CSCI 341 Computer Organization
– Chapter 2: RISC-V Instruction Set Architecture

I speak Spanish to God, Italian to women, French to men, and German to my horse.

Charles V, King of France

Topics for Chapter 2

- **Part 1**
  - RISC-V Instruction Set (Chapter 2.1)
  - Arithmetic instructions (Chapter 2.2)
  - Introduction to RARS (Canvas resources)
  - Memory access instructions (Chapters 2.3)
  - Bitwise instructions (Chapter 2.6)
  - Decision making instructions (Chapter 2.7)
  - Arrays vs. pointers (Chapter 2.14)
- **Part 2:**
  - Procedure Calls (Chapters 2.8)
- **Part 3:**
  - RISC-V Instruction Format (Chapter 2.5, RISC-V reference card)
  - RISC-V Addressing Modes (Chapter 2.10)

Assembly Language

- **Assembly language vs. higher-level language**
  - Few, simple types of data and control
  - Does not specify variable type
  - Control flow is implemented with **goto/jump**
  - Assembly language programming
    - more difficult and error-prone,
    - machine-specific
    - longer

- **Assembly language vs. machine language**
  - Symbolic representation

- **When is assembly language needed**
  - Speed and size, (eg. Embedded computer)
  - Time-critical parts of a program
  - Specialized instructions

Instructions

- **Primitive operations that the CPU may execute**
- **Different CPUs implement different sets of instructions.**
- ***Instruction Set Architecture* (*ISA*):**
  - The set of instructions a particular CPU implements
  - Examples:
    - Intel 80x86 (Pentium 4)
    - IBM/Motorola PowerPC (Macintosh)
    - RISC-V
    - Intel IA64
    - ...

Instruction Set Architectures

- **CISC: Complex Instruction Set Computer (eg. 80x86)**
  - Instructions are of variable length and may be very complex
  - Instructions typically include string processing operations, complex record operations, etc.
  - Designed to ease of assembly language programming
  - Many memory-to-register/register-to-register operations
- **RISC: Reduced Instruction Set Computer (e.g., RISC-V, ARM, RISC-V)**
  - Cocke IBM, Patterson, Hennessy, 1980s
  - Keep the instruction set small and simple, making it easier to build faster hardware.
  - Let software do complicated operations by composing simpler ones.
  - Our focus: Open source RISC-V (https://riscv.org/)

Assembly Operands: Registers

- **Unlike HLL such as C or Java, assembly cannot use variables**
  - Keep Hardware Simple
- **Assembly Operands are <u>registers</u>**
  - Limited number of storage cells
    - RISC-V code must efficiently use registers
  - Built directly inside the processor
    - Directly accessible through machine instructions
    - Faster than memory

RISC-V Registers

- **32 registers**
  - Why 32? Smaller is faster
  - Can be referred to by number or name
    - Number references: `x0, x1, x2, … x30, x31`
    - Name references: make code more readable

      `x18 - x27` → `s0 - s11`

      (correspond to C variables, s for “saved”)

      `x28 – x31` → `t3 – t6`

      (correspond to intermediate results, t for “temporary”)
- **Each RISC-V register is 32 bits wide**
  - Groups of 32 bits called a word

RISC-V Registers

REGISTER	**NAME**	**USE**
x0	zero	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

Use names for registers -- code is clearer!

C Variables vs. Registers

- **In C (and most High Level Languages) variables declared first and given a type**
  - Example:

    ```
    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**

Comments in Assembly

- **Hash (#) is used for RISC-V comments**
  - anything from hash mark to end of line is a comment and will be ignored
- **Note: Different from C.**
  - C comments have format
    `/* comment */`
    so they can span many lines

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

Topics for Chapter 2

- **Part 1**
  - RISC-V Instruction Set (Chapter 2.1)
  - ➔ Arithmetic instructions (Chapter 2.2)
  - Introduction to RARS
  - Memory access instructions (Chapters 2.3)
  - Bitwise instructions (Chapter 2.6)
  - Decision making instructions (Chapter 2.7)
  - Arrays vs. pointers (Chapter 2.14)
- **Part 2:**
  - Procedure Calls (Chapters 2.8)
- **Part 3:**
  - RISC-V Instruction Format (Chapter 2.5)
  - RISC-V Addressing Modes (Chapter 2.10)

RISC-V Addition and Subtraction

- **Syntax of Instructions:**

  1 2,3,4

  where:

  1) operation by name

  2) operand getting result ("destination")

  3) 1st operand for operation ("source1")

  4) 2nd operand for operation ("source2")

- **Syntax is rigid:**
  - 1 operator, 3 operands
  - Why? Keep Hardware simple via regularity

RISC-V Addition and Subtraction

Addition in Assembly

- Example: `add s0,s1,s2` (in RISC-V)

  Equivalent to: `a = b + c` (in C)

  where RISC-V registers `s0,s1,s2` are associated with C variables `a, b, c`

Subtraction in Assembly

- Example: `sub s3,s4,s5` (in RISC-V)

  Equivalent to: `d = e - f` (in C)

  where RISC-V registers `s3,s4,s5` are associated with C variables `d, e, f`

RISC-V Addition and Subtraction: Example 1

- **How to do the following C statement?**

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

- **Break into multiple instructions**

```
add t0, s1, s2 # temp = b + c
add t0, t0, s3 # temp = temp + d
sub s0, t0, s4 # a = temp - e
```

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

RISC-V Addition and Subtraction: Example 2

- How to do this?

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

- Use intermediate temporary register

```
add t0,s1,s2 # temp = g + h
add t1,s3,s4 # temp = i + j
sub s0,t0,t1 # f=(g+h)-(i+j)
```

Register Zero

- **Register zero `(x0 or zero)` always has the value 0**
- **Example:**

  `add s0,s1,zero` (in RISC-V)

  `f = g` (in C)

  where RISC-V registers `s0,s1` are associated with C variables `f, g`
- **Defined in hardware, so an instruction**

  `add zero,zero,s0`

  will not do anything!

Immediates

- **Numerical constants.**
- **Appear often in code, so there are special instructions for them.**
- **Add Immediate:**

  `addi s0,s1,10` (in RISC-V)

  `f = g + 10` (in C)

  where RISC-V registers `s0,s1` are associated with C variables `f, g`
- **Syntax similar to `add` instruction, except that last argument is a number instead of a register.**
- **Limited to 12-bit binary values**

Immediates

- **No Subtract Immediate in RISC-V**
  - 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 s0,s1,-10` (in RISC-V)**

  `f = g - 10` (in C)

  where RISC-V registers `s0,s1` are associated with C variables `f, g`

RISC-V Reference Card

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
`add`	ADD	R	`0110011`	`0x0`	`0x00`	`rd = rs1 + rs2`	
`sub`	SUB	R	`0110011`	`0x0`	`0x20`	`rd = rs1 - rs2`	

