



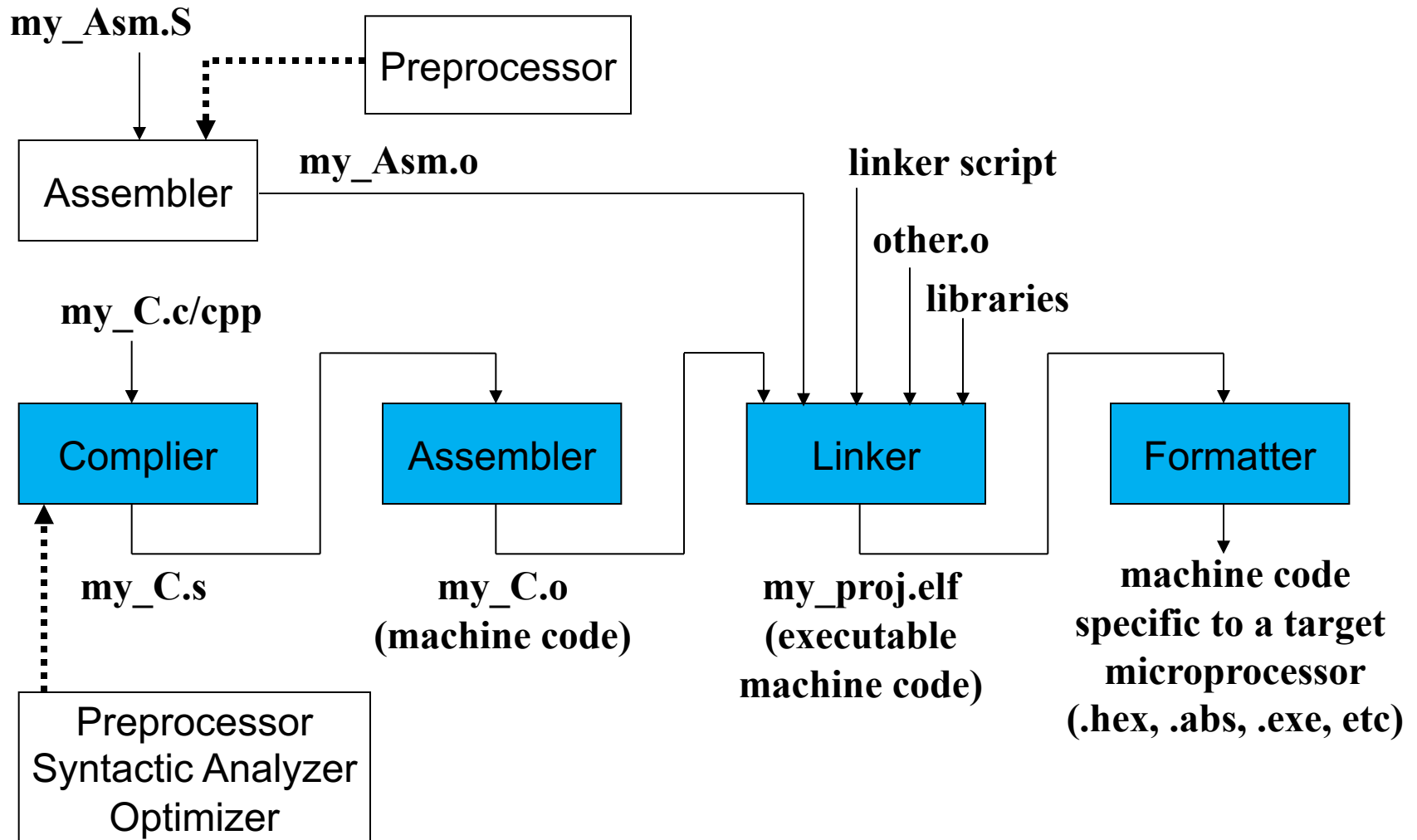
Topic 2

Assembly Programming - Operations and Operands

Levels of Program Code

- High-level language (translator: compiler)
 - Level of abstraction closer to problem domain
 - Provides productivity and portability
- Assembly language (translator: assembler)
 - Low-level language
 - Symbolic representation of binary machine code
 - Direct correspondence to machine code
 - More readable than machine code
- Machine language
 - Binary digits (bits) – language of digital circuits
 - Composed of instructions (commands for computer) and data
 - Instructions and data encoded in binary digitals

Processing Different Languages



Assembly Language

- When to use?
 - High level languages and compilers introduce uncertainty about execution time and size
 - Use assembly when speed and size of program are critical
 - Can mix high-level language with assembly

Assembly Language

- Drawbacks of Assembly language
 - Can be very time consuming
 - No assembler optimization
 - Almost impossible to be portable
 - Different computers support different assembly languages that requires different assembler
 - Assembly languages are similar
 - Hard to debug

Instruction Set

- or Instruction Set Architecture (ISA), a collection of instructions that a computer understands
- Different computers have different instruction sets
 - But with many common aspects
- Types of ISAs
 - Reduced Instruction Set Computer – RISC
 - Complex Instruction Set Computer – CISC

The RISC-V Instruction Set

- Used as the example throughout the book
- Developed at UC Berkeley as open ISA
- Now managed by the RISC-V Foundation (riscv.org)
- Large share of embedded core market
- Typical features of many modern ISAs
 - See RISC-V Reference Data on Canvas
- Has 32-bit (RV32), 64-bit (RV64), 128-bit (RV128) variants
 - We will base our discussions on RV32 in this class

Arithmetic Operations

- Add and subtract, three operands
 - Two sources and one destination

add a, b, c # a gets b + c
- All arithmetic operations have this form
- *Design Principle 1: Simplicity favors regularity*
 - Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost

Arithmetic Example

- C/C++ code:

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

- Corresponding pseudo-RISC-V code:

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

Operands in RISC-V Assembly

- Register operands
- Memory operands
- Immediate operands (constant)

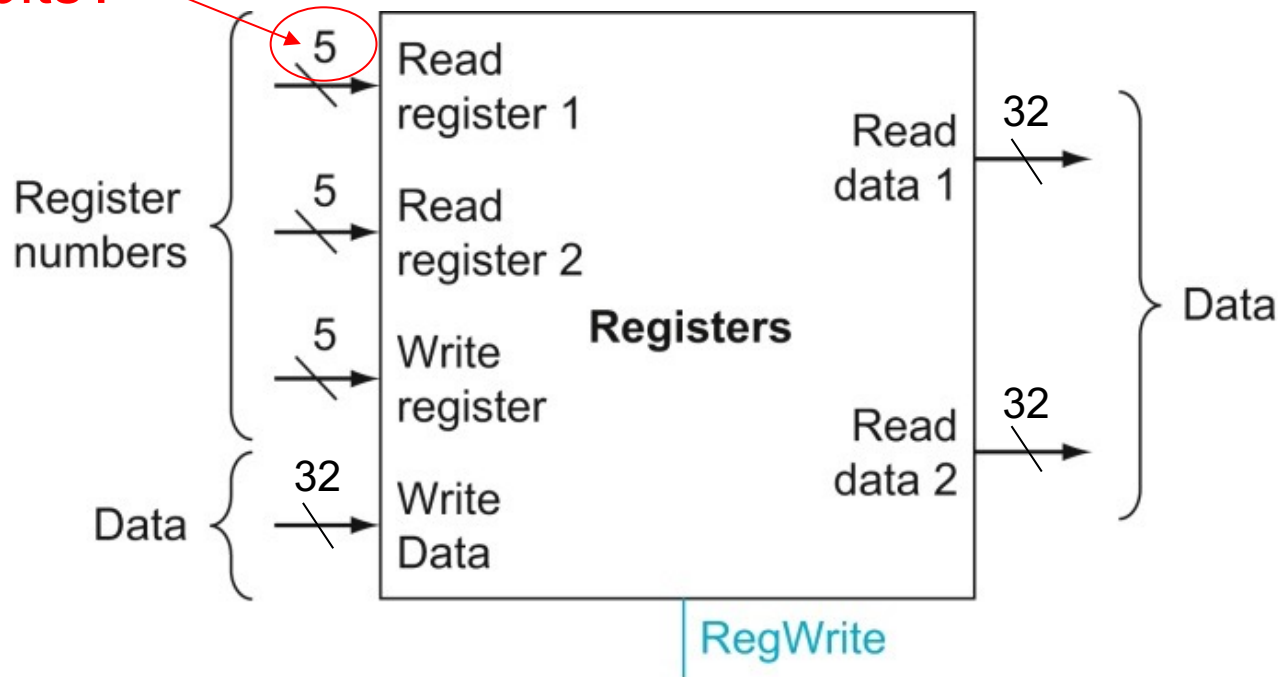
Register Operands

- Arithmetic instructions use register operands
- RISC-V RV32 has a 32×32 -bit **register file**
 - Referred to as x0 to x31
 - Used for frequently accessed data
 - 32×64 -bit registers in RV64 variant
- *Design Principle 2: Smaller is faster*
 - Small number of registers

Register File

The register file in this class is a bit different from what we designed previously

Why 5 bits?



Register Operands

- x0: the constant value 0
- x1 (ra): return address
- x2 (sp): stack pointer
- x3 (gp): global pointer
- x4 (tp): thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

Register Operand Example

- C/C++ code:

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

- Assume: f, g, h, i, and j are put in x19, x20, x21, x22, and x23, respectively

- Compiled RISC-V code:

add x5, x20, x21

add x6, x22, x23

sub x19, x5, x6

Operands in RISC-V Assembly

- Register operands
- **Memory operands**
- Immediate operands (constant)

Memory Operands

- Memory used mainly for composite data
 - Arrays, structures, dynamic data
- Steps to use memory operands
 - Load values from memory into registers
 - Perform arithmetic operations with registers
 - Store result from register to memory

RISC-V Memory organization

- RISC-V memory is **byte addressable**
 - Each 8-bit **byte** has a **unique address**
 - A **word** has 4 bytes, and **also has an address** which is the same as the smallest byte address in the word
 - So, a word address must be integer multiples of 4, i.e. the last digit of a word address must be 0, 4, 8, C

Word Addresses

(same as its smallest byte address)

Byte Addresses

	0xffff_0000	0xffff_0001	0xffff_0002	0xffff_0003
0xffff_0000				
	0xffff_0004	0xffff_0005	0xffff_0006	0xffff_0007
0xffff_0004				
	0xffff_0008	0xffff_0009	0xffff_000A	0xffff_000B
0xffff_0008				

RISC-V Memory organization

- RISC-V memory is Little Endian
 - Least-significant byte at smallest byte address of a word
 - Big Endian: most-significant byte at smallest address
- E.g.: 32-bit number 0x1020A0B0 (hexadecimal)

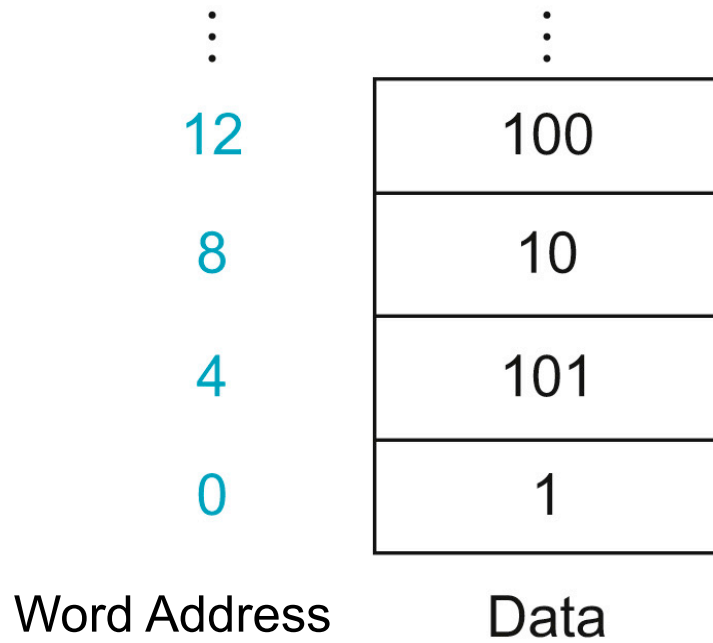
Big
Endian

	0xffff_0000	0xffff_0001	0xffff_0002	0xffff_0003
0xffff_0000	10	20	A0	B0

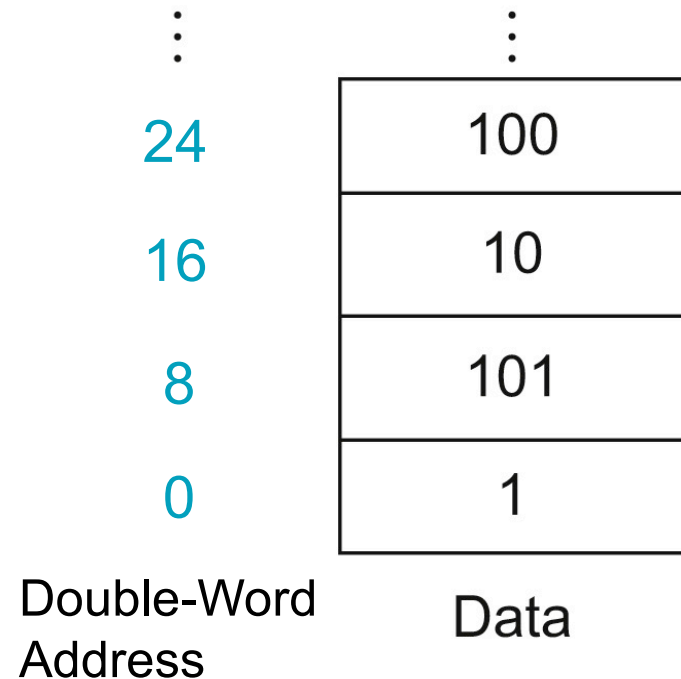
Little
Endian

	0xffff_0000	0xffff_0001	0xffff_0002	0xffff_0003
0xffff_0000	B0	A0	20	10

RISC-V Memory Organization



RV32 Memory



RV64 Memory

This is what we will be considering in this class

Array in Memory

Addr	Content
Base Address	.
.	.
(A[0])0xFFEE0000	0xABCD3305
(A[1])0xFFEE0004	0x0000FF00
(A[2])0xFFEE0008	0x12345678
(A[3])0xFFEE000C	0x00000001
.	.
.	.
(A[8])0xFFEE0020	0xAABB5566
(A[9])0xFFEE0024	0x00000000
.	.
.	.

Memory

- **A[] is an integer array. Each element occupies a word (4 bytes)**
- **A[] has a base address which is the address of A[0]**

Address of A[i]
= Base Address + Offset
= Base Address + (i × 4)

So,
Address of A[8]
= Base Address + (i × 4)
= 0xFFEE0000 + (8 × 4)
= 0xFFEE0020

Memory Operand Example 1

- C/C++ code:

`g = h + A[8];` #g, h, A are words

- g in x20, h in x21, base address of A in x22

- Compiled RISC-V code:

- Index 8 requires offset of 32 (4 bytes/word)

`lw x9, 32(x22) # load word`
`add x20, x21, x9`

Signed offset

base address register

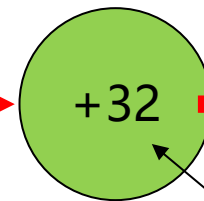
Load Word

lw x9, 32(x22)

Index Content

x0	
x1	
.	.
.	.
x9	0xAABB5566
.	.
.	.
x22	0xFFEE0000
.	.
.	.
x31	

Register File



base
address

offset

(A[0])0xFFEE0000

0xFFEE0004

0xFFEE0008

0xFFEE000C

0xFFEE0020

0xFFEE0024

Addr

Content

.	.
.	.
(A[0])0xFFEE0000	0xABCD3305
0xFFEE0004	0x0000FF00
0xFFEE0008	0x12345678
0xFFEE000C	0x00000001
.	.
.	.
0xFFEE0020	0xAABB5566
0xFFEE0024	0x00000000
.	.
.	.

Memory

Memory Operand Example 2

- C code:

`A[12] = h + A[8];`

- `h` in `x21`, base address of `A` in `x22`

- Compiled RISC-V code:

- Index 8 requires offset of 32

`lw x9, 32(x22) # load word`

`add x9, x21, x9`

`sw x9, 48(x22) # store word`

Signed offset

base address register

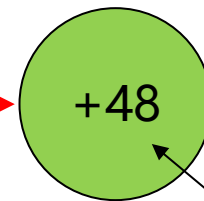
Store Word

SW x9, 48(x22)

Index Content

x0	
x1	
.	.
.	.
x9	0xAABB5566
.	.
.	.
x22	0xFFEE0000
.	.
.	.
x31	

Register File



base
address

offset

(A[0])0xFFEE0000

0xFFEE0004

0xFFEE0008

0xFFEE000C

0xFFEE0020

0xFFEE0030

Addr Content

.	.
.	.
(A[0])0xFFEE0000	0xABCD3305
0xFFEE0004	0x0000FF00
0xFFEE0008	0x12345678
0xFFEE000C	0x00000001
.	.
.	.
0xFFEE0020	0xAABB5566
.	.
.	.
0xFFEE0030	0xAABB5566
.	.
.	.

Memory

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
 - More instructions to be executed
- Compiler must use registers for variables as much as possible
 - Only spill to memory for less frequently used variables
 - Register optimization is important!

Operands in RISC-V Assembly

- Register operands
- Memory operands
- Immediate operands (constant)

Immediate Operands

- Immediate operands – 12-bit signed constant data specified in an instruction
`addi x22, x22, 4`
- No subtract immediate instruction
 - Just use a negative constant
`addi x22, x22, -1`
- *Design Principle 3: Make the common case fast*
 - Small constants are common
 - Immediate operand avoids loading data from memory which takes time

The Constant Zero

- RISC-V register 0 (**x0**) is the constant 0
 - Cannot be changed
- Useful for common operations
 - E.g., move between registers
add x9, x21, x0
 - E.g., clear a register
add x9, x0, x0

Other RISC-V Instructions

- Logical Operations
- Conditional Operations
- Load upper immediate instruction
- Signed and unsigned variants of instructions

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Shift left	<<	<<	slli
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	

- Useful for extracting and inserting groups of bits in a word

Shift Operations

sll*i* x5, x6, 3 # x5 = x6 << 3

sll x5, x6, x7 # x5 = x6 << x7

srai x5, x6, 3 # x5 = x6 >> 3

- **sll** or **sll***i*: shift left logical

- Fill vacated bits with 0 bits
- **sll** by *i* bits = multiplies by 2^i

- **sr***l* or **sr***l**i*: shift right logical

- Fill vacated bits with 0 bits
- **sr***l* by *i* bits = divides by 2^i (unsigned only)

- **srai**: shift right arithmetic

- Fill vacated bits with sign bit

AND Operations

- Useful to mask bits in a word
 - Select some bits, clear others to 0

and x9, x10, x11

x11	0000 0000 0000 0000 0000 1101 1100 0000
x10	0000 0000 0000 0000 0011 1100 0000 0000
x9	0000 0000 0000 0000 0000 1100 0000 0000

OR Operations

- Useful to include bits in a word
 - Set some bits to 1, leave others unchanged

or x9, x10, x11

x11	0000 0000 0000 0000 0000 1101 1100 0000
x10	0000 0000 0000 0000 0011 1100 0000 0000
x9	0000 0000 0000 0000 0011 1101 1100 0000

XOR Operations

- E.g., differencing operation
 - Set some bits to 1, leave others unchanged

xor x9, x10, x12 # NOT operation

x10	00000000	00000000	00001101	11000000
x12	11111111	11111111	11111111	11111111
x9	11111111	11111111	11110010	00111111

