class information contains an array of virtual function pointers (or references)
  - Individual functions are often referred to by slot-index
    - e.g. slot 0 = **toRGB()**, slot1 = **getGrayScale()**
- Some languages (like Java) refer to functions by name

## OOP MODEL (3)

- Note 3: Our example is distinctly hard-coded...
  - Mapped our example classes to C structs and code
  - Doesn't support the same ability to dynamically load and run code that the Java VM provides!
- Java virtual machine uses sophisticated data structures to represent class information
  - ...including fields, method signatures, method definitions, class hierarchy information...
- Allows Java VM to dynamically load class definitions and execute them
  - Even allows Java programs to generate new classes on the fly, then load and run them!



# CS24: INTRODUCTION TO COMPUTING SYSTEMS

Spring 2015  
Lecture 11

# EXCEPTION HANDLING

- Many higher-level languages provide exception handling
- Concept:
  - One part of the program knows how to detect a problem, but not how to handle it in a general way
  - Another part of the program knows how to handle the problem, but can't detect it
- When a problem is detected, the code throws an exception
  - An “exception” is a value representing the error
  - Frequently, an object that contains the error’s details
  - The exception’s type indicates the category of error
- Code that knows how to handle the problem can catch the exception
  - Provides an exception handler that responds to the error

# JAVA EXCEPTION HANDLING

- Java exception handling uses **try/catch** blocks

```
public static void main(String[] args) {
 loadConfig(args);
 try {
 double x = readInput();
 double result = computeValue(x);
 System.out.println("Result = " + result);
 }
 catch (IllegalArgumentException e) {
 printError(e.getMessage());
 }
}
```

- If input is invalid, **computeValue()** throws an exception
- Execution immediately transfers to the exception handler for **IllegalArgumentException**

# JAVA EXCEPTION HANDLING (2)

- Only exceptions from within **try** block are handled!

```
public static void main(String[] args) {
 loadConfig(args);
 try {
 double x = readInput();
 double result = computeValue(x);
 System.out.println("Result = " + result);
 }
 catch (IllegalArgumentException e) {
 printError(e.getMessage());
 }
}
```

- If **loadConfig()** throws, the exception isn't handled here
- try**: “If *this* code throws, I want to handle the exceptions.”
  - (Assuming the exception matches one of the **catch** blocks...)

## JAVA EXCEPTION HANDLING (3)

- Code can report an exception by **throw**-ing it:

```
double computeValue(double x) {
 if (x < 3.0) {
 throw new IllegalArgumentException(
 "x must be at least 3");
 }
 return Math.sqrt(x - 3.0);
}
```

- Now the function can complete in two ways:
  - Normal completion: returns the computed result
  - Abnormal termination:
    - Function stops executing immediately when **throw** occurs
    - Program execution jumps to the nearest enclosing **try/catch** block with a matching exception type

# EXCEPTIONS WITHIN A FUNCTION

- Exceptions can be used within a single function

```
static void loadConfig(String[] args) {
 try {
 for (int i = 0; i < args.length; i++) {
 if (args[i].equals("-n")) {
 i++;
 if (i == args.length)
 throw new Exception("-n requires a value");
 ...
 }
 else if ...
 }
 }
 catch (Exception e) {
 System.err.println(e.getMessage());
 showUsage(); System.exit(1);
 }
}
```

- Used to signal an error in argument-parsing code

# EXCEPTIONS SPANNING FUNCTIONS

- Exceptions can also span multiple function calls
  - Doesn't have to be handled by immediate caller of function that throws!
- Example:

```
Webpage loadPage(URL url) {
 try {
 InputStream in = sendHttpRequest(url);
 ...
 }
 catch (UnknownHostException e) ...
}

InputStream sendHttpRequest(URL url) {
 Socket sock = new Socket(url.getHost(), url.getPort());
 ...
}
```

- Socket** constructor could throw an exception
  - Propagates out of **sendHttpRequest()** function...
  - Exception is handled in **loadPage()** function

# EXCEPTION HANDLING REQUIREMENTS

- A challenging feature to implement!
  - Can throw objects containing arbitrary information
  - Exception can stay within a single function, or propagate across multiple function calls
  - Actual **catch**-handler that receives the exception, depends on who called the function that threw
    - A function can be called from multiple places...
    - A thrown exception should be handled by the nearest *dynamically-enclosing* **try/catch** block
- Also want exception passing to be fast
  - Ideally, won't impose any overhead on the program until an exception is actually thrown
  - Assumption: exceptions aren't thrown very often
    - ...*hence the name “exception”...*
    - (Not always a great assumption these days, but oh well!)