`addi`	ADD Immediate	I	`0010011`	`0x0`		`rd = rs1 + imm`	

Topics for Chapter 2

- **Part 1**
  - RISC-V Instruction Set (Chapter 2.1)
  - Arithmetic instructions (Chapter 2.2)
  - → Introduction to RARS (Canvas)
  - Memory access instructions (Chapters 2.3)
  - Bitwise instructions (Chapter 2.6)
  - Decision making instructions (Chapter 2.7)
  - Arrays vs. pointers (Chapter 2.14)
- **Part 2:**
  - Procedure Calls (Chapters 2.8)
- **Part 3:**
  - RISC-V Instruction Format (Chapter 2.5)
  - RISC-V Addressing Modes (Chapter 2.10)

RISC-V Simulators

- **Software simulator for processors that implement RISC-V architecture**
  - Read and execute assembly language files
- **RARS (we will use)**
  - a lightweight IDE for programming in RISC-V assembly language
  - Check Canvas for RARS setup instructions
- **Highlights**
  - Assembler directives (pseudo operations)
  - System calls
  - Pseudo instructions and enhanced instructions
    - Turn this off in settings menu to check your work
    - Not all pseudo instructions are allowed in this class (la unavoidable)

Assembler Syntax

- Program includes .data and .text
- Identifier names are sequence of letters, numbers, underbars (_) and dots (.).
- Labels are declared by putting them at beginning of line followed by a colon. Use labels for variables and code locations.
- Operands may be literal values or registers.
- Register is hardware primitive, can stored 32-bit value: s0
- Numbers are base 10 by default. 0x prefix indicates hexadecimal.
- Strings are enclosed in quotes. May include \n=newline or \t=tab. Used for prompts.

Assembler Directives

- .data <addr>
- .text <addr>
- .globl sym
- .ascii “str”: produces string NOT null terminated
- .asciz “str”: produces null-terminated string
- .byte b1, … , bn
- .double d1, …, dn
- .float f1, … , fn
- .word w1, …., wn
- .space n (in bytes)

System Calls

Service	System call code	Arguments	Result
print_int	1	a0 = integer	
print_float	2	fa0 = float	
print_double	3	fa0 = double	
print_string	4	a0 = address of string	
read_int	5		integer in a0
read_float	6		float in fa0
read_double	7		double in fa0
read_string	8	a0=address of buffer, a1=length of buffer	
exit	10		

See RARS Help for a full list of system calls

Using System Calls

- **Load the system call code into a7**
- **Load arguments into registers if needed**
- **Results will be in a0 if returning values**
- **e.g. print the integer in s0**

```
addi a7, zero, 1
add a0, s0, zero
ecall
```

Steps to Follow

- Open new file (star on paper icon)
- Enter your RISC-V code
- Save the file (use .s as file extension)
- Assemble (wrench)
- Run (green arrow)

Sample Program: Input/Output

```
.data

.text

main:
        addi a7, zero, 5            # get an integer from user
        ecall                       # integer in a0

        addi a7, zero, 1            # display integer in decimal
        ecall
```

Sample Program: Display Prompts

```
.data
        prompt: .asciz "Enter a number: “
        output: .asciz "The number you entered was: "
.text 0x00400000
.globl main
main:
        addi a7, zero, 4                # display prompt
        la a0, prompt
        ecall

        addi a7, zero, 5                # get number from user
        ecall
        add s0, a0, zero                # store 1st in s0

        addi a7, zero, 4                # display output
        la a0, output
        ecall

        add a0, s0, zero                 # put number in a0
        addi a7, zero, 1                 # and display it
        ecall

        addi a7, zero, 10               #code to exit cleanly
        ecall
```

Hello World (using .asciz)

```
.data
        hello_msg: .asciz "Hello World\n”

.text
main:
         la a0, hello_msg # load the addr of hello_msg into a0
         addi a7, zero, 4      # 4 is the print_string syscall.
         ecall            # do the syscall.

         addi a7, zero,10      # 10 is the exit syscall.
         ecall            # do the syscall.
```

Hello World (using .ascii)

Another way to declare the string “Hello World\n” and get the exact same output

```
.data
    hello_msg: .ascii "Hello"          # The word "Hello"
    .ascii " "                         # the space.
    .ascii "World"                     # The word "World"
    .ascii "\n"                        # A newline.
    .byte 0                            # a 0 byte.
```

Hello World (using Bytes)

Yet another way to declare the string “Hello World\n” and get the exact same output

.data

```
    hello_msg: .byte 0x48 # hex for ASCII "H"
    .byte 0x65 # hex for ASCII "e"
    .byte 0x6C # hex for ASCII "l"
    .byte 0x6C # hex for ASCII "l"
    .byte 0x6F # hex for ASCII "o"
    ... # and so on...
    .byte 0xA # hex for ASCII newline
    .byte 0x0 # hex for ASCII NUL
```

Live Coding Time

- **Download code from Canvas**
  - io.s:
    - Did you get an error message after execution? How to fix it?
    - Can you modify the code to display the number in hex?
  - prompt.s:
    - understand how to display prompts
  - HelloWorld.s
    - try the other two versions and compare the output
- **Run in RARS**
  - RARS help menu has lots of useful information

