CSE340: Computer Architecture

Chapter - 2: Instructions: Language of the Computer

The RISC-V Instruction Set

- The words of a computer's language are called instruction.
- The vocabulary is called instruction set.
- RISC-V instruction set is used as example throughout the course.
- Developed at UC Berkeley as open ISA
- Now managed by the RISC-V Foundation (riscv.org)

Arithmetic Operations: add, sub

- 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

- C code:

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

- Compiled RISC-V code:

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

RISC-V Registers

What is a register?
=> A **register** is a small, fast storage location within the CPU used to temporarily hold data.

- Arithmetic instructions use register operands.
- RISC-V has a 32 X 64-bit register file.
- 32 general purpose registers.
- Each register can store 64 bits of data.
- 64-bit data is called a "Double word"
- ***Design Principle-2:*** Smaller is faster

Register Number	Functionalities
x0	the constant value 0
x1	return address
x2	stack pointer
x3	global pointer
x4	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 code:

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

  - f, …, j in x19, x20, …, x23

- Compiled RISC-V code:

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

Memory Operand

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- RISC-V is Little Endian
  - Least-significant byte at least address of a word
  - *c.f.* Big Endian: most-significant byte at least address
- RISC-V does not require words to be aligned in memory
  - Unlike some other ISAs

Memory Operand Example

- 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 64
    - 8 bytes per doubleword

```
ld      x9, 64(x22)
add     x9, x21, x9
sd      x9, 96(x22)
```

Register 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!

Immediate Operands

- Constant data specified in an instruction
  addi x22, x22, 4

- Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

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

- In RISC-V instruction set
  - lb: sign-extend loaded byte
  - lbu: zero-extend loaded byte

Representing Instructions

- Instructions are encoded in binary
  - Called machine code

- RISC-V instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

RISC-V R-format Instructions

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Instruction fields
  - opcode: operation code
  - rd: destination register number
  - funct3: 3-bit function code (additional opcode)
  - rs1: the first source register number
  - rs2: the second source register number
  - funct7: 7-bit function code (additional opcode)

R-format Example

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

add x9,x20,x21

0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_2 = 015A04B3_{16}$

RISC-V I-format Instructions

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

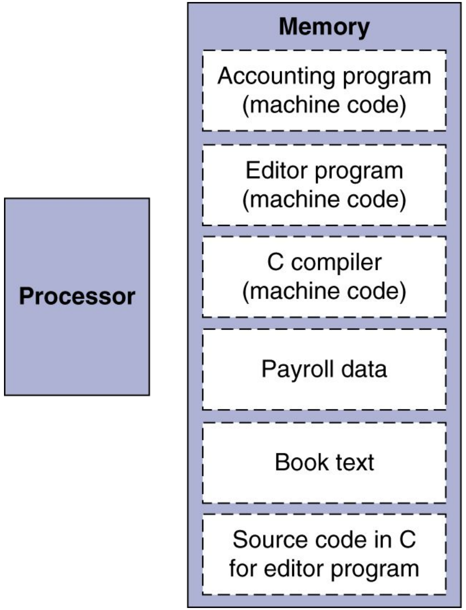
- Immediate arithmetic and load instructions
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2s-complement, sign extended
- *Design Principle 3:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

RISC-V S-format Instructions

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place

Stored Program Computers



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

Logical Operations

- Instructions for bitwise manipulation

<table>
<tr><th>Operation</th><th>C</th><th>Java</th><th>RISC-V</th></tr>
<tr><td>Shift left</td><td><<</td><td><<</td><td>slli</td></tr>
<tr><td>Shift right</td><td>>></td><td>>>></td><td>srli</td></tr>
<tr><td>Bit-by-bit AND</td><td>&</td><td>&</td><td>and, andi</td></tr>
<tr><td>Bit-by-bit OR</td><td>|</td><td>|</td><td>or, ori</td></tr>
<tr><td>Bit-by-bit XOR</td><td>^</td><td>^</td><td rowspan="2">xor, xori</td></tr>
<tr><td>Bit-by-bit NOT</td><td>~</td><td>~</td></tr>
</table>

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

Shift Operations

funct6	immed	rs1	funct3	rd	opcode
6 bits	6 bits	5 bits	3 bits	5 bits	7 bits

- immed: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - slli by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - srli by *i* bits divides by 2^i (unsigned only)

AND Operations

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

and x9,x10,x11

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9	00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000

OR Operations

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

or x9,x10,x11

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9	00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000

XOR Operations

- Differencing operation
  - Set some bits to 1, leave others unchanged

xor x9,x10,x12 // NOT operation

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

Conditional Operations

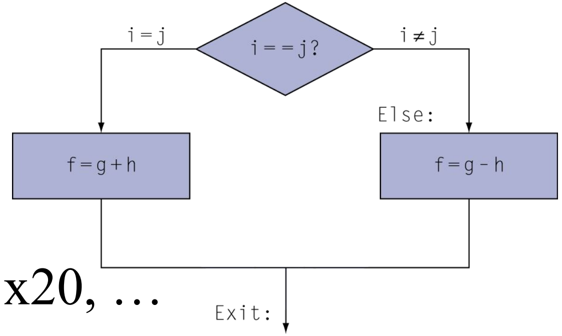
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- beq rs1, rs2, L1
  - if (rs1 == rs2) branch to instruction labeled L1
- bne rs1, rs2, L1
  - if (rs1 != rs2) branch to instruction labeled L1
- blt rs1, rs2, L1
  - if (rs1 < rs2) branch to instruction labeled L1
- bge rs1, rs2, L1
  - if (rs1 >= rs2) branch to instruction labeled L1

Compiling Conditional Statements

- C code:

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

  - f, g, … in x19, x20, …



- Compiled RISC-V code:

  bne x22, x23, Else
  add x19, x20, x21
  beq x0,x0,Exit //
  unconditional
  Else: sub x19, x20, x21
  Exit: …

C code:

- if (a > b) a += 1;
- a in x22, b in x23

Compiled RISC-V code:

bge x23, x22, Exit
addi x22, x22, 1
Exit:

Assembler calculates addresses

Compiling a Loop

Here is a traditional loop in C:

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

Assume that i and k correspond to registers x22 and x24 and the base of the array save is in x25. What is the RISC-V assembly code corresponding to this C code? - **Try yourself**

Signed vs. Unsigned

- Signed comparison: blt, bge
- Unsigned comparison: bltu, bgeu
- Example
  - x22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - x23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - x22 < x23 // signed
    - –1 < +1
  - x22 > x23 // unsigned
    - +4,294,967,295 > +1

Byte/Halfword/Word Operations

- RISC-V byte/halfword/word load/store
  - Load byte/halfword/word: Sign extend to 64 bits in rd
    - lb rd, offset(rs1)
    - lh rd, offset(rs1)
    - lw rd, offset(rs1)
  - Load byte/halfword/word unsigned: Zero extend to 64 bits in rd
    - lbu rd, offset(rs1)
    - lhu rd, offset(rs1)
    - lwu rd, offset(rs1)
  - Store byte/halfword/word: Store rightmost 8/16/32 bits
    - sb rs2, offset(rs1)
    - sh rs2, offset(rs1)
    - sw rs2, offset(rs1)

LUI: 32-bit Constants

- Most constants are small
  - 12-bit immediate is sufficient
- For the occasional 32-bit constant

  lui rd, constant
  - Copies 20-bit constant to bits [31:12] of rd
  - Extends bit 31 to bits [63:32]
  - Clears bits [11:0] of rd to 0

lui x19, 976 // 0x003D0

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000

addi x19,x19,1280 // 0x500

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000

Branch Addressing

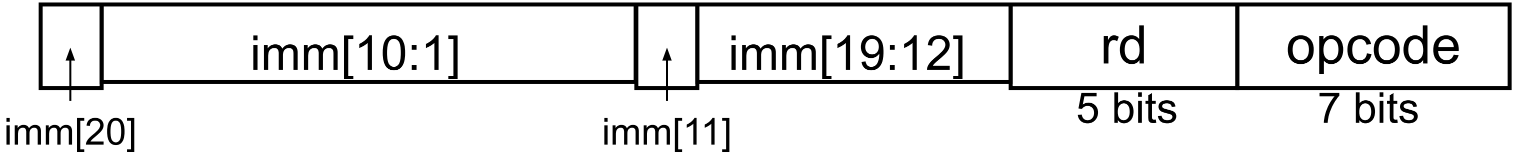
- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward
- SB format:

imm[12]	imm [10:5]	rs2	rs1	funct3	imm [4:1]	imm[11]	opcode

- PC-relative addressing
  - Target address = PC + immediate × 2

Jump Addressing

- Jump and link (jal) target uses 20-bit immediate for larger range
- UJ format:

