UNIVERSITY of WASHINGTON

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

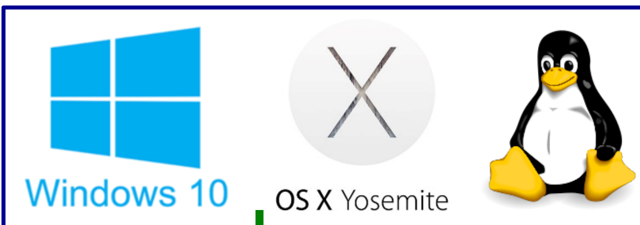
Assembly language:

```
get_mpg(car*):
        lw      a5,0(a0)
        lw      a4,4(a0)
        divw    a5,a5,a4
        fcvt.s.w        fa0,a5
        ret
```

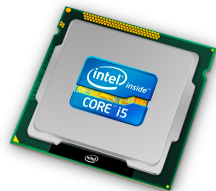
Machine code:

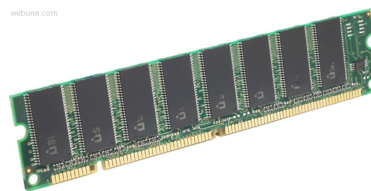
```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:



Computer system:





Memory & data
Arrays & structs
Integers & floats
RISC V assembly
Procedures & stacks
Executables
Memory & caches
Processor Pipeline
Performance
Parallelism

UNIVERSITY of WASHINGTON

From Writing to Running

Compiler gcc -S → Assembler gcc -c → Linker gcc -o

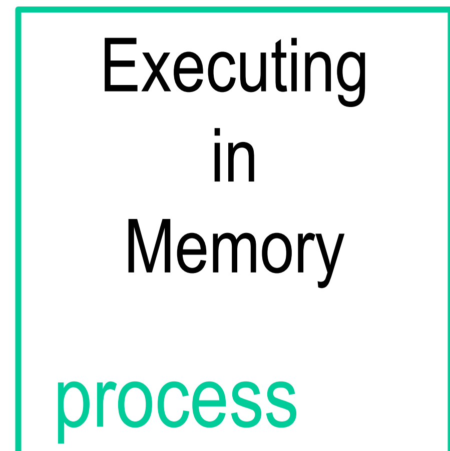
sum.c → sum.s → sum.o → sum

C source files | assembly files | obj files | executable program

sum: exists on disk

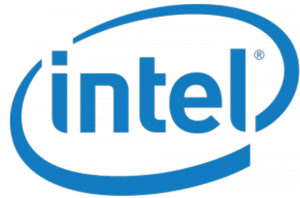
When most people say "compile" they mean the entire process:
compile + assemble + link

"It's alive!"



UNIVERSITY of WASHINGTON

Mainstream ISAs



x86

Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register-memory
Encoding	Variable (1 to 15 bytes)
Endianness	Little

Macbooks & PCs
(Core i3, i5, i7, M)
x86 Instruction Set

ARM architectures

Designer	ARM Holdings
Bits	32-bit, 64-bit
Introduced	1985; 31 years ago
Design	RISC
Type	Register-Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility[1]
Endianness	Bi (little as default)

Smartphone-like devices
(iPhone, iPad, Raspberry Pi)
ARM Instruction Set

RISC-V

Designer	University of California, Berkeley
Bits	32, 64, 128
Introduced	2010
Version	2.2
Design	RISC
Type	Load-store
Encoding	Variable
Branching	Compare-and-branch
Endianness	Little

Versatile and open-source
Relatively new, designed for cloud computing, high-end phones, small embedded sys.
RISCV Instruction Set

UNIVERSITY of WASHINGTON

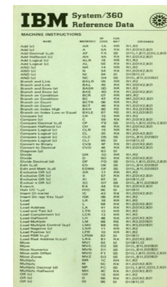
Complex/Reduced Instruction Set Computing

- Early trend: add more and more instructions to do elaborate operations – *Complex Instruction Set Computing* (CISC)
  - difficult to learn and comprehend language
  - super-complicated (slow?) hardware
- Opposite philosophy later began to dominate: *Reduced Instruction Set Computing* (RISC)
  - Simpler (and smaller) instruction set makes it easier to build fast hardware
  - Let software do the complicated operations by composing simpler ones

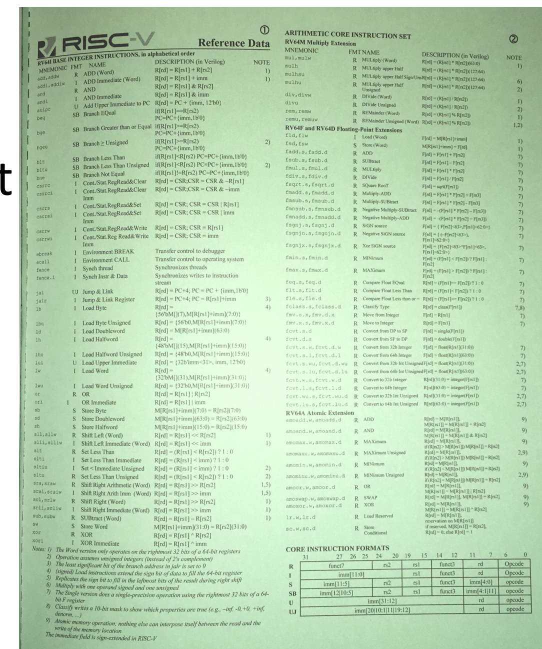
UNIVERSITY of WASHINGTON

RISC-V Architecture

IBM 360 Green Card



- New open-source, license-free ISA spec
  - Supported by growing shared software ecosystem
  - Appropriate for all levels of computing system, from microcontrollers to supercomputers
  - 32-bit, 64-bit, and 128-bit variants (we're using 32-bit in class, textbook uses 64-bit)
- Why RISC-V instead of Intel 80x86?
  - RISC-V is simple, elegant. Don't want to get bogged down in gritty details.
  - RISC-V has exponential adoption rate



RISC-V Green Card

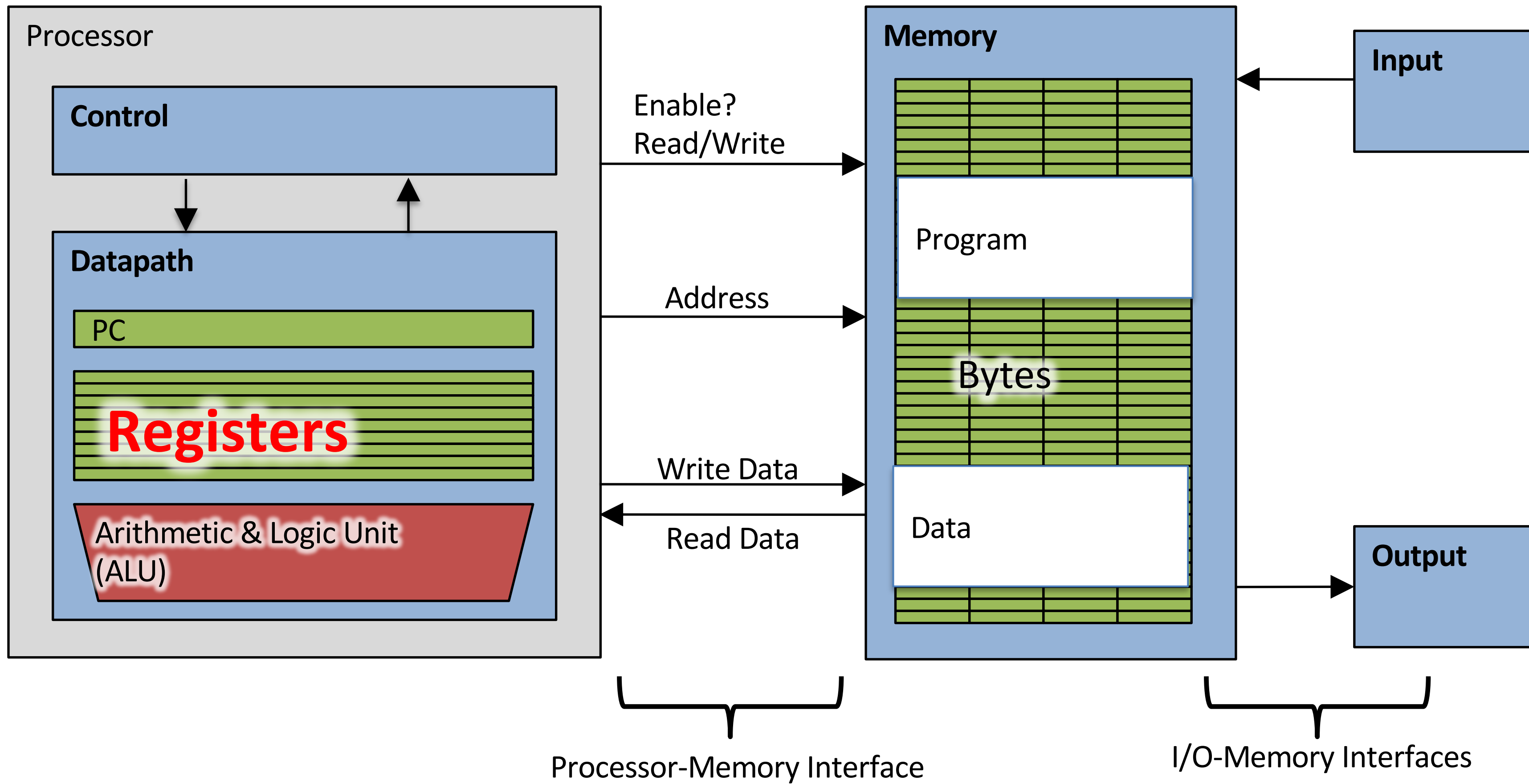
UNIVERSITY of WASHINGTON

Registers -- Summary

- In high-level languages, number of variables limited only by available memory
- ISAs have a fixed, small number of operands called registers
  - Special locations built directly into hardware
  - **Benefit:** Registers are EXTREMELY FAST (faster than 1 billionth of a second)
  - **Drawback:** Operations can only be performed on these predetermined number of registers

UNIVERSITY of WASHINGTON

Aside: Registers are Inside the Processor



UNIVERSITY of WASHINGTON

RISCV -- How Many Registers?

- Tradeoff between speed and availability
  - more registers → can house more variables simultaneously; all registers are slower.
- RISCV has 32 registers (x0-x31)
  - Each register is 32 bits wide and holds a word

REGISTER NAME, USE, CALLING CONVENTION

REGISTER	NAME	USE
x0	zero	The constant value 0
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7	t0-t2	Temporaries
x8	s0/fp	Saved register/Frame pointer
x9	s1	Saved register
x10-x11	a0-a1	Function arguments/Return values
x12-x17	a2-a7	Function arguments
x18-x27	s2-s11	Saved registers
x28-x31	t3-t6	Temporaries
f0-f7	ft0-ft7	FP Temporaries
f8-f9	fs0-fs1	FP Saved registers
f10-f11	fa0-fa1	FP Function arguments/Return valu
f12-f17	fa2-fa7	FP Function arguments
f18-f27	fs2-fs11	FP Saved registers
f28-f31	ft8-ft11	R[rd] = R[rs1] + R[rs2]

UNIVERSITY of WASHINGTON

Memory vs. Registers

Memory	vs.	Registers
Addresses	vs.	Names
`0x7FFFD024C3DC`		`%x0`
Big	vs.	Small
~ 8 GiB		(16 x 8 B) = 128 B
Slow	vs.	Fast
~50-100 ns		sub-nanosecond timescale
Dynamic	vs.	Static
Can “grow” as needed while program runs		fixed number in hardware

UNIVERSITY of WASHINGTON

RISCV Registers

- Register denoted by 'x' can be referenced by number (x0-x31) or name:
  - Registers that hold programmer variables:
    - s0-s1 ↔ x8-x9
    - s2-s11 ↔ x18-x27
  - Registers that hold temporary variables:
    - t0-t2 ↔ x5-x7
    - t3-t6 ↔ x28-x31
  - You'll learn about the other 13 registers later
- *Registers have no type* (C concept); the operation being performed determines how register contents are treated

UNIVERSITY of WASHINGTON

C, Java variables vs. registers

- In C (and most High Level Languages) variables declared first and given a type. E.g.,