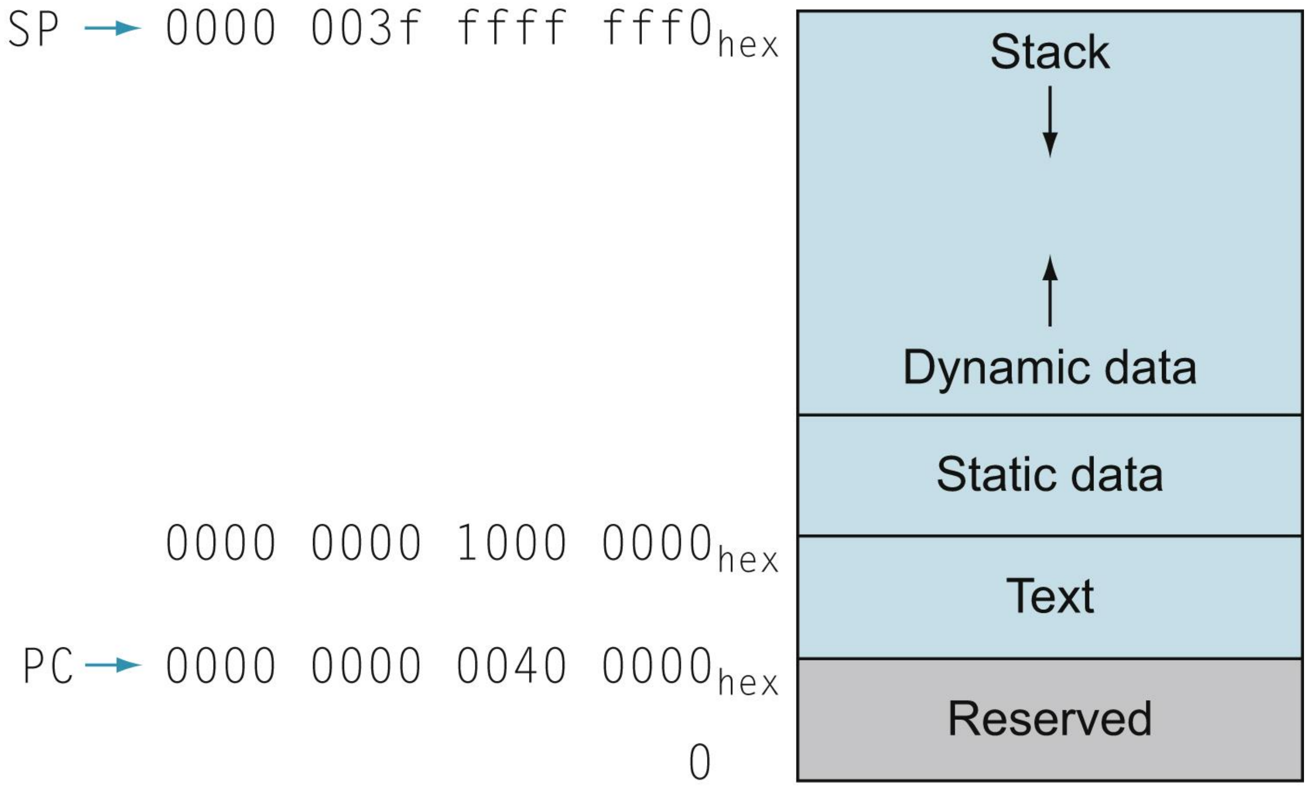
Topics for Chapter 2

- **Part 1**
  - RISC-V Instruction Set (Chapter 2.1)
  - Arithmetic instructions (Chapter 2.2)
  - Introduction to RARS
  - → Memory access instructions (Chapters 2.3, 2.9)
  - Bitwise instructions (Chapter 2.6)
  - Decision making instructions (Chapter 2.7)
  - Arrays vs. pointers (Chapter 2.14)
- **Part 2:**
  - Procedure Calls (Chapters 2.8)
- **Part 3:**
  - RISC-V Instruction Format (Chapter 2.5)
  - RISC-V Addressing Modes (Chapter 2.10)

Role of Registers vs. Memory

- **Which is faster to access?**
- **Which do we have more of?**
- **What to keep in register?**
- **What if more variables than registers?**
  - Compiler tries to keep most frequently used variable in registers
  - Less common in memory: spilling
- **Why not keep all variables in memory?**
  - Smaller is faster: registers are faster than memory

RISC-V Memory Allocation

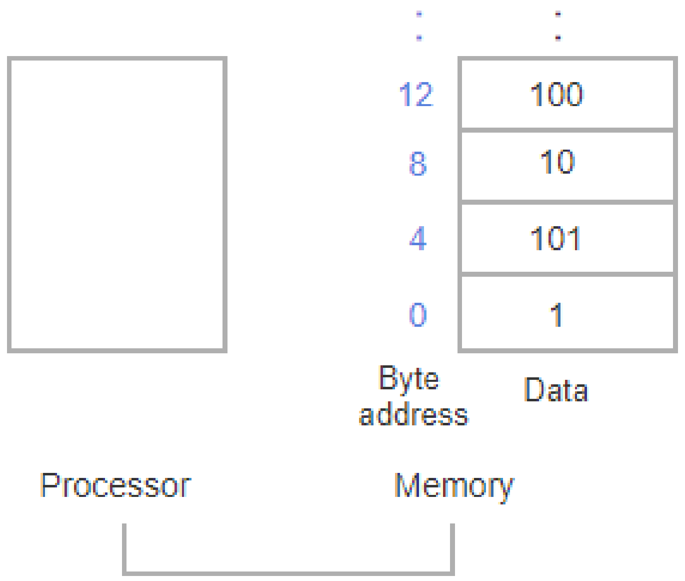


- **Static: Variables declared once per program, cease to exist only after execution completes. e.g., C globals, arrays**
- **Heap: Variables declared dynamically (malloc)**
- **Stack: Space to be used by procedure during execution; this is where we can save register values**

Memory Addressing

- **Every word in memory has an address, similar to an index in an array**
- **Each address is a 32-bit word => 4GB of memory**
- **Byte Addressed**
  - Computers need to access 8-bit bytes as well as words (4 bytes/word)
  - ⇒ address memory as bytes
  - ⇒32-bit (4 byte) word addresses differ by 4
  - ⇒ `Memory[0],Memory[4],Memory[8], …`
- **sequential word addresses in machines with byte addressing differ by 4, not 1**

Memory Addressing Illustrated



15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	0

Showing all byte addresses

Memory for Large Data Structures

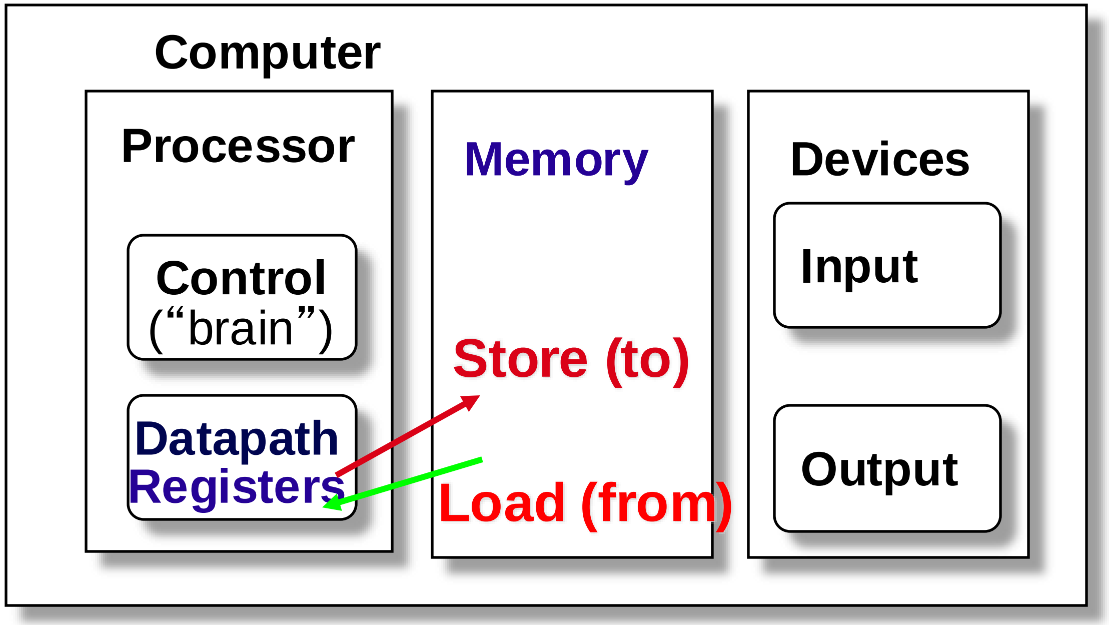
- **C variables map onto registers; what about large data structures like arrays?**
  - use memory
- **think of memory as a single one-dimensional array**

Array[index]	Address	Data or Content
A[5]	5020	32
A[4]	5016	23
A[3]	5012	14
A[2]	5008	72
A[1]	5004	56
A[0]	5000	44