Other RISC-V Instructions

- Logical Operations
- **Conditional Operations**
- Load upper immediate instruction
- Signed and unsigned variants of instructions

Conditional Operations

- Each instruction can have a label (optional)
- Branch to a labeled instruction if a condition is true
 - Otherwise, continue sequentially
- **beq** rs1, rs2, L1
 - if (rs1 == rs2) branch to instruction labeled L1
- **bne** rs1, rs2, L1
 - if (rs1 != rs2) branch to instruction labeled L1

Compiling If Statements

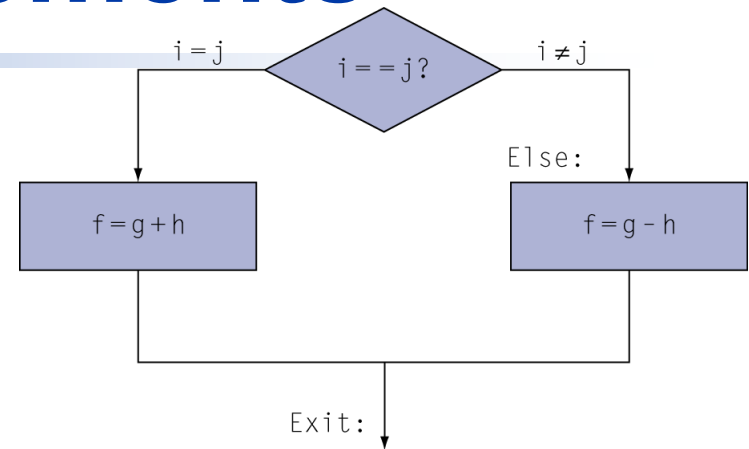
■ C code:

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

- f, g, h, i, j in x19, x20, x21, x22, x23

■ Compiled RISC-V code:

```
    bne x22, x23, Else # i==j?  
    add x19, x20, x21  # yes  
    beq x0,x0,Exit #unconditional branch  
Else: sub x19, x20, x21 # no  
Exit: ...
```



Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

- i in x22, k in x24, base address of save in x25

- Compiled RISC-V code:

```
Loop: slli x10, x22, 2    #calculate offset
      add  x10, x10, x25  #base + offset
      lw   x9, 0(x10)     #get save[i]
      bne  x9, x24, Exit  #save[i]==k?
      addi x22, x22, 1    #i += 1;
      beq  x0, x0, Loop   #uncond. branch
Exit: ...
```

More Conditional Operations

- **blt** rs1, rs2, L1
 - if ($rs1 < rs2$) branch to instruction labeled L1
- **bge** rs1, rs2, L1
 - if ($rs1 \geq rs2$) branch to instruction labeled L1
- Example
 - if ($a > b$) $a += 1$; #a in x22, b in x23
 - Compiled RISC-V code:
 bge x23, x22, Exit #branch if $b \geq a$
 addi x22, x22, 1 #if $b < a$, $a++$
Exit:.....

Other RISC-V Instructions

- Logical Operations
- Conditional Operations
- Load upper immediate instruction
- Signed and unsigned variants of instructions

Load 20-bit Constants

lui rd, constant

- Copies a 20-bit constant to bits [31:12] of rd and clears bits [11:0]
- Most constants are small
 - 20-bit immediate is sufficient
- For the occasional 32-bit constant

lui x19, 0x0003D

0000 0000 0000 0011 1101	0000 0000 0000
--------------------------	----------------

addi x19, x19, 0x090

0000 0000 0000 0011 1101	0000 1001 0000
--------------------------	----------------

Load 32-bit constant 0x0003D090 into x19

Other RISC-V Instructions

- Logical Operations
- Conditional Operations
- Load upper immediate instruction
- Signed and unsigned variants of instructions

2's-Complement Signed Integers

- Bit 31 is sign bit
 - 1 for negative numbers
 - 0 for non-negative numbers
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - Most-negative: 1000 0000 ... 0000
 - Most-positive: 0111 1111 ... 1111