# IMPLEMENTING EXCEPTION HANDLING

- With exception handling, there are two important points in program execution
- When execution enters a **try** block:
  - Some exceptions might be handled by this **try/catch** block...
  - May need to do some kind of bookkeeping so we know where to jump back to in case an exception is thrown
- When an exception is actually thrown:
  - Need to jump to the appropriate **catch** block
  - Need to access information from previous **try**-point, so that we can examine the proper set of catch blocks
  - This will frequently span multiple stack frames

# EXCEPTIONS WITHIN A FUNCTION

- When exception is thrown and caught within a single function:

```
void foo() {
 try {
 ...
 if (failed)
 throw new Exception();
 ...
 }
 catch (Exception e) {
 ... // Handle the exception
 }
}
```

- In this case, can translate **throw** into a simple jump to the appropriate exception handler
  - Types are available at compile time

## EXCEPTIONS WITHIN A FUNCTION (2)

- Still need some way to pass exception object to the handler...

```
void foo() {
 try {
 ...
 if (failed)
 throw new Exception();
 ...
 }
 catch (Exception e) {
 ... // Handle the exception
 }
}
```

- Assume there will be at most one exception in flight at any given time
- Store [reference to] the exception in a global variable

# EXCEPTIONS WITHIN A FUNCTION (3)

- One possible translation of our code:

```
void foo() {
 ... // Code that sets up failed flag
 if (failed) {
 set_exception(new Exception()); // throw
 goto fooCatchException;
 }
 ... // Other code within try block
foo_end_try: // End of try-block
 goto foo_end_trycatch;

fooCatchException: {
 e = get_exception();
 ... // Handle the exception
 goto foo_end_trycatch;
}

foo_end_trycatch:
 return;
}
```

# EXCEPTIONS SPANNING FUNCTIONS

- Not a good general solution! Normal case is to have exceptions spanning multiple function calls.
- Can't implement with **goto**, since **goto** can't span multiple functions!
  - Really can't hard-code the jump now, anyway...
- Really want a way to *record* where to jump, dynamically
  - i.e. when we enter **try** block
- Then, a way to jump back to that place, even across multiple function calls
  - i.e. when exception is thrown

```
int f(int x) {
 try {
 return g(3 * x);
 } catch (Exception e) {
 return -1;
 }
}

int g(int x) {
 return h(15 - x);
}

int h(int x) {
 if (x < 5)
 throw new Exception();
 return Math.sqrt(x - 5);
}
```

# `setjmp()` AND `longjmp()`

- C standard includes two very interesting functions:
- `int setjmp(jmp_buf env)`
  - Records current execution state into `env`, at exact time of `setjmp()` call
    - Information recorded includes callee-save registers, `esp`, and caller return-address
  - Always returns 0
- `void longjmp(jmp_buf env, int val)`
  - Restores execution state from `env`, back into all registers saved by `setjmp()`
  - `esp` is restored from `env`:
    - Any intervening stack frames are discarded
  - Stack is restored to the state as when `setjmp()` was called
    - Caller return-address on stack when `setjmp()` was called
  - Then `longjmp()` returns, with `val` in `%eax`
    - (or `%eax = 1` if `val` is 0)
- To caller, it appears that `setjmp()` returned again!

## setjmp() AND longjmp() (2)

- Previous example is simple enough to implement using `setjmp()` and `longjmp()`:

```
int f(int x) {
 try {
 return g(3 * x);
 } catch (Exception e) {
 return -1;
 }
}

int g(int x) {
 return h(15 - x);
}

int h(int x) {
 if (x < 5)
 throw new Exception();

 return Math.sqrt(x - 5);
}
```

```
static jmp_buf env;

int f(int x) {
 if (setjmp(env) == 0)
 return g(3 * x);
 else
 return -1;
}

int g(int x) {
 return h(15 - x);
}

int h(int x) {
 if (x < 5)
 longjmp(env, 1);

 return sqrtl(x - 5);
}
```

## **setjmp()** AND **longjmp()** (3)

- When we enter **try** block, record execution state in case an exception is thrown
- If an exception is thrown, use **longjmp()** to return to where **try** was entered
  - Stack frames of intervening function calls are discarded!*
- Return value of **setjmp()** indicates whether an exception was thrown
  - Example has only one kind of exception, so any return-value will do

```
static jmp_buf env;

int f(int x) {
 if (setjmp(env) == 0)
 return g(3 * x);
 else
 return -1;
}

int g(int x) {
 return h(15 - x);
}

int h(int x) {
 if (x < 5)
 longjmp(env, 1);

 return sqrtl(x - 5);
}
```