Assuming A[ ] is an integer array

Data Transfer Instructions

- RISC-V arithmetic instructions only operate on registers, never directly on memory.
- Registers are in the datapath of the processor; if operands are in memory, we must transfer them to the processor to operate on them, and then transfer back to memory when done.
- Data transfer instructions
  - Memory to register
  - Register to memory



Memory to Register Transfer

- **To transfer a word of data, we need to specify two things:**
  - Register: specify this by number (x0 - x31) or symbolic name (zero, a0, …)
  - Memory address:
    - sum of two values
      - A register containing a pointer to memory address
      - Numerical offset from this pointer (in bytes)
    - Example: `8(t0)`
      - specifies the memory address pointed to by the value in t0, plus 8 bytes
- **Load FROM memory**

Load Word

- **Load Instruction Syntax:**

  1   2,3(4)

  - where

    1) operation name

    2) register that will receive value

    3) numerical offset in bytes

    4) register containing pointer to memory

- **RISC-V Instruction Name:**
  - `lw` : meaning Load Word, so 32 bits (i.e., one word) are loaded at a time

Load Word: Example

Example: `lw t0,12(s0)`

This instruction will take the pointer in `s0`, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register `t0`

- **Notes:**
  - `s0`: base register
    - Often points to beginning of an array or structure
  - 12: offset
    - often used in accessing elements of array or structure

Register to Memory Transfer (sw)

- **Store from register into memory**
  - syntax is identical to lw
  - `sw` (meaning Store Word, so 32 bits or one word are loaded at a time)

- **Example:** `sw t0,12(s0)`

  This instruction will take the pointer in `s0`, add 12 bytes to it, and then store the value from register `t0` into that memory address

- **Store INTO memory**

Example 1

- **Compile using registers:**
  `g = h + A[5];`
  - `g`: `s1`, `h`: `s2`, `s3`: base address of integer array `A`
- **What offset in `lw` to select `A[5]`**
- **4x5=20 to select `A[5]`: byte vs. word**

Array[index]	Memory Address	Data or Content
A[5]	5020	32
A[4]	5016	23
A[3]	5012	14
A[2]	5008	72
A[1]	5004	56
A[0]	5000	44

1. transfer from memory to register:

lw t0,<u>20</u>(s3) # *add* <u>20</u> to s3 to select A[5],
#put into t0

2. **add it to `h` and place in `g`**

add s1,s2,t0 # *s1 = h+A[5]*

s1	?	g
s2	25	h
s3	5000	&A[ ]
t0		

registers

Example 2

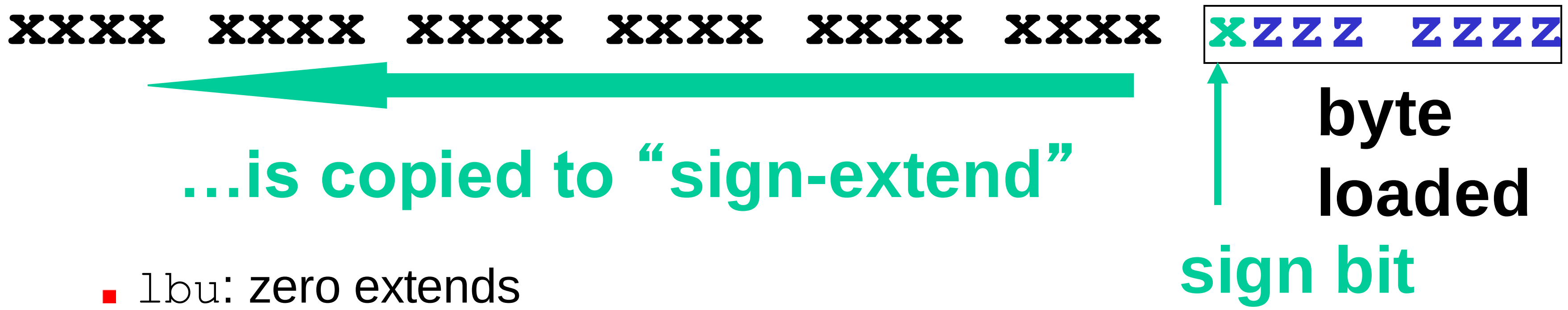
- **What offset in `lw` to select `A[i] (integer array)`**
  - 4i
- **Compile using registers:**

  `g = h + A[i];`
  - `g`: `s1`, `h`: `s2`, `s3`:base address of A; i: s4

```
add s4, s4, s4
add s4, s4, s4
add s4, s4, s3
lw  t0,0(s4)
add s1,s2,t0
```

Loading and Storing Bytes

- **byte data transfers:**
  - load byte: `lb`
  - store byte: `sb`
- **same syntax as `lw` and `sw`**
- **What to do with other 24 bits in the 32-bit register?**
  - `lb`: sign extends



  - `lbu`: zero extends
    - e.g., for chars

Working with Characters and Strings

- **ASCII**
  - American Standard Code for Information Interchange
  - 8-bit representation for English alphabet, numbers
  - RARS time: .word 0x65666768
- **Null-terminated series of characters are strings (same as in C)**
- **lb, sb used to read/write one char at a time**

C code:

```
char name [20];
name[3]
```

RISC-V code:

```
name: .space 20
la t0, name
lb t1, 3 (t0)
```

Other Load/Store Instructions

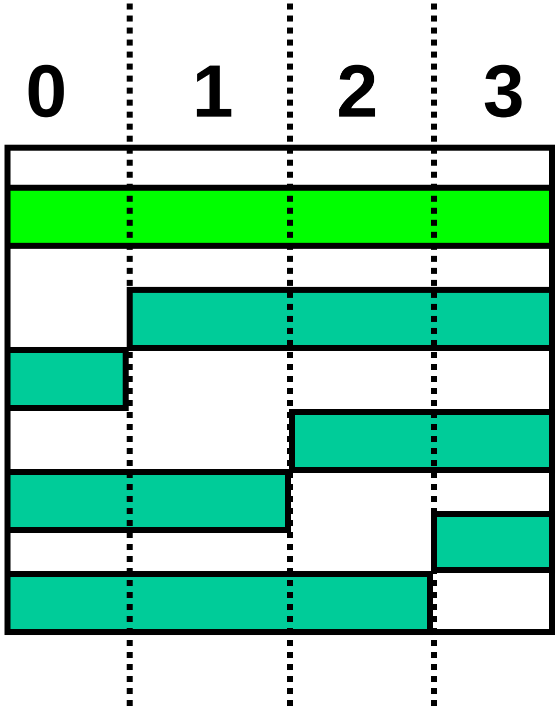
- **Half word:**
  - 2 bytes
  - Half word aligned
  - lh, lhu: load half word
  - sh: store halfword
- **For floating point numbers**
  - Use FP registers f0-f31
  - flw, fld – load float SP/DP
  - fsw, fsd – store float SP/DP

RISC-V Reference Card

lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	

Memory Alignment

- **RISC-V requires that all words start at byte addresses that are multiples of 4 bytes**



Last hex digit of memory address:

Word-aligned: low two bits are 0 (0,4,8,C)