```
int fahr, celsius;
char a, b, c, d, e;
```

- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match `int` and `char` variables).

- In Assembly Language, the registers have no type
  - Operation determines how register contents are treated

UNIVERSITY of WASHINGTON

RISCV Agenda

- Basic Arithmetic Instructions
- Comments
- x0 (zero)
- Immediates
- Data Transfer Instructions
- Decision Making Instructions
- Bonus: C to RISCV Practice
- Bonus: Additional Instructions

UNIVERSITY of WASHINGTON

RISCV Instructions (1/2)

- Instruction Syntax is rigid:

```
op dst, src1, src2
```

  - 1 operator, 3 operands
    - `op` = operation name ("operator")
    - `dst` = register getting result ("destination")
    - `src1` = first register for operation ("source 1")
    - `src2` = second register for operation ("source 2")
- Keep hardware simple via regularity

UNIVERSITY of WASHINGTON

RISCV Instructions (2/2)

- One operation per instruction, at most one instruction per line
- Assembly instructions are related to C operations (=, +, -, *, /, &, |, etc.)
  - Must be, since C code decomposes into assembly!
  - A single line of C may break up into several lines of RISCV

UNIVERSITY of WASHINGTON

RISCV Instructions Example

- Your very first instructions!
  (assume here that the variables a, b, and c are assigned to registers s1, s2, and s3, respectively)
- Integer Addition (add)
  - C: a = b + c
  - RISCV: add s1, s2, s3
- Integer Subtraction (sub)
  - C: a = b - c
  - RISCV: sub s1, s2, s3

UNIVERSITY of WASHINGTON

RISCV Instructions Example

- Suppose `a` → `s0`, `b` → `s1`, `c` → `s2`, `d` → `s3` and `e` → `s4`. Convert the following C statement to RISCV:

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

```
add t1, s3, s4
add t2, s1, s2
sub s0, t2, t1
```

Ordering of instructions matters (must follow order of operations)

Utilize temporary registers

UNIVERSITY of WASHINGTON

Assembly Instructions

- In assembly language, each statement (called an Instruction), executes exactly one of a short list of simple commands
- Unlike in C (and most other High Level Languages), each line of assembly code contains at most 1 instruction
- Instructions are related to operations (=, +, -, *, /) in C or Java
- Ok, enough already…gimme my RV32!

UNIVERSITY of WASHINGTON

RISC V Assembly "Data Types"

- Integral data of 1, 2, 4, or 8 bytes **(we focus on 4 bytes)**
  - Data values
  - Addresses
- Floating point data of 4, 8, 10 or 2x8 or 4x4 or 8x2
  - Different registers for those (*e.g.* `%f0`, `%f31`)

Not covered In 351

- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
- "AT&T": used by our course, slides, textbook, gnu tools, …

UNIVERSITY of WASHINGTON

RISC V Integer Registers – 32 bits wide

Register	ABI name	Use
x0	zero	zero
x1	ra	return address
x2	sp	stack pointer
x3	gp	global data pointer
x4	tp	thread pointer
x5	t0	**temps (caller save)**
x6	t1	
x7	t2	
x8	s0/fp	frame pointer
x9	s1	**saved (callee save)**
x10	a0	function args or return values
x11	a1	
x12	a2	function arguments
x13	a3	
x14	a4	

Register	ABI name	Use
x15	a5	function arguments
x16	a6	
x17	a7	
x18	s2	**saved (callee save)**
x19	s3	
x20	s4	
x21	s5	
x22	s6	
x23	s7	
x24	s7	
x25	s9	
x26	s10	
x27	s11	
x28	t3	**temps (caller save)**
x29	t4	
x30	t5	
x31	t6	

UNIVERSITY of WASHINGTON

Three Basic Kinds of Instructions

1) Transfer data between memory and register
   - *Load* data from memory into register
     - `%reg` = Mem[address]
   - *Store* register data into memory
     - Mem[address] = `%reg`

Remember: Memory is indexed just like an array of bytes!

2) Perform arithmetic operation on register or memory data
   - `c = a + b;` `z = x << y;` `i = h & g;`

3) Control flow: what instruction to execute next
   - Unconditional jumps to/from procedures
   - Conditional branches

UNIVERSITY of WASHINGTON

Operand types

- ***Immediate:*** Constant integer data
  - Examples: **`$0x400`**, **`$-533`**
  - Like C literal, but prefixed with **`'$'`**
  - Encoded with 1, 2, 4, or 8 bytes *depending on the instruction*
- ***Register:*** 22 integer registers
  - Examples: **`%x9`** … **`%x31`**
  - But **`%x0-x4 and x8`** reserved for special use
  - Others have special uses for particular instructions
- ***Memory:*** Consecutive bytes of memory at a computed address
  - Simplest example: **`(%x18)`**

UNIVERSITY of WASHINGTON

RISC-V Addition and Subtraction (1/4)

- Syntax of Instructions:
  - One two, three, four
  - where:
  - One = operation by name
  - two = operand getting result ("destination")
  - three = 1st operand for operation ("source1")
  - four = 2nd operand for operation ("source2")
- Syntax is rigid:
  - 1 operator, 3 operands
  - Why? Keep Hardware simple via regularity

```
add x1, x2, x3
```

UNIVERSITY of WASHINGTON

Addition and Subtraction of Integers (2/4)

- Addition in Assembly
  - Example: `add x1,x2,x3` (in RISC-V)
  - Equivalent to: a = b + c (in C)
  - where C variables ⇔ RISC-V registers are:
    a ⇔ `x1`, b ⇔ `x2`, c ⇔ `x3`
- Subtraction in Assembly
  - Example: `sub x3,x4,x5` (in RISC-V)
  - Equivalent to: d = e - f (in C)
  - where C variables ⇔ RISC-V registers are:
    d ⇔ `x3`, e ⇔ `x4`, f ⇔ `x5`

UNIVERSITY of WASHINGTON

Addition and Subtraction of Integers (3/4)

- How to do the following C statement?

  a = b + c + d - e;

- Break into multiple instructions