## setjmp() / longjmp() EXAMPLE

- What happens with **f(5)** ?
- Things will go badly... ☺

```
static jmp_buf env;

int f(int x) {
 if (setjmp(env) == 0)
 return g(3 * x);
 else
 return -1;
}

int g(int x) {
 return h(15 - x);
}

int h(int x) {
 if (x < 5)
 longjmp(env, 1);

 return sqrtl(x - 5);
}
```

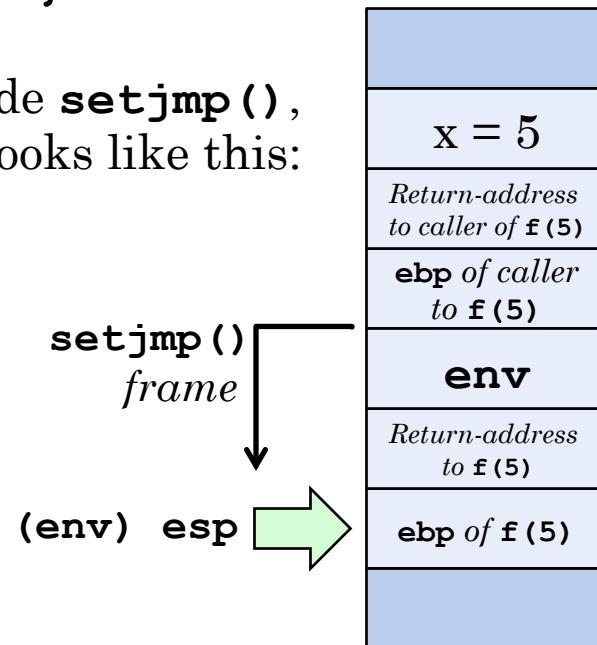
## **setjmp()**/**longjmp()** EXAMPLE (2)

- **f(5)** calls **setjmp()** to prepare for an exception
  - Will return 0 since it's actually the **setjmp()** call
- **setjmp()** stores:
  - Callee-save registers, including current **esp**
  - **setjmp()**-caller's **ebp** and return address
    - (grab these from stack)
- **env** now holds everything necessary for **longjmp()** to act like it's **setjmp()**...

```
static jmp_buf env;

int f(int x) {
 if (setjmp(env) == 0)
 return g(3 * x);
 else
 return -1;
}
```

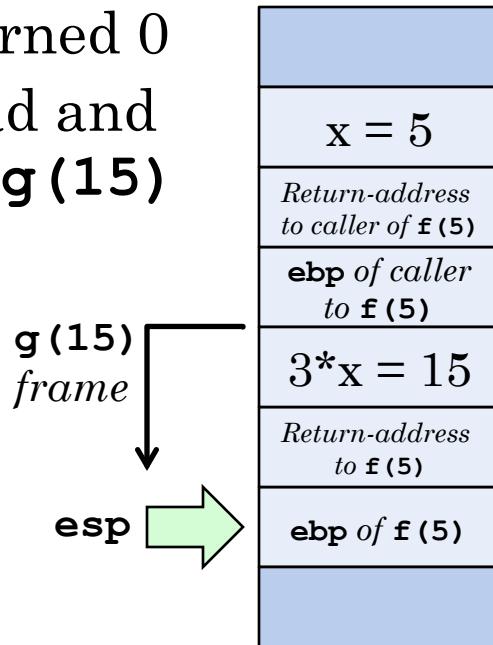
Inside **setjmp()**, stack looks like this:



## setjmp() / longjmp() EXAMPLE (3)

- **setjmp()** returned 0
- **f(5)** goes ahead and calls **g(3\*x) = g(15)**

- Now the stack looks like this:



- Note that the stack frame from calling **setjmp()** is long gone...
  - (along with the return-address to where **setjmp()** was called from)
  - **env** still contains these values!

```
static jmp_buf env;

int f(int x) {
 if (setjmp(env) == 0)
 return g(3 * x);
 else
 return -1;
}

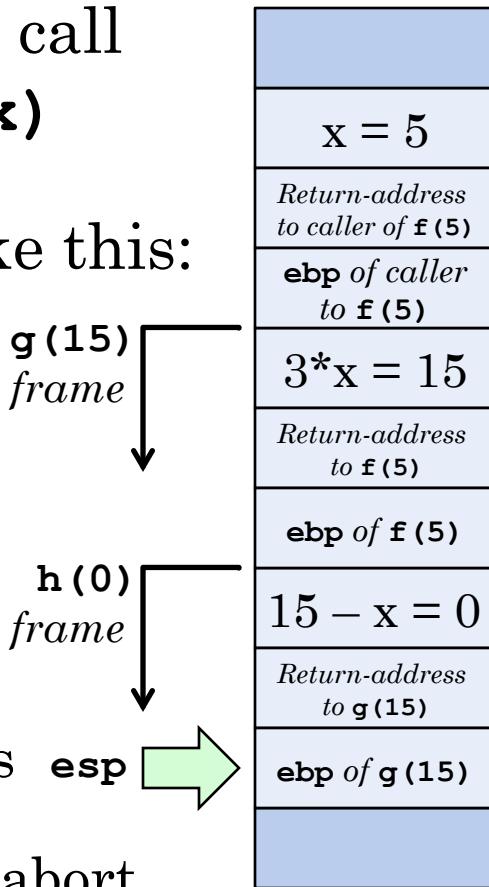
int g(int x) {
 return h(15 - x);
}

int h(int x) {
 if (x < 5)
 longjmp(env, 1);

 return sqrtl(x - 5);
}
```