Byte-aligned: anything

Halfword-aligned: low one bit is 0

Byte-aligned

- **Alignment: objects must fall on address that is multiple of their size**
  - For both lw and sw, the sum of the base address and the offset must be a multiple of 4 (to be word aligned)

RARS: Memory Alignment

- **.align <n>**
  - Ensure the next field is aligned to a multiple of 2^n
- **.align 2**
  - gives **word** alignment
  - Useful if you want to start a word array, but not sure if the previous items are of size in multiples of 4

Endian-ness

- **Storage order of bytes**
- **Little-endian (RISC-V)**
  - Store the little end of the number (least significant bits) first
- **Big-endian**
  - Store the big end of the number first
- **Example: 32766 (0x00007ffe)**

Memory address	**Little-endian**	**Big-endian**
0x10000003	**0x00**	**0xfe**
0x10000002	**0x00**	**0x7f**
0x10000001	**0x7f**	**0x00**
0x10000000	**0xfe**	**0x00**

Pointers vs. Values in Registers

- **Key Concept: A register can hold any 32-bit value.**
  - a (signed) `int`,
  - an `unsigned int`,
  - a pointer (memory address)
  - …
- **`add t2,t1,t0`**
  - `t0` and `t1` better contain **integers**
- **`lw t2,0(t0)`**
  - `t0` better contains a **pointer**

C code:	Compile assuming	RISC-V code:
int d, int *c;	c in s1	lw s2, 0(s1)
d = *c;	d in s2	

RARS: Variables

- **All globals**
  - Compilers created them, not a human
- **C code:**

```
int a, b, c = 0;
```

- **RISC-V code:**

```
.data
a: .space 4    # allocate 4 bytes, uninitialized
b: .space 4    # allocate 4 bytes, uninitialized
c: .word 0     # 4 bytes, initialized to 0
```

RARS: Defining Arrays

- **C code:**

```
int ray[100];
char str[128];
int nums[ ] = {3, 0, 1, 2, 6, -2, 4, 7, 3, 7};
```

- **RISC-V code**

```
.data
ray: .space 400  # 100 4-byte locations = 400 bytes
str: .space 128  # 1 byte/char = 128 bytes
nums: .word 3, 0, 1, 2, 6, -2, 4, 7, 3, 7 # 10 words!

.text
…
   la a0, ray  # first address of ray (&ray[0])
   la a1, nums # first address of nums - not nums[0]!
               # memory operations used to get to values
```

RARS: Using Arrays

- **C Code:**

```
ray [0] = 24;
ray [4] = 48;
```

- **RISC-V Code:**

```
la a0, ray
addi t0, zero, 24
sw t0, 0(a0)
addi t0, zero, 48
sw t0, 16(a0)
```

- **C Code:**

```
str [0] = `a';
str [4] = `b';
```

- **RISC-V Code:**

```
la a0, str
addi t0, zero, `a'
sb t0, 0(a0)
addi t0, zero, `b'
sb t0, 4(a0)
```

Macros in RARS

- **Macros in RARS**
  - See RARS Help
  - Understand example-macros.s from Canvas and use it in your work

Live Coding Time

- **Download code from Canvas**
  - Work on mem.s
  - Work on data-transfer.s
  - Work on printbyte.s

Live Coding Time

- **Turn this C function into a RISC-V program**

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

  - **read in b and c, output value rather than return result - for now**
  - **assume b+c<8 (just make sure you only give good values as input)**

Topics for Chapter 2

- **Part 1**
  - RISC-V Instruction Set (Chapter 2.1)
  - Arithmetic instructions (Chapter 2.2)
  - Introduction to RARS
  - Memory access instructions (Chapters 2.3, 2.9)
  - ➔ Bitwise instructions (Chapter 2.6)
  - Decision making instructions (Chapter 2.7)
  - Arrays vs. pointers (Chapter 2.14)
- **Part 2:**
  - Procedure Calls (Chapters 2.8)
- **Part 3:**
  - RISC-V Instruction Format (Chapter 2.5)
  - RISC-V Addressing Modes (Chapter 2.10)

Two Different Views of a Register

- **View contents of register as a single quantity (such as a signed or unsigned integer)**
  - Arithmetic (`add`, `sub`, `addi`) and memory access (`lw` and `sw`) instructions
- **View register as 32 raw bits**
  - Access individual bits (or groups of bits) rather than the whole 32-bit number
  - Bitwise operations
    - Logical and Shift Operations
    - bit 0 of the output is produced by the respective bit 0's of the inputs, bit 1 by the bit 1's, etc.

Logical Operations

- **Basic logical operators:**
  - AND: outputs 1 only if both inputs are 1
  - OR: outputs 1 if at least one input is 1
  - XOR: outputs 1 if only one input is 1
- **Truth Table: standard table listing all possible combinations of inputs and resultant output for each**

A	B	A AND B	A OR B	A XOR B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

- **Why no NOT?**
  - Can XOR with -1
  - **not t0, t2 can be done with xori t0, t2, -1**

Syntax of Logical Instructions

- **Syntax:**

  1 2,3,4

  1. operation name: and, or, xor, andi, ori, xori
  2. Destination register that will receive result
  3. first operand (register)
  4. second operand (register) or
     immediate (12-bit numerical constant)

     - immediate values are sign extended!

- **RISC-V assembly accepts exactly 2 inputs and produces 1 output**
  - rigid syntax, simpler hardware

Logical Instructions

operation	C	RISC-V
AND	a = b & c	and s0, s1, s2
OR	d = e | f	or s3, s4, s5
XOR	g = h ^ i	xor s6, s7, s8
NOT	a = ~b	xori s0, s1, -1

Usage of and

- `and`-ing a bit with 0 produces a 0 at the output
- `and`-ing a bit with 1 produces the original bit.
- can be used to create a mask.
  - Example:

```
                     1011 0110 1010 0100 0011 1101 1001 1010
mask:     &          0000 0000 0000 0000 0000 0111 1111 1111
---------------------------------------------------------------
                     0000 0000 0000 0000 0000 0101 1001 1010
```

mask last 11 bits

  - Mask: used to isolate the rightmost 11 bits of the first bitstring by masking out the rest of the string (e.g. setting it to all 0s).

```
andi      t0,t0,0x7FF
```

Usage of or

- `or`–ing a bit with 1 produces a 1 at the output
- `or`–ing a bit with 0 produces the original bit.
- or can be used to force certain bits of a string to 1s.
  - Example: set the rightmost 11 bits to 1, other bits are untouched

```
          1011 0110 1010 0100 0011 1101 1001 1010
|         0000 0000 0000 0000 0000 0111 1111 1111
---------------------------------------------------
          1011 0110 1010 0100 0011 1111 1111 1111
```

```
ori t0,t0,0x7FF
```

Exercise

NOR: NOT OR

nor (t0,t1) = not (or (t0,t1))

A	B	A nor B
0	0	1
0	1	0
1	0	0
1	1	0

How would you perform this in RISC-V?