```
add x10, x1, x2   # a_temp = b + c
add x10, x10, x3  # a_temp = a_temp + d
sub x10, x10, x4  # a = a_temp - e
```

- Notice: A single line of C may break up into several lines of RISC-V.
- Notice: Everything after the hash mark on each line is ignored (comments). Check Apollo-11 comments!

UNIVERSITY of WASHINGTON

Addition and Subtraction of Integers (4/4)

- How do we do this?

  f = (g + h) - (i + j);

- Use intermediate temporary register
  - `add x5, x20, x21 # a_temp = g + h`
  - `add x6, x22, x23 # b_temp = i + j`
  - `sub x19, x5, x6  # f = (g + h)- (i + j)`

UNIVERSITY of WASHINGTON

RISCV Agenda

- Basic Arithmetic Instructions
- Comments
- x0 (zero)
- Immediates
- Data Transfer Instructions
- Decision Making Instructions
- Bonus: C to RISCV Practice
- Bonus: Additional Instructions

UNIVERSITY of WASHINGTON

Comments in Assembly

- Another way to make your code more readable: comments!
- Hash (#) is used for RISC-V comments
  - anything from hash mark to end of line is a comment and will be ignored
  - This is just like the C99 //
- Note: Different from C.
  - C comments have format
    /* comment */
    so they can span many lines

UNIVERSITY of WASHINGTON

The Zero Register

- Zero appears so often in code and is so useful that it has its own register!
- Register zero (`x0` or `zero`) always has the value 0 and cannot be changed!
  - i.e. any instruction with `x0` as `dst` has no effect
- Example Uses:
  - `add s3, x0, x0 # c=0`
  - `add s1, s2, x0 # a=b`

UNIVERSITY of WASHINGTON

"And in Conclusion…"

- In RISC-V Assembly Language:
  - Registers replace C variables
  - One instruction (simple operation) per line
  - Simpler is Better, Smaller is Faster
- In RV32, words are 32b
- Instructions:

  `add`, `addi`, `sub`
- Registers:
  - 32 registers, referred to as `x0` – `x31`
  - Zero: `x0`

UNIVERSITY of WASHINGTON

RISCV Agenda

- Basic Arithmetic Instructions
- Comments
- x0 (zero)
- Immediates
- Data Transfer Instructions
- Decision Making Instructions
- Bonus: C to RISCV Practice
- Bonus: Additional Instructions

UNIVERSITY of WASHINGTON

Immediates

- Immediates are numerical constants.
- They appear often in code, so there are special instructions for them.
- Add Immediate:
  - `addi x3,x4,10` (in RISC-V)
  - f = g + 10 (in C)
  - where RISC-V registers `x3,x4` are associated with C variables f, g
- Syntax similar to add instruction, except that last argument is a number instead of a register.

UNIVERSITY of WASHINGTON

Immediates

- There is no Subtract Immediate in RISC-V: Why?
  - There are add and sub, but no addi counterpart
- Limit types of operations that can be done to absolute minimum
  - if an operation can be decomposed into a simpler operation, don't include it
  - addi …, -X = subi …, X => so no subi

`addi x3,x4,-10` (in RISC-V)

f = g - 10 (in C)

  - where RISC-V registers `x3, x4` are associated with C variables f, g

Immediates

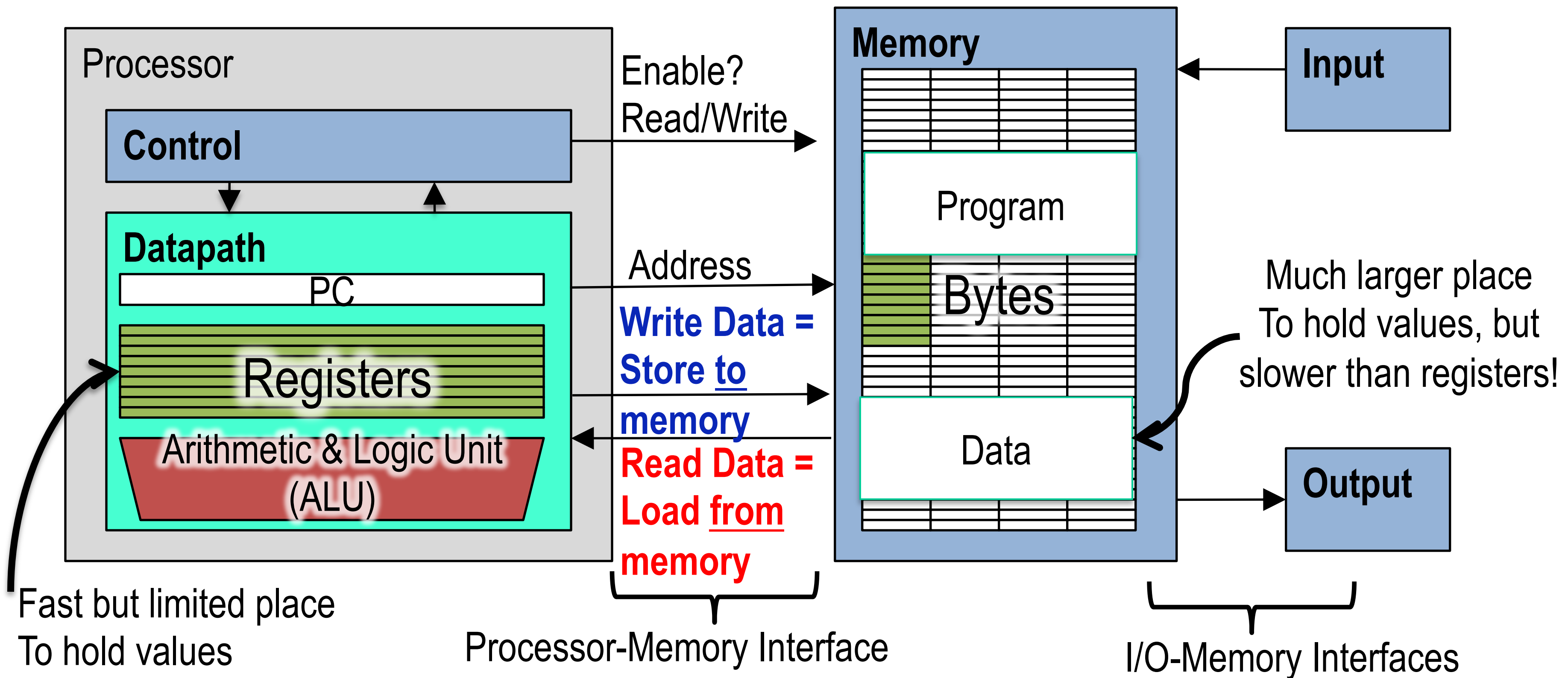
- Numerical constants are called immediates
- Separate instruction syntax for immediates:

```
opi dst, src, imm
```

  - Operation names end with 'i', replace 2nd source register with an immediate
- Example Uses:
  - `addi s1, s2, 5   # a=b+5`
  - `addi s3, s3, 1   # c++`
- Why no `subi` instruction?

UNIVERSITY of WASHINGTON

Data Transfer:
Load from and Store to memory



UNIVERSITY of WASHINGTON

Memory Addresses are in Bytes

- Data typically smaller than 32 bits, but rarely smaller than 8 bits (e.g., char type)–works fine if everything is a multiple of 8 bits
- 8 bit chunk is called a *byte* (1 word = 4 bytes)
- Memory addresses are really in *bytes*, not words
- Word addresses are 4 bytes apart
  - Word address is same as address of rightmost byte – least-significant byte (i.e. **Little-endian** convention)

Least-significant byte in a word

31 24	23 16	15 8	7 0
...	...	...	...
15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	0

Least-significant byte gets the smallest address

UNIVERSITY of WASHINGTON

Big Endian vs. Little Endian

en.wikipedia.org/wiki/Big_endian

- The order in which BYTES are stored in memory
- Bits always stored as usual. (E.g., 0xC2=0b 1100 0010)

Consider the number 1025 as we normally write it:

BYTE2	BYTE1	BYTE0
00000000	00000100	00000001

Big Endian

ADDR3	ADDR2	ADDR1	ADDR0
BYTE0	BYTE1	BYTE2	
00000001	00000100	00000000	

Examples

Names in China (e.g., Garcia Dan)

Java Packages: (e.g., org.mypackage.HelloWorld)

Dates done correctly ISO 8601 YYYY-MM-DD (e.g., 2018-09-07)

Eating Pizza crust first

Unix file structure (e.g., /usr/local/bin/python)

Little Endian

ADDR3	ADDR2	ADDR1	ADDR0
	BYTE2	BYTE1	BYTE0
	00000000	00000100	00000001

Examples

Names in the US (e.g., Dan Garcia)

Internet names (e.g., www.cs.berkeley.edu)

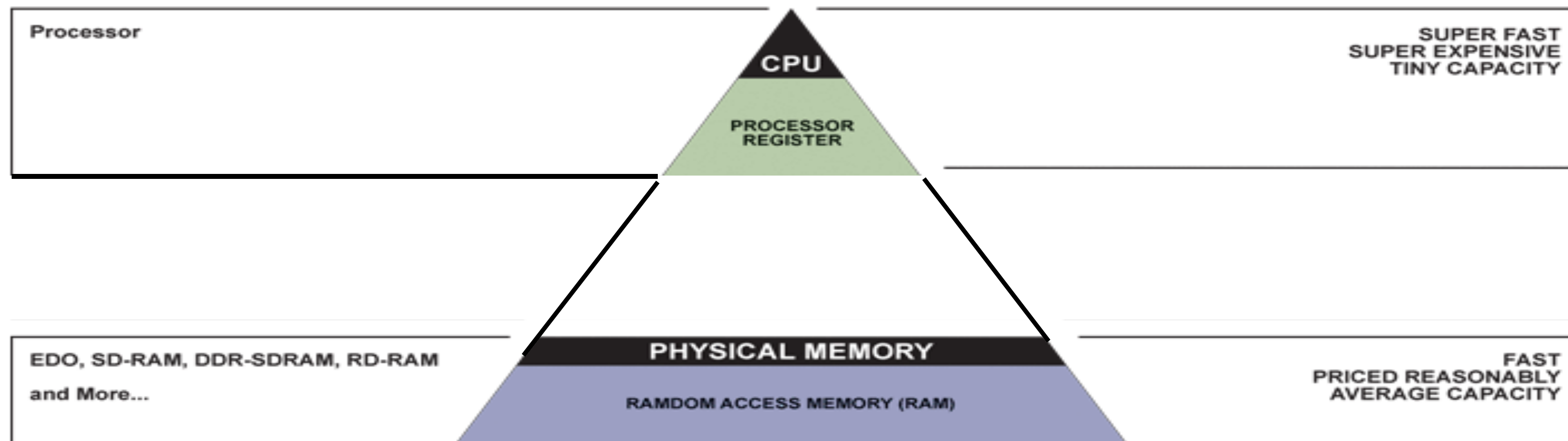
Dates written in England DD/MM/YYYY (e.g., 07/09/2018)

Eating Pizza skinny part first (the normal way)

Big-endian and little-endian derive from Jonathan Swift's *Gulliver's Travels* in which the Big Endians were a political faction that broke their eggs at the large end ("the primitive way") and rebelled against the Lilliputian King who required his subjects (the Little Endians) to break their eggs at the small end.

UNIVERSITY of WASHINGTON

Great Idea #3: Principle of Locality / Memory Hierarchy



UNIVERSITY of WASHINGTON

Speed of Registers vs. Memory

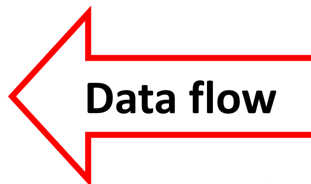
- Given that
  - Registers: 32 words (128 Bytes)
  - Memory (DRAM): Billions of bytes (2 GB to 64 GB on laptop)
- and physics dictates…
  - Smaller is faster
- How much faster are registers than DRAM??
  - About 100-500 times faster! (in terms of *latency* of one access)

UNIVERSITY of WASHINGTON

Load from Memory to Register

- C code

```
int  A[100];
g = h + A[3];
```



- Using Load Word (lw) in RISC-V:

```
lw   x10,12(x15)  # Reg x10 gets A[3]
add x11,x12,x10  # g = h + A[3]
```

Note: x15 – base register (pointer to A[0])
12 – offset in bytes

Offset must be a constant known at assembly time

UNIVERSITY of WASHINGTON

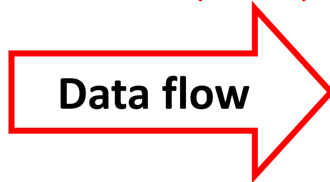
Store from Register to Memory

- C code

```
int  A[100];
A[10] = h + A[3];
```

- Using Store Word (sw) in RISC-V:

```
lw   x10,12(x15)    # Temp reg x10 gets A[3]
add  x10,x12,x10    # Temp reg x10 gets h + A[3]
sw    x10,40(x15)    # A[10] = h + A[3]
```



Note: x15 – base register (pointer)

12,40 – offsets in bytes

x15+12 and x15+40 must be multiples of 4

UNIVERSITY of WASHINGTON

Loading and Storing Bytes

- In addition to word data transfers (**`lw`**, **`sw`**), RISC-V has byte data transfers:
  - load byte: **`lb`**
  - store byte: **`sb`**
- Same format as **`lw`**, **`sw`**
- E.g., **`lb x10,3(x11)`**
  - contents of memory location with address = sum of "3" + contents of register x11 is copied to the low byte position of register x10.

RISC-V also has "unsigned byte" loads (**`lbu`**) which zero extends to fill register. Why no unsigned store byte **`sbu`**?

UNIVERSITY of WASHINGTON

RISCV Agenda

- Basic Arithmetic Instructions
- Comments
- x0 (zero)
- Immediates
- Data Transfer Instructions
- Decision Making Instructions
- Bonus: C to RISCV Practice
- Bonus: Additional Instructions

UNIVERSITY of WASHINGTON

Decision Making Instructions

- Branch If Equal (`beq`)
  - `beq reg1,reg2,label`
  - If value in `reg1` = value in `reg2`, go to `label`
- Branch If Not Equal (`bne`)
  - `bne reg1,reg2,label`
  - If value in `reg1` ≠ value in `reg2`, go to `label`
- Jump (`j`)
  - `j label`
  - Unconditional jump to `label`

UNIVERSITY of WASHINGTON

Types of Branches

- **Branch** – change of control flow
- **Conditional Branch** – change control flow depending on outcome of comparison
  - branch *if* equal (**`beq`**) or branch *if not* equal (**`bne`**)
  - Also branch if less than (**`blt`**) and branch if greater than or equal (**`bge`**)
- **Unconditional Branch** – always branch
  - a RISC-V instruction for this: *jump* (**`j`**), as in **`j label`**

UNIVERSITY of WASHINGTON

Breaking Down the If Else

C Code:

```
if(i==j) {
   a = b   /* then */
} else {
   a = -b /* else */
}
```

In English:

- If TRUE, execute the THEN block
- If FALSE, execute the ELSE block

RISCV (beq):

```
# i→s0, j→s1
# a→s2, b→s3

beq s0,s1,???
???
sub s2, x0, s3
j   end
then:
add s2, s3, x0
end:
```

This label unnecessary (points to the `???` label line)

UNIVERSITY of WASHINGTON

Breaking Down the If Else

C Code:

```
if(i==j) {
   a = b  /* then */
} else {
   a = -b /* else */
}
```

In English:

- If TRUE, execute the THEN block
- If FALSE, execute the ELSE block

RISCV (bne):

```
# i→s0, j→s1
# a→s2, b→s3
bne s0,s1,???
???
add s2, s3, x0
j   end
else:
sub s2, x0, s3
end:
```

UNIVERSITY of WASHINGTON

Branching on Conditions other than (Not) Equal

- Set Less Than (slt)
  - `slt dst, reg1,reg2`
  - If value in `reg1` < value in `reg2`, `dst` = 1, else 0
- Set Less Than Immediate (slti)
  - `slti dst, reg1,imm`
  - If value in `reg1` < `imm`, `dst` = 1, else 0

UNIVERSITY of WASHINGTON

Breaking Down the If Else

C Code:

```
if(i<j) {
   a = b   /* then */
} else {
   a = -b /* else */
}
```

In English:

- If TRUE, execute the THEN block
- If FALSE, execute the ELSE block

RISCV (???):

```
# i→s0, j→s1
# a→s2, b→s3

slt t0 s0 s1
??? t0,??? else
then:
add s2, s3, x0
j    end
else:
sub s2, x0, s3
end:
```

UNIVERSITY of WASHINGTON

Branching on Conditions other than (Not) Equal

- Branch Less Than (blt)
  - `blt reg1,reg2, label`
  - If value in `reg1` < value in `reg2`, go to `label`
- Branch Greater Than or Equal (bge)
  - `bge reg1,reg2, label`
  - If value in `reg1` >= value in `reg2`, go to `label`

UNIVERSITY of WASHINGTON

Breaking Down the If Else

C Code:

```
if(i<j) {
  a = b  /* then */
} else {
  a = -b /* else */
}
```

In English:

- If TRUE, execute the THEN block
- If FALSE, execute the ELSE block

RISCV (???):

```
# i→s0, j→s1
# a→s2, b→s3

??? s0,s1,else
then:
add s2, s3, x0
j   end
else:
sub s2, x0, s3
end:
```

UNIVERSITY of WASHINGTON

Example *if* Statement

- Assuming translations below, compile *if* block

  f → `x10` g → `x11` h → `x12`

  i → `x13` j → `x14`

```
if (i == j)                                   bne x13,x14,Exit
   f = g + h;                                 add x10,x11,x12
                                              Exit:
```

- May need to negate branch condition

UNIVERSITY of WASHINGTON

Example *if-else* Statement

- Assuming translations below, compile

  f → x10 g → x11 h → x12

  i → x13 j → x14

```
if (i == j)
  f = g + h;
else
  f = g - h;
```

```
bne x13,x14,Else
add x10,x11,x12
j Exit
Else:
sub x10,x11,x12
Exit:
```

UNIVERSITY of WASHINGTON

Magnitude Compares in RISC-V

- Until now, we've only tested equalities (== and != in C); General programs need to test < and > as well.
- RISC-V magnitude-compare branches:

  "Branch on Less Than"

  Syntax: **blt reg1,reg2, label**

  Meaning: if (reg1 < reg2) // treat registers as signed integers
  goto label;

- "Branch on Less Than Unsigned"

  Syntax: **bltu reg1,reg2, label**

  Meaning: if (reg1 < reg2) // treat registers as unsigned integers
  goto
  label;

UNIVERSITY of WASHINGTON

C Loop Mapped to RISC-V Assembly