Signed vs. Unsigned

- Signed comparison: blt, bge
- Unsigned comparison: bltu, bgeu
- Example
 - $x_{22} = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$
 - $x_{23} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001$
 - $x_{22} < x_{23}$ if signed: $-1 < +1$
 - $x_{22} > x_{23}$ if unsigned: $+4,294,967,295 > +1$

Sign Extension

- Representing a number using more bits
 - Preserve the numeric value
- Replicate the sign bit to the left
 - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
 - +2: 0000 0010 => 0000 0000 0000 0010
 - -2: 1111 1110 => 1111 1111 1111 1110

Byte/Halfword Operations

- Transfer byte/halfword in RV32

lb rd, offset(rs1) / **lh** rd, offset(rs1)

#Sign extend to 32 bits in R[rd]

#R is register file

lbu rd, offset(rs1) / **lhu** rd, offset(rs1)

#Zero extend to 32 bits in R[rd]

sb rs2, offset(rs1) / **sh** rs2, offset(rs1)

#Store just lowest byte/halfword of R[rs2]

- In RV64: ld (load double), sd (store double)

RISC-V Instruction Set

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	$x5 = x6 + x7$	Three register operands; add
	Subtract	sub x5, x6, x7	$x5 = x6 - x7$	Three register operands; subtract
	Add immediate	addi x5, x6, 20	$x5 = x6 + 20$	Used to add constants
Data transfer	Load word	lw x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Word from memory to register
	Load word, unsigned	lwu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Unsigned word from memory to register
	Store word	sw x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Word from register to memory
	Load halfword	lh x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Unsigned halfword from memory to register
	Store halfword	sh x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Halfword from register to memory
	Load byte	lb x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Byte unsigned from memory to register
	Store byte	sb x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Byte from register to memory
	Load reserved	lr.d x5, (x6)	$x5 = \text{Memory}[x6]$	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	$\text{Memory}[x6] = x5; x7 = 0/1$	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345	$x5 = 0x12345000$	Loads 20-bit constant shifted left 12 bits
Logical	And	and x5, x6, x7	$x5 = x6 \& x7$	Three reg. operands; bit-by-bit AND
	Inclusive or	or x5, x6, x8	$x5 = x6 \mid x8$	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x9	$x5 = x6 \wedge x9$	Three reg. operands; bit-by-bit XOR
	And immediate	andi x5, x6, 20	$x5 = x6 \& 20$	Bit-by-bit AND reg. with constant
	Inclusive or immediate	ori x5, x6, 20	$x5 = x6 \mid 20$	Bit-by-bit OR reg. with constant
	Exclusive or immediate	xori x5, x6, 20	$x5 = x6 \wedge 20$	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	sll x5, x6, x7	$x5 = x6 \ll x7$	Shift left by register
	Shift right logical	srl x5, x6, x7	$x5 = x6 \gg x7$	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	$x5 = x6 \gg x7$	Arithmetic shift right by register
	Shift left logical immediate	slli x5, x6, 3	$x5 = x6 \ll 3$	Shift left by immediate
	Shift right logical immediate	srl_i x5, x6, 3	$x5 = x6 \gg 3$	Shift right by immediate
	Shift right arithmetic immediate	srai x5, x6, 3	$x5 = x6 \gg 3$	Arithmetic shift right by immediate

RISC-V Instruction Set

Conditional branch	Branch if equal	beq x5, x6, 100	if (x5 == x6) go to PC+100	PC-relative branch if registers equal
	Branch if not equal	bne x5, x6, 100	if (x5 != x6) go to PC+100	PC-relative branch if registers not equal
	Branch if less than	blt x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greater or equal	bge x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
	Branch if less, unsigned	bltu x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	bgeu x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	jal x1, 100	x1 = PC+4; go to PC+100	PC-relative procedure call
	Jump and link register	jalr x1, 100(x5)	x1 = PC+4; go to x5+100	Procedure return; indirect call

To be discussed in next topic