Answer:

```
or t0, t0, t1
xori t0, t0, -1
```

Immediate Ranges

- **For addi/andi/ori/**
- **Immediate can be at most 12-bit signed integer (i.e., 3 hex digits)**
  - [-2^11, 2^11 -1] = [0xFFFFF800, 0x000007FF] = [-2048, 2047]
- **RARS is weird**
  - Every value represented in hex is interpreted as an unsigned value
    - 0xFFF will be interpreted as 0x00000FFF
    - 0xFFFFFFFF or -1 acceptable, but not 0xFFF
    - allows values in the range [-2048 … 2047], but does not permit hexadecimal values with MSB set to 1 (0x8_ _ … 0xF_ _): "operand is out of range"
  - Bottom line: Use the value that you actually want rather than a truncated one.

Syntax of Shift Instructions

- **Syntax: 1 2,3,4**
  - 1) operation name
  - 2) register that will receive value
  - 3) first operand (register)
  - 4) shift amount (register or constant < 32)
- **RISC-V shift instructions:**
  - sll, slli (shift left logical): shifts left and fills emptied bits with 0s
  - srl, srli (shift right logical): shifts right and fills emptied bits with 0s
  - sra, srai (shift right arithmetic): shifts right and fills emptied bits by sign extending

Example: srli and slli

- **Move (shift) all the bits in a word to the left or right by a number of bits.**
  - srli s0, s0, 8 (shift right by 8 bits)

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

  - slli s0, s0, 8 (shift left by 8 bits)

0001 0010 0011 0100 0101 0110 0111 1000

0011 0100 0101 0110 0111 1000 0000 0000

Example: srai

- **srai s0, s0, 8 : shift right arith by 8 bits**

  0001 0010 0011 0100 0101 0110 0111 1000

  0000 0000 0001 0010 0011 0100 0101 0110

- **srai s0, s0, 8 : shift right arith by 8 bits**

  1001 0010 0011 0100 0101 0110 0111 1000

  1111 1111 1001 0010 0011 0100 0101 0110

Usage of Shift Instructions

- **shifting left to multiply by powers of 2**
  - a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:

`a *= 8;` (in C)

would compile to:

`slli   s0,s0,3` (in RISC-V)

- **shift right to divide by powers of 2**
  - srl/srli with positive/unsigned values only
  - sra/srai rounds down

Shift Left and Overflow

- **For unsigned representation, when the first "1" is shifted out of the left edge, the operation has overflowed.**
  - i.e., the result of the multiplication is larger than the largest possible.

- **Shifting left on signed values also works, but overflow occurs when the most significant bit changes values (from 0 to 1, or 1 to 0).**

How to Set Bits in Upper Half?

- **Example: mask desired is 0xCAFE0000**
- **Solution 1**
  - Build the value in a register: use addi, slli, ori
  - and/or with the register

```
addi t0, zero, 0xCA    # t0 = 0xCA
slli t0, t0, 8         # t0 = 0xCA00
ori  t0, t0, 0xFE      # t0 = 0xCAFE
slli t0, t0, 16        # t0 = 0xCAFE0000
```

- **Solution 2: use lui (load upper immediate) for up to 20 bits**

```
add t0, zero, zero  # t0 = 0x00000000
lui t0, 0xCAFE0       # t0 = 0xCAFE0000
```

Immediate in lui

- **lui s1, 0xFFFAABBD**
  - Assembler: “operation completed with errors”
- **lui s1, 0x123AABBD**
  - Assembler: “operation completed with errors”
- **lui s1, 0xAABB**
  - s1=0xAABB0 000
  - If the immediate is less than 20 bits, 0 will be added on the right
- **Bottom line:**
  - can only take no more than 20 bits of real value in immediate
  - does not sign or zero extend. It works with the immediate directly

How to Set Large Values?

- **Set s1 to hold the value 0xAABBCCDD**

Solution 1:

```
# load and shift each byte
    addi s1, zero, 0x000000AA
    slli s1, s1, 8
    ori s1, s1, 0x000000BB
    slli s1, s1, 8
    ori s1, s1, 0x000000CC
    slli s1, s1, 8
    ori s1, s1, 0x000000DD
```

Solution 2:

```
lui s1, 0xAABBC
lui t0, 0xCDD00
srli t0, t0, 20
or s1, s1, t0
```

Solution 3:

```
lui s1, 0x000AABBD
```

#Add 1 to 0xAABBD because addi (next instruction) sign extends #(effectively subtracts 1 from the upper value)

```
addi s1, s1, 0xFFFFFCDD
```

#FFFFF is to keep the value in range (2^{11}-1 to -2^{11} for two's complement)

Live Coding Time (bitwise.s)

Assume that x (using s0) and y (using s1) are both integers. Assume further that integers are 4 bytes. Note that bits are numbered from *right to left* in a word starting with bit 0. Write two lines of RISC-V code to set x to the value contained in bits 15 to 10 of y.

```
srli s1, s1, 10       #  x = y>>10
andi s0, s1, 0x3F  # x = x&0x3F =(y >> 10) & 0x3F;
```

```
slli s1, s1, 16  # x = y <<16
srli s0, s1, 26 # x = x >> 26 (=(y<<16)>>26)
```

RARS: Pseudo instructions

- **Definition**
  - A RISC-V instruction that doesn’t turn directly into a machine language instruction, but broken up into several other RISC-V instructions
  - Provided by an assembler but not implemented in hardware
- **Examples**
  - Load address: la t0, str
  - Load immediate: li t0, 0x40044005
  - Move register content: mv t1, t2 # copy t2 to t1

RARS: Pseudo Instructions

- **Assembler converts pseudoinstructions to real RISC-V instructions**
- **Example 1, move:**

  mv t1, t2 # copy t2 into t1

  becomes

  add t1, t2, zero
- **Example 2, load immediate**

li s1, 0xAABBCCDD **becomes**

lui s1, 0x000AABBD

addi s1, s1, 0xFFFFFCDD

Topics for Chapter 2

- **Part 1**
  - RISC-V Instruction Set (Chapter 2.1)
  - Arithmetic instructions (Chapter 2.2)
  - Introduction to RARS
  - Memory access instructions (Chapters 2.3)
  - Bitwise instructions (Chapter 2.6)
  - → Decision making instructions (Chapter 2.7)
  - Arrays vs. pointers (Chapter 2.14)
- **Part 2:**
  - Procedure Calls (Chapters 2.8)
- **Part 3:**
  - RISC-V Instruction Format (Chapter 2.5)
  - RISC-V Addressing Modes (Chapter 2.10)

Making Decisions

- **To build a calculator**
  - instructions that manipulate data: arithmetic, memory access, logical and shift
- **To build a computer**
  - need instructions to make decisions
  - provide labels to support "`goto`" that jumps to places in code
    - Horrible style for C, but necessary for RISC-V

C Decisions: `if` Statements

- 2 kinds of `if` statements in C
  - `if (`*condition*`)` *clause*
  - `if (`*condition*`)` *clause1* `else` *clause2*

- Rearrange for RISC-V:

```
    if  (condition) goto L1;
             clause2;
             goto L2;
    L1: clause1;
L2:
```