```
int A[20];
int sum = 0;
for (int i=0; i < 20; i++)
    sum +=  A[i];
```

```
   add x9, x8, x0   # x9=&A[0]
   add x10, x0, x0  # sum=0
   add x11, x0, x0  # i=0
   addi x13,x0, 20  # x13=20
Loop:
   bge x11,x13,Done
   lw x12, 0(x9)    # x12=A[i]
   add x10,x10,x12  # sum+=
   addi x9, x9,4    # &A[i+1]
   addi x11,x11,1   # i++
   j Loop
Done:
```

UNIVERSITY of WASHINGTON

"And in Conclusion…"

- Memory is byte-addressable, but lw and sw access one word at a time.
- A pointer (used by lw and sw) is just a memory address, we can add to it or subtract from it (using offset).
- A Decision allows us to decide what to execute at run-time rather than compile-time.
- C Decisions are made using conditional statements within `if`, `while`, `do while`, `for`.
- RISC-V Decision making instructions are the conditional branches: `beq` and `bne`.
- New Instructions:

`lw, sw, lb, sb, lbu, beq, bne, blt, bltu, bge, j`

UNIVERSITY of WASHINGTON

BONUS SLIDES

You are responsible for the material contained on the following slides, though we may not have enough time to get to them in lecture.

You may learn the material just by doing other coursework, but hopefully these slides will help clarify the material.

UNIVERSITY of WASHINGTON

C to RISCV Practice

- Let’s put our all of our new RISCV knowledge to use in an example: “Fast String Copy”
- C code is as follows:

```
/* Copy string from p to q */
char *p, *q;
while((*q++ = *p++) != '\0') ;
```

- What do we know about its structure?
  - Single `while` loop
  - Exit condition is an equality test

UNIVERSITY of WASHINGTON

C to RISCV Practice

- Start with code skeleton:

```
# copy String p to q
# p→s0, q→s1 (pointers)
Loop:                        # t0 = *p
                             # *q = t0
                             # p = p + 1
                             # q = q + 1
                             # if *p==0, go to Exit
     j Loop                  # go to Loop
Exit:
```

UNIVERSITY of WASHINGTON

C to RISCV Practice

- Fill in lines:

```
# copy String p to q
# p→s0, q→s1 (pointers)
Loop: lb   t0,0(s0)      # t0 = *p
      sb   t0,0(s1)      # *q = t0
      addi s0,s0,1    # p = p + 1
      addi s1,s1,1       # q = q + 1
      beq  t0,0,Exit     # if *p==0, go to Exit
      j Loop             # go to Loop
Exit:
```

UNIVERSITY of WASHINGTON

C to **RISCV** Practice

- Finished code:

```
# copy String p to q
# p→$s0, q→$s1 (pointers)
Loop: lb   t0,0(s0)     # t0 = *p
      sb   t0,0(s1)     # *q = t0
      addi s0,s0,1      # p = p + 1
      addi s1,s1,1      # q = q + 1
      beq  t0,x0,Exit   # if *p==0, go to Exit
      j Loop            # go to Loop
Exit: # N chars in p => N*6 instructions
```

UNIVERSITY of WASHINGTON

C to **RISCV** Practice

- Alternate code using bne:

```
# copy String p to q
# p→s0, q→s1 (pointers)
Loop: lb   t0,0(s0)    # t0 = *p
      sb   t0,0(s1)    # *q = t0
      addi s0,s0,1     # p = p + 1
      addi s1,s1,1     # q = q + 1
      bne  t0,x0,Loop  # if *p!=0, go to Loop
# N chars in p => N*5 instructions
```

UNIVERSITY of WASHINGTON

RISCV Arithmetic Instructions

- Multiplication (`mul and mulh`)
  - `mul  dst, src1, src2`
  - `mulh dst, src1, src2`
  - `src1*src2`: lower 32-bits through `mul`, upper 32-bits in `mulh`
- Division (`div`)
  - `div  dst, src1, src2`
  - `rem  dst, src1, src2`
  - `src1/src2`: quotient via `div`, remainder via `rem`

UNIVERSITY of WASHINGTON

RISCV Bitwise Instructions

Note: a→s1, b→s2, c→s3

Instruction	C	RISCV
And	`a = b & c;`	`and  s1,s2,s3`
And Immediate	`a = b & 0x1;`	`andi s1,s2,0x1`
Or	`a = b | c;`	`or   s1,s2,s3`
Or Immediate	`a = b | 0x5;`	`ori  s1,s2,0x5`
Exclusive Or	`a = b ^ c;`	`xor  s1,s2,s3`
Exclusive Or Immediate	`a = b ^ 0xF;`	`xori s1,s2,0xF`

UNIVERSITY of WASHINGTON

Shifting Instructions

- In binary, shifting an unsigned number left is the same as multiplying by the corresponding power of 2
  - Shifting operations are faster
  - Does not work with shifting right/division
- *Logical shift*: Add zeros as you shift
- *Arithmetic shift*: Sign-extend as you shift
  - Only applies when you shift right (preserves sign)
- Shift by immediate or value in a register

UNIVERSITY of WASHINGTON

Shifting Instructions

Instruction Name	RISCV
Shift Left Logical	`sll s1,s2,s3`
Shift Left Logical Imm	`slli s1,s2,imm`
Shift Right Logical	`srl s1,s2,s3`
Shift Right Logical Imm	`srli s1,s2,imm`
Shift Right Arithmetic	`sra s1,s2,s3`
Shift Right Arithmetic Imm	`srai s1,s2,imm`

- When using immediate, only values 0-31 are practical
- When using variable, only lowest 5 bits are used (read as unsigned)

W UNIVERSITY of WASHINGTON

Shifting Instructions