## setjmp() / longjmp() EXAMPLE (4)

- Now in **g(15)** call
- g** calls **h(15-x) = h(0)**
- Stack looks like this:



- Problem:
  - h()** can't handle values **esp** less than 5
  - h()** needs to abort the computation

```
static jmp_buf env;

int f(int x) {
 if (setjmp(env) == 0)
 return g(3 * x);
 else
 return -1;
}

int g(int x) {
 return h(15 - x);
}

int h(int x) {
 if (x < 5)
 longjmp(env, 1);

 return sqrtl(x - 5);
}
```

## `setjmp()`/`longjmp()` EXAMPLE (5)

- `h(0)` needs to abort!
  - Got a bad argument...
- `h()` “throws an exception”
  - Calls `longjmp()` to switch back to nearest enclosing `try`-block
  - `env` contains details of where nearest enclosing `try`-block is...
- `longjmp()` restores execution state back to execution in `f()`
- `f()` “catches the exception”
  - It now sees `setjmp()` return a nonzero result, indicating there was an exception...
  - `f()` returns -1 as final result

```
static jmp_buf env;

int f(int x) {
 if (setjmp(env) == 0)
 return g(3 * x);
 else
 return -1;
}

int g(int x) {
 return h(15 - x);
}

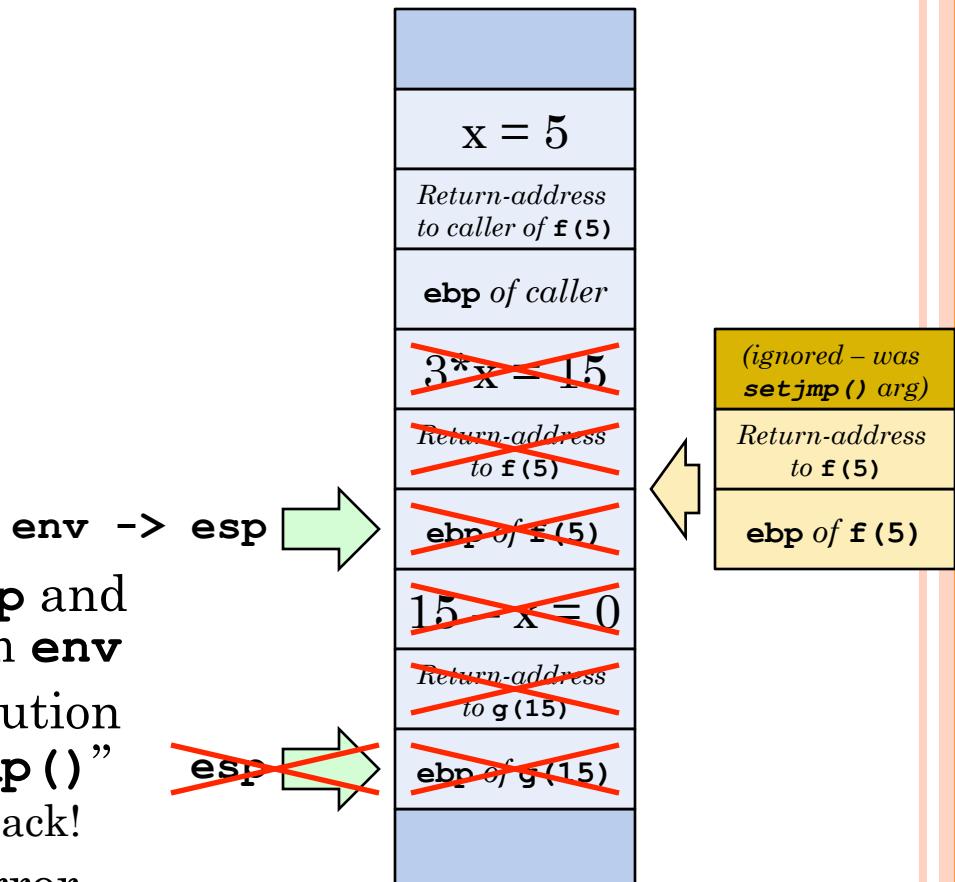
int h(int x) {
 if (x < 5)
 longjmp(env, 1);

 return sqrtl(x - 5);
}
```