Conditional Branch

- **Conditional branch instruction:**
  - `beq    register1, register2, L1`
  - `beq` is "Branch if (registers are) equal"
    Same meaning as (using C):
    `if  (register1==register2) goto L1`
- **Complementary decision instruction**
  - `bne    register1, register2, L1`
  - `bne` is "Branch if (registers are) not equal"
    Same meaning as (using C):
    `if  (register1!=register2) goto L1`

Unconditional Branch

- **Unconditional branch instruction**

  jal zero, label

  - jump (or branch) directly to the given label without needing to satisfy any condition
  - “zero” used as we won’t be returning
  - A label is a sequence of letters, numbers, underbars (_) and dots
- **Same as:**

  ```
  beq        zero,zero,label
  ```

  since it always satisfies the condition.
- **Same meaning as (using C):**

  ```
  goto label
  ```

Labels

- **Labels provided by programmer correspond to actual memory locations**
- **Labels get translated to locations embedded in instructions**
- **No memory used to store labels**

Memory Location	Instruction
`5048`	`add s0, x0, x0`
`5052`	`addi s1, s0, 1`
`5056`	`beq s1, s0, exit`
`5060`	
`5064`	
`5068`	
`...`	
`6000(exit)`	`ori …`

After `beq` evaluated, at which location does the instruction to be executed exist?

What if `beq` were replaced with `bne`?

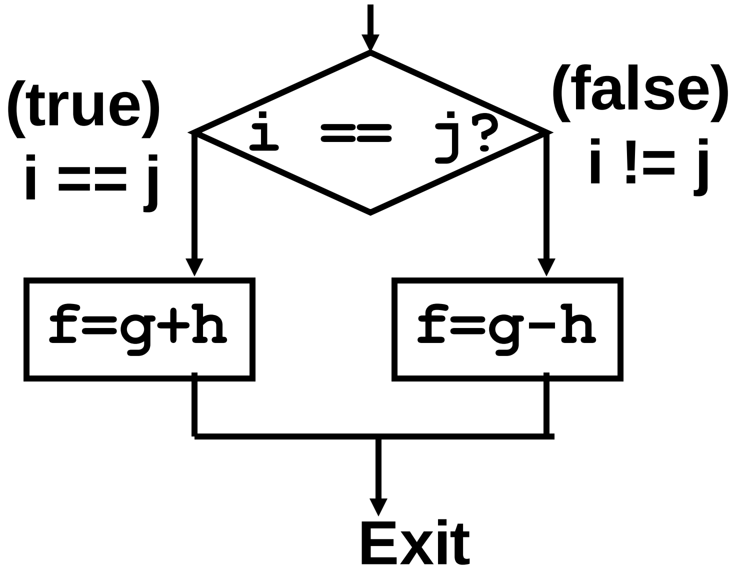
Example

- **Compile**

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

- **Use this mapping:**

f: s0
g: s1
h: s2
i: s3
j: s4



Example: Answer

- **Compile**

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

f: s0	g: s1
h: s2	i: s3
j: s4	

- **Final compiled RISC-V code:**

```
      beq s3,s4,True # branch i==j
      sub s0,s1,s2   # f=g-h(false)
      jal zero, Fin  # goto Fin
True: add s0,s1,s2   # f=g+h (true)
Fin:
```

- **Note: Compiler automatically creates labels to handle decisions (branches). Generally labels are not found in HLL code.**

Loops in C

- Simple loop in C; `A[]` is an array of `ints`

```
    do {
  g = g + A[i];
  i = i + j;
    } while (i != h);
```

- Rewrite this as:

```
    Loop:   g = g + A[i];
            i = i + j;
            if (i != h) goto Loop;
```

- Use this mapping:

g	h	i	j	Base of A
s1	s2	s3	s4	s5

Loops in Assembly

g	h	i	j	Base of A
s1	s2	s3	s4	s5

- **Original code:**

```
Loop:g = g + A[i];
          i = i + j;
          if (i != h) goto Loop;
```

- **Final compiled RISC-V code:**

```
Loop: slli t1,s3,2   #t1= 4*i
       add t1,t1,s5  #t1=addr A
       lw  t1,0(t1)    #t1=A[i]
       add s1,s1,t1  #g=g+A[i]
       add s3,s3,s4 #i=i+j
       bne s3,s2,Loop # goto Loop
                       # if i!=h
```

Loops in C/Assembly

- **Three types of loops in C:**
  - `while`
  - `do… while`
  - `for`

- **Each can be rewritten as either of the other two, so the method used in the previous example can be applied to `while` and `for` loops as well.**

- **Key Concept: Though there are multiple ways of writing a loop in RISC-V, the key to decision making is <u>conditional branch</u>**

Switch Statement in C: Example

- Choose among four alternatives depending on whether `k` has the value 0, 1, 2 or 3. Compile this C code:

```
switch (k) {
 case 0: f=i+j; break; /* k=0 */
 case 1: f=g+h; break; /* k=1 */
 case 2: f=g-h; break; /* k=2 */
 case 3: f=i-j; break; /* k=3 */
}
```

Switch Statement in C: Rewrite

- Rewrite it as a chain of if-else statements, which we already know how to compile:

```
if(k==0) f=i+j;
  else if(k==1) f=g+h;
    else if(k==2) f=g-h;
      else if(k==3) f=i-j;
```

- Use this mapping:

```
f:s0, g:s1, h:s2,
i:s3, j:s4, k:s5
```

Switch Statement in RISC-V Code

- Final compiled RISC-V code:

```
     bne s5,0,L1     # branch k!=0
     add  s0,s3,s4 #k==0 so f=i+j
     jal zero,Exit          # end of case so Exit
L1:  addi t0,s5,-1   # t0=k-1
     bne  t0,0,L2    # branch k!=1
     add  s0,s1,s2 #k==1 so f=g+h
     jal zero,Exit          # end of case so Exit
L2:  addi t0,s5,-2   # t0=k-2
     bne  t0,0,L3    # branch k!=2
     sub  s0,s1,s2 #k==2 so f=g-h
     jal zero,Exit          # end of case so Exit
L3:  addi t0,s5,-3   # t0=k-3
     bne  t0,0,Exit # branch k!=3
     sub  s0,s3,s4 #k==3 so f=i-j
Exit:
```

```
   if(k==0) f=i+j;
     else if(k==1) f=g+h;
       else if(k==2) f=g-h;
         else if(k==3) f=i-j;
```

```
f:s0, g:s1,
h:s2,i:s3, j:s4,
k:s5
```

Inequalities in RISC-V

- **Testing equalities**
  - == and != in C
  - beq and bne in RISC-V
- **What about <, >= ?**
  - blt t1, s1, TGT # if (t1 < s1) goto TGT
  - bge t1, s1, TGT # if (t1 >= s1) goto TGT
- **No ble or bgt. Why?**
  - RISC-V goal: Simpler is Better

>, ≤ in RISC-V

- **Compiler swaps the logic of the comparison to fit the available branch operators**

if (c > d)
goto Less;
	blt s3, s2, Less # if (d<c) goto Less

if (c <= d)
goto Less2;
	bge s3, s2, Less2 # if (d>=c) goto Less2

c: s2; d: s3

Complete the RISC-V Code

```
for(i = 0; i < 100; i++)
   b = b + A[i];
```

Assume:
i is in t1
b is in s1
base address of A is in s0

Label	Instruction
	addi t1, x0, 0
	addi t2, x0, 100
Loop:	lw t0, 0(s0)
	addi t1, t1, 1
	add s1, s1, t0
	addi s0, s0, 4
	blt t1, t2, Loop

Exercise

Given: for (i = 50; i > 10; --i), where i is in s0, and 10 is in t0, which branch is the most appropriate?

a) blt s0, t0, Loop
b) blti s0, 10, Loop
c) bge s0, t0, Loop
✓ d) blt t0, s0, Loop

Signed vs. Unsigned Comparison

- **Comparison instructions must deal with dichotomy between signed and unsigned numbers: blt vs. bltu**
  - MSB = 1
    - Signed: negative
    - Unsigned: large number
- **Example**
  - s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - blt s0, s1, L1 # -1<1, hence branch to L1
  - bltu s0, s1, L1 # 2^{32}-1 >1, hence do not branch

Bounds Check Shortcut

- **if ((k>=size) || (k<0)) goto IndexOutofBounds**
- **a1:k; a2: size, assuming size>0**

bgeu a1, a2, IndexOutofBounds # if k>=size or k<0

- **bgeu can do BOTH out-of-bounds check and a negative check**
  - All negative numbers, if interpreted as unsigned, are greater than positive numbers

RISC-V Decision-Making Instructions

beq	Branch ==	B	1100011	0x0		`if(rs1 == rs2) PC += imm`	
bne	Branch !=	B	1100011	0x1		`if(rs1 != rs2) PC += imm`	
blt	Branch <	B	1100011	0x4		`if(rs1 <  rs2) PC += imm`	
bge	Branch ≥	B	1100011	0x5		`if(rs1 >= rs2) PC += imm`	
bltu	Branch < (U)	B	1100011	0x6		`if(rs1 <  rs2) PC += imm`	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		`if(rs1 >= rs2) PC += imm`	zero-extends

Topics for Chapter 2

- **Part 1**
  - RISC-V Instruction Set (Chapter 2.1)
  - Arithmetic instructions (Chapter 2.2)
  - Introduction to RARS
  - Memory access instructions (Chapters 2.3)
  - Bitwise instructions (Chapter 2.6)
  - Decision making instructions (Chapter 2.7)
  - ➔ Arrays vs. pointers (Chapter 2.14)
- **Part 2:**
  - Procedure Calls (Chapters 2.8)
- **Part 3:**
  - RISC-V Instruction Format (Chapter 2.5)
  - RISC-V Addressing Modes (Chapter 2.10)

C Pointers

- **Each variable is assigned to memory or a register**
  - If assigned to memory, it is a stored value and has an address
  - Registers do not have addresses
- **In C**
  - & (reference operator): used to get the memory address of a stored value
  - * (dereference operator): used to get the value at a memory address
- **Array access is really just dereference**
  - *var is equivalent to var[0]
  - *(var+n) is equivalent to var[n]
  - Compiler knows how big the entries are, so it does the correct arithmetic to get the correct offset (i.e., n*size of the entry)
  - You are the compiler in this class

Arrays vs. Pointers

```
ClearArray (int array[ ], int size)
{
    int i;
    for (i=0; i<size; i+=1)
          array[ i ]=0;
}
```

```
ClearPtr(int *array, int size)
{
    int *p;
    for (p= &array[0]; p< &array[size]; p=p+1)
          *p=0;
}
```

increments by sizeof(int)!!

Arrays Version (C)

```
ClearArray (int array[ ], int size)
{
   int i;
   for (i=0; i<size; i+=1)
        array[ i ]=0;
}
```

Assume size>0

array:	a0
size:	a1
i:	t0

Translated C Code:

```
              i = 0;
loop1:        array[i] = 0;
              i++;
              if (i < size) goto loop1;
```

Arrays Version (RISC-V)

Translated C Code:

```
        i = 0;
loop1:  array[i] = 0;
        i++;
        if (i < size) goto Loop1;
```

Assume size>0

array:	a0
size:	a1
i:	t0

```
        add    t0, zero, zero      # i=0
loop1:  slli   t1, t0, 2           # t1=i*4
        add    t2, a0, t1          # t2=&array[i]
        sw     zero, 0 (t2)        # array[i]=0
        addi   t0, t0, 1           # i= i+1
        blt    t0, a1, loop1       # if () goto loop1
```

Pointer Version (C)

```
ClearPtr(int *array, int size)
{
    int *p;
    for (p= &array[0]; p< &array[size]; p=p+1)
          *p=0;
}
```

Assume size>0
array: a0
size: a1
p: t0

Translated C Code:

```
            p = &array[0];
loop2:      *p = 0;
            p ++;
            if (p < &array[size]) goto loop2;
```

Pointer Version (RISC-V)

Translated C Code:

Assume size>0
array: a0
size: a1
p: t0

```
            p = &array[0];
loop2:      *p = 0;
            p ++;
            if (p < &array[size]) goto loop2;
```

```
        add    t0, a0, x0       # p=address of array[0]
loop2:  sw     zero, 0(t0)      # memory[p]=0
        addi   t0, t0, 4        # p= p+4
        slli   t1, a1, 2        # t1=size *4
        add    t2, a0, t1       # t2=address of array[size]
        blt    t0, t2, loop2    # if (p<&array[size]) goto loop2
```

Arrays vs. Pointers (RISC-V)

```
        add t0, zero, zero  # i=0
loop1:  sll    t1, t0, 2          # t1=i*4
        add    t2, a0, t1     # t2=&array[i]
        sw     zero, 0 (t2)  # array[i]=0
        addi   t0, t0, 1          # i= i+1
        blt    t0, a1, loop1  # if (i<size) goto loop1
```

```
        slli   t1, a1, 2          # t1=size *4
        add    t2, a0, t1     # t2=address of array[size]
        add    t0, a0, 0      # p=address of array[0]
loop2:  sw     zero, 0(t0)    # memory[p]=0
        addi   t0, t0, 4          # p= p+4
        blt    t0,t2, loop2   # if (p<&array[size]) goto loop2
```

Summary

- **In RISC-V Assembly Language:**
  - Registers replace C variables
  - One Instruction (simple operation) per line
- **No types in RISC-V**
- **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, so we can add to it or subtract from it (using offset).**

Summary

- **RISC-V Instructions:**
  - Arithmetic: add, addi, sub,
  - Data transfer: lw, sw, lb, sb, lbu, lh,sh,lhu
  - Bitwise: and,andi, or,ori, sll,slli, srli,sra, srai
  - Branch: bne, beq, bge, bgeu, blt, bltu, jal
  - Special: ecall, la
- **Registers:**
  - C Variables: s0 – s11
  - Temporary Variables: t0 – t6
  - Zero: x0/zero
  - Arguments to functions: a0-a7
  - Return values: a0, a1