```
# sample calls to shift instructions
addi  t0,x0 ,-256 # t0=0xFFFFFF00
slli  s0,t0,3     # s0=0xFFFFF800
srli  s1,t0,8     # s1=0x00FFFFFF
srai  s2,t0,8     # s2=0xFFFFFFFF

addi t1,x0 ,-22   # t1=0xFFFFFFEA
                  # low 5: 0b01010
sll s3,t0,t1      # s3=0xFFFC0000
# same as slli s3,t0,10
```

UNIVERSITY of WASHINGTON

Shifting Instructions

- Example 1:

```
# lb using lw:  lb s1,1(s0)
lw      s1,0(s0)   # get word
andi    s1,s1,0xFF00 # get 2nd byte
srli    s1,s1,8      # shift into lowest
```

UNIVERSITY of WASHINGTON

Shifting Instructions

- Example 2:

```
# sb using sw:  sb s1,3(s0)
lw     t0,0(s0)   # get current word
andi   t0,t0,0xFFFFFF # zero top byte
slli   t1,s1,24   # shift into highest
or     t0,t0,t1   # combine
sw     t0,0(s0)   # store back
```

UNIVERSITY of WASHINGTON

Shifting Instructions

- Extra for Experts:
  - Rewrite the two preceding examples to be more general
  - Assume that the byte offset (e.g. 1 and 3 in the examples, respectively) is contained in `s2`
- Hint:
  - The variable shift instructions will come in handy
  - Remember, the offset can be negative

```c
long switch_ex
    (long x, long y, long z)
{
    long w = 1;
    switch (x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
        /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

Switch Statement Example

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4
- Implemented with:
  - *Jump table*
  - *Indirect jump instruction*

UNIVERSITY of WASHINGTON

Jump Table Structure

Switch Form

```
switch (x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table

JTab:

Targ0
Targ1
Targ2
• • •
Targn-1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

• • •

Targn-1: Code Block n–1

Approximate Translation

```
target = JTab[x];
goto target;
```

UNIVERSITY of WASHINGTON

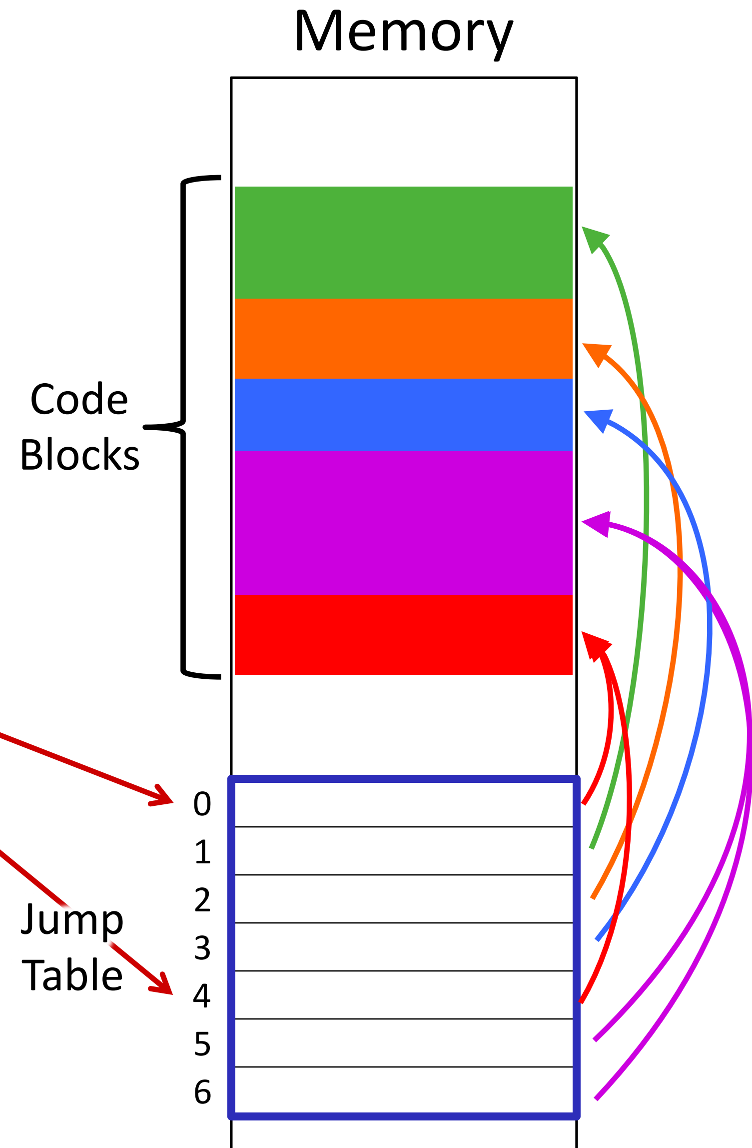
Jump Table Structure

C code:

```
switch (x) {
  case 1: <some code>
          break;
  case 2: <some code>
  case 3: <some code>
          break;
  case 5:
  case 6: <some code>
          break;
  default: <some code>
}
```

Use the jump table when x ≤ 6:

```
if (x <= 6)
  target = JTab[x];
  goto target;
else
  goto default;
```



UNIVERSITY of WASHINGTON

Jump Table

declaring data, not instructions

8-byte memory alignment

Jump table

```
.section .rodata
  .align 8
.L4:
  .quad  .L8 # x = 0
  .quad  .L3 # x = 1
  .quad  .L5 # x = 2
  .quad  .L9 # x = 3
  .quad  .L8 # x = 4
  .quad  .L7 # x = 5
  .quad  .L7 # x = 6
```

this data is 64-bits wide

```
switch(x) {
case 1:         // .L3
    w = y*z;
    break;
case 2:         // .L5
    w = y/z;
    /* Fall Through */
case 3:         // .L9
    w += z;
    break;
case 5:
case 6:         // .L7
    w -= z;
    break;
default:        // .L8
    w = 2;
}
```

UNIVERSITY of WASHINGTON

Handling Fall-Through

```
long w = 1;
   . . .
switch (x) {
   . . .
  case 2:   // .L5
    w = y/z;
  /* Fall Through */
  case 3:   // .L9
    w += z;
    break;
   . . .
}
```

```
case 2:
      w = y/z;
      goto merge;
```

```
case 3:
      w = 1;
```

```
merge:
      w += z;
```

More complicated choice than "just fall-through" forced by "migration" of `w = 1;`

- *Example compilation trade-off*