## setjmp () / longjmp () EXAMPLE (6)

```
int f(int x) {
 if (setjmp(env) == 0)
 ...
}
...
int h(int x) {
 if (x < 5)
 longjmp(env, 1);
 ...
}
```

- When **h()** calls **longjmp()**, **esp** and caller **eip/ebp** are restored from **env**
- When **longjmp()** returns, execution resumes in **f()**, “back at **setjmp()**”
  - Caller has no idea who returned back!
- Result of **setjmp()** indicates error
  - (but it's technically **longjmp()**'s result...)
- f()** handles the error appropriately



# HOW DO THESE THINGS WORK?!

- **setjmp()** and **longjmp()** must be implemented in assembly language
  - No C mechanism for saving and restoring registers
  - No C mechanism for manipulating the stack this way
- Implementation is also very platform-specific!
  - Size of **jmp\_buf** corresponds to how many registers need to be saved and restored
  - Linux on IA32 uses 8 dwords
  - MacOS X on IA32 uses 18 dwords (!)
  - MacOS X on PPC uses 192 dwords (!!!)
    - RISC processors tend to have a large number of registers, due to load/store architecture
  - Specification is ambiguous about exactly what needs to be saved...

# HOW DO THESE THINGS WORK?! (2)

- Implementing **setjmp()**/**longjmp()** is surprisingly straightforward
  - Simply requires understanding of stack frames in cdecl
- In **setjmp()**, must know how to save the caller's execution state, to fake a return from **setjmp()**
  - Return-address where caller invoked **setjmp()** from
  - **esp** value inside **setjmp()**
  - (also the callee-save registers, since they will change before **longjmp()** is called...)
- In **longjmp()**, just need to manipulate the stack to restore this execution state, then **ret**!
  - Caller will see return-value in **eax** like usual
  - Returns to where caller invoked **setjmp()** from
  - *They'll never know the difference!*

# MULTIPLE CATCH BLOCKS

- A **try** block can have multiple **catch** blocks

```
Webpage loadPage(String urlText) {
 try {
 Socket s = httpConnect(urlText);
 ...
 }
 catch (MalformedURLException e) {
 ...
 }
 catch (UnknownHostException e) {
 ...
 }
}
```

- Easy to support this with **setjmp()** and **longjmp()**
  - **longjmp()** can simply pass a different integer value for each kind of exception
  - Compiler can assign integer values to all exception types

# MULTIPLE CATCH BLOCKS (2)

- One possible translation:

```
jmp_buf env;
...
switch (setjmp(env)) {
case 0: /* Normal execution */
 ... // Translation of
 ... // Socket s = httpConnect(urlText);
 break;

case 1037: /* Caught MalformedURLException */
 ...
 break;
case 1053: /* Caught UnknownHostException */
 ...
 break;
}
```

- Code that calls `longjmp()` passes exception-type in call
- Many details left out, involving variable scoping, etc.*

# NESTED EXCEPTION HANDLERS

- One major flaw in our implementation:
  - **try/catch** blocks can be nested within each other!
  - We only have one **jmp\_buf** variable in our example
  - A nested **try/catch** block would overwrite the outer **try**-block's values stored in the **jmp\_buf**
- Solution is straightforward:
  - Introduce a “try-stack” for managing the **jmp\_buf** values of nested **try/catch** blocks
  - When we enter into a new **try**-block, push the new **try/catch** handler state (**jmp\_buf**) onto try-stack
  - This is separate from the regular program stack
  - (It doesn't strictly *have* to be separate, but to keep things simple, we will keep it separate!)

## NESTED EXCEPTION HANDLERS (2)

- Once we have a try-stack for nested handlers, need some basic exception handling operations
- When program enters a **try** block:
  - Call **setjmp()**, and if it returns 0 then push the **jmp\_buf** onto the **try**-stack
  - **Note:** Cannot call **setjmp()** in a separate helper function that does these things for us! **Why not?**
    - Left as an exercise for the student...
- When program exits the **try**-block without any exceptions:
  - Need to pop the topmost **jmp\_buf** off of the try-stack
  - Can do this in a helper function

# NESTED EXCEPTION HANDLERS (3)

- When an exception is thrown:

```
void throw_exception(int exception_id)
```

- Helper function that pops the topmost `jmp_buf` off of the try-stack, and then uses it to do `longjmp(exception_id)`

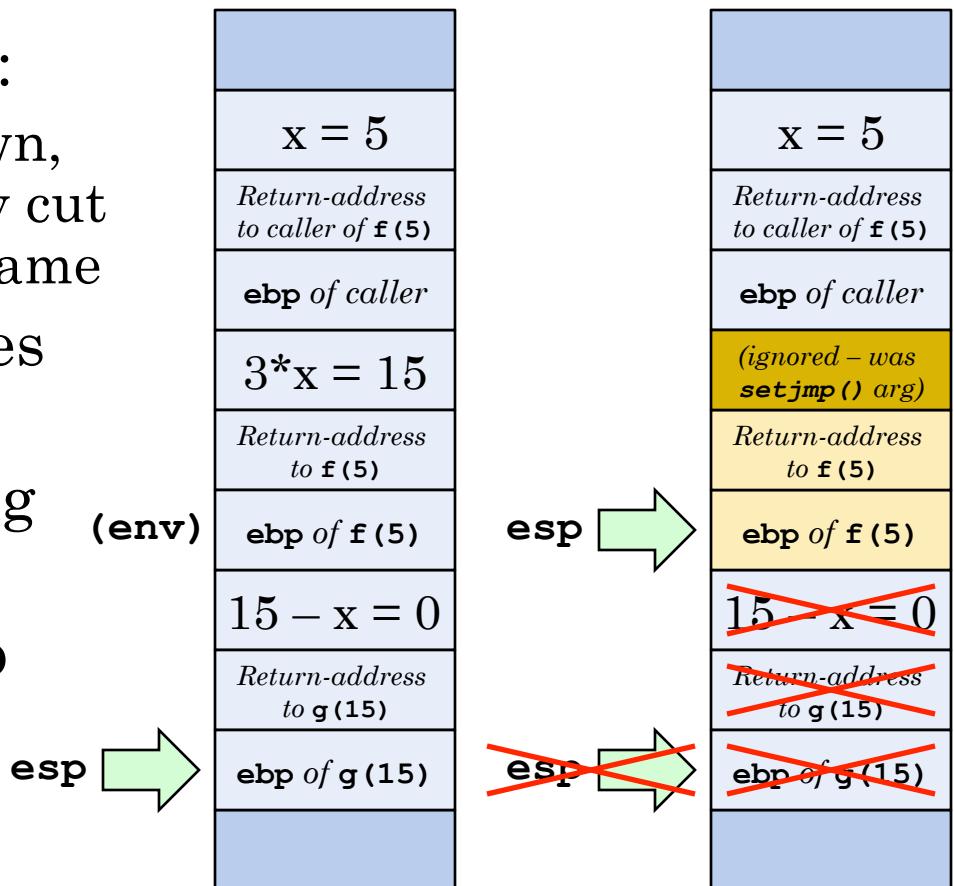
- If an exception isn't handled by a `try/catch` block, or if `catch`-block re-throws the exception:

- Just invoke `throw_exception()` again with same ID
- Next enclosing `try`-block's `jmp_buf` is now on top of stack
- (Do this in `default` branch, or `else`-clause if using `if`.)

- With these tools in place, can easily handle nested exception-passing scenarios
- An example of this is provided in Assignment 4! ☺

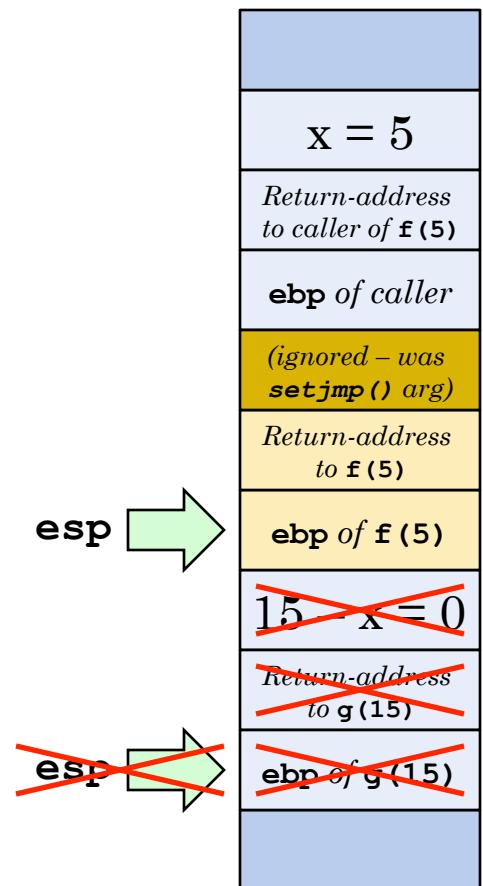
# STACK CUTTING

- This kind of exception-handling implementation is called stack cutting
- From previous example:
  - When exception is thrown, the stack is immediately cut down to the handler's frame
- Intervening stack frames are simply eliminated!
- *Very fast* for propagating exceptions...
- Unacceptable if cleanup needs to be done for intervening functions!



## STACK CUTTING (2)

- Can perform cleanup for intervening functions, if we keep track of what needs to be done
  - e.g. manage a list of resources that need cleaned up after each function returns
  - When exception is thrown, can use this info to clean up properly
  - Starting to get a bit *too* complicated...
- For languages with GC, don't really have much to clean up from functions
  - Just drop object-references from stack
  - Garbage collector will detect that objects are no longer reachable, and will eventually reclaim the space



# STACK UNWINDING

- Another solution to the exception-propagation problem: stack unwinding
  - Solution used by Java Virtual Machine, most C++ implementations, etc.
  - Unlike stack cutting, each stack frame is cleaned up individually. Much better for resource management!
- Remember, the important times in exception handling are:
  - When we are inside of a **try**-block – a thrown exception might be handled by this **try/catch**
  - When an exception is actually thrown
- The compiler generates an exception table for every single function in the program
  - All exception handling is driven from these tables

# EXCEPTION TABLES

- Each function has an exception table, containing:
  - A range of addresses [*from\_eip*, *to\_eip*], specifying the instructions the **try**-block encapsulates
  - An exception that the **try**-block can handle
  - The address of the handler for that exception
- Our example from before:

```
Webpage loadPage(String urlText) {
 try {
 Socket s = httpConnect(urlText);
 ...
 }
 catch (MalformedURLException e) {
 ...
 }
 catch (UnknownHostException e) {
 ...
 }
}
```

Exception table for `loadPage()`

|  | from_eip | to_eip | exception     | handler_eip |
|--|----------|--------|---------------|-------------|
|  | 0x3019   | 0x315C | malformed_url | 0x316B      |
|  | 0x3019   | 0x315C | unknown_host  | 0x3188      |

## EXCEPTION TABLES (2)

- When an exception is thrown within a function:
- Two important pieces of information!
  - What is the current program-counter?
  - What is the type of the exception that was thrown?
- Exception table for the current function is searched

*Exception table for `loadPage()`*

| from_eip | to_eip | exception     | handler_eip |
|----------|--------|---------------|-------------|
| 0x3019   | 0x315C | malformed_url | 0x316B      |
| 0x3019   | 0x315C | unknown_host  | 0x3188      |

- Try to find a row where the program-counter is in the specified range, also having the same exception type
  - If found, dispatch to the specified exception handler
- If no matching row is found, the current stack frame is cleaned up, and process repeats in parent frame

# NESTED TRY/CATCH EXAMPLE

- Code with nested **try/catch** blocks
- Compiler generates an exception table for each function:

*Exception table for f(x)*

| from_eip | to_eip | exception | handler_eip |
|----------|--------|-----------|-------------|
| 0x2005   | 0x203B | a         | 0x2043      |
| 0x2005   | 0x203B | b         | 0x204C      |

*Exception table for g(x)*

| from_eip | to_eip | exception | handler_eip |
|----------|--------|-----------|-------------|
| 0x2116   | 0x214A | b         | 0x2159      |
| 0x2116   | 0x214A | c         | 0x215E      |

*Exception table for h(x)*

| from_eip | to_eip | exception | handler_eip |
|----------|--------|-----------|-------------|
|          |        |           |             |

```
int f(int x) {
 try {
 return g(x * 3);
 } catch (A a) {
 return -5;
 } catch (B b) {
 return -10;
 }
}
```

```
int g(int x) {
 try {
 return h(8 - x);
 } catch (B b) {
 return -15;
 } catch (C c) {
 return -20;
 }
}
```

```
int h(int x) {
 if (x > 23)
 throw new A();
 else if (x < -15)
 throw new B();

 return x - 1;
}
```

## NESTED TRY/CATCH (2)

- Call **f (-9)**
- **f (-9)** calls **g (-9 \* 3) = g (-27)**
- **g (-27)** calls **h (8 - -27) = h (35)**

- **Important point:**

- So far, no overhead for entering **try**-blocks, or any other aspect of exception handling!
- But, we know that **h (35)** is going to throw...

```
int f(int x) {
 try {
 return g(x * 3);
 } catch (A a) {
 return -5;
 } catch (B b) {
 return -10;
 }
}

int g(int x) {
 try {
 return h(8 - x);
 } catch (B b) {
 return -15;
 } catch (C c) {
 return -20;
 }
}

int h(int x) {
 if (x > 23)
 throw new A();
 else if (x < -15)
 throw new B();

 return x - 1;
}
```

# NESTED TRY/CATCH (3)

- **h(35)** throws exception **A**
- Use our exception tables to direct the exception propagation
- **h(35)** throws **A**. **eip** = **0x226C**.
  - Check exception table for **h**:  
*Exception table for h(x)*

| from_eip | to_eip | exception | handler_eip |
|----------|--------|-----------|-------------|
|----------|--------|-----------|-------------|

    - Nothing matches. (*duh...*)
    - Clean up local stack frame, then return to caller of **h**

```
int f(int x) {
 try {
 return g(x * 3);
 } catch (A a) {
 return -5;
 } catch (B b) {
 return -10;
 }
}
```

```
int g(int x) {
 try {
 return h(8 - x);
 } catch (B b) {
 return -15;
 } catch (C c) {
 return -20;
 }
}
```

```
int h(int x) {
 if (x > 23)
 throw new A();
 else if (x < -15)
 throw new B();

 return x - 1;
}
```

# NESTED TRY/CATCH (4)

- Now inside of **g(-27)**
- g(-27)** throws **A**. **eip = 0x2123**.
  - Check exception table for **g**:

*Exception table for g(x)*

| from_eip | to_eip | exception | handler_eip |
|----------|--------|-----------|-------------|
| 0x2116   | 0x214A | b         | 0x2159      |
| 0x2116   | 0x214A | c         | 0x215E      |

- g** does have entries in its table, but none match combination of exception **A** and **eip = 0x2123**.
- Again, clean up local stack frame, then return to caller of **g**

```
int f(int x) {
 try {
 return g(x * 3);
 } catch (A a) {
 return -5;
 } catch (B b) {
 return -10;
 }
}
```

```
int g(int x) {
 try {
 return h(8 - x);
 } catch (B b) {
 return -15;
 } catch (C c) {
 return -20;
 }
}
```

```
int h(int x) {
 if (x > 23)
 throw new A();
 else if (x < -15)
 throw new B();

 return x - 1;
}
```

# NESTED TRY/CATCH (5)

- Finally, back to **f (-9)**
- f (-9)** throws **A**. **eip = 0x201B**.
  - Check exception table for **f**:

*Exception table for f (x)*

| from_eip | to_eip | exception | handler_eip |
|----------|--------|-----------|-------------|
| 0x2005   | 0x203B | a         | 0x2043      |
| 0x2005   | 0x203B | b         | 0x204C      |

- f** also has exception table entries, and the first entry matches our combination of exception type and instruction-pointer value!
- Dispatch to specified handler:
  - return -5;**
- Exception propagation is complete.

```
int f(int x) {
 try {
 return g(x * 3);
 } catch (A a) {
 return -5;
 } catch (B b) {
 return -10;
 }
}
```

```
int g(int x) {
 try {
 return h(8 - x);
 } catch (B b) {
 return -15;
 } catch (C c) {
 return -20;
 }
}
```

```
int h(int x) {
 if (x > 23)
 throw new A();
 else if (x < -15)
 throw new B();

 return x - 1;
}
```

# COMPARISON OF METHODOLOGIES

- Stack cutting approach is optimized for the exception-handling phase
  - Transfers control to handler code in one step
    - (Presuming resources don't need to be cleaned up from intervening function calls...)
- Additional costs in the normal execution paths!
  - Need to record execution state every time a **try**-block is entered
  - Need to push and pop these state records, too!
- These costs will *quickly* add up in situations where execution passes through many **try**-blocks

## COMPARISON OF METHODOLOGIES (2)

- Stack unwinding approach is optimized for the normal execution phase
  - No exception-handler bookkeeping is needed at run-time...
  - ...because all bookkeeping is done at compile-time!
- Additional costs in the exception-handling paths
  - Each function call on stack is dealt with individually
  - Must search through each function's exception table, performing several comparisons per record
- If a program *frequently* throws exceptions, especially from deep within call-sequences, this will definitely add up

# COMPARISON OF METHODOLOGIES (3)

- Typical assumption is that exception handling is a relatively uncommon occurrence
  - (*That's why we call them exceptions!!!*)
  - Additionally, most languages have resources to clean up, within each function's stack frame
  - e.g. even though Java has garbage collection, it also has **synchronized** blocks; monitors need unlocked
- Most common implementation: stack unwinding
- Many other optimizations are applied to exception handling as well!
  - Dramatically reduce or even eliminate overhead of searching for exception handlers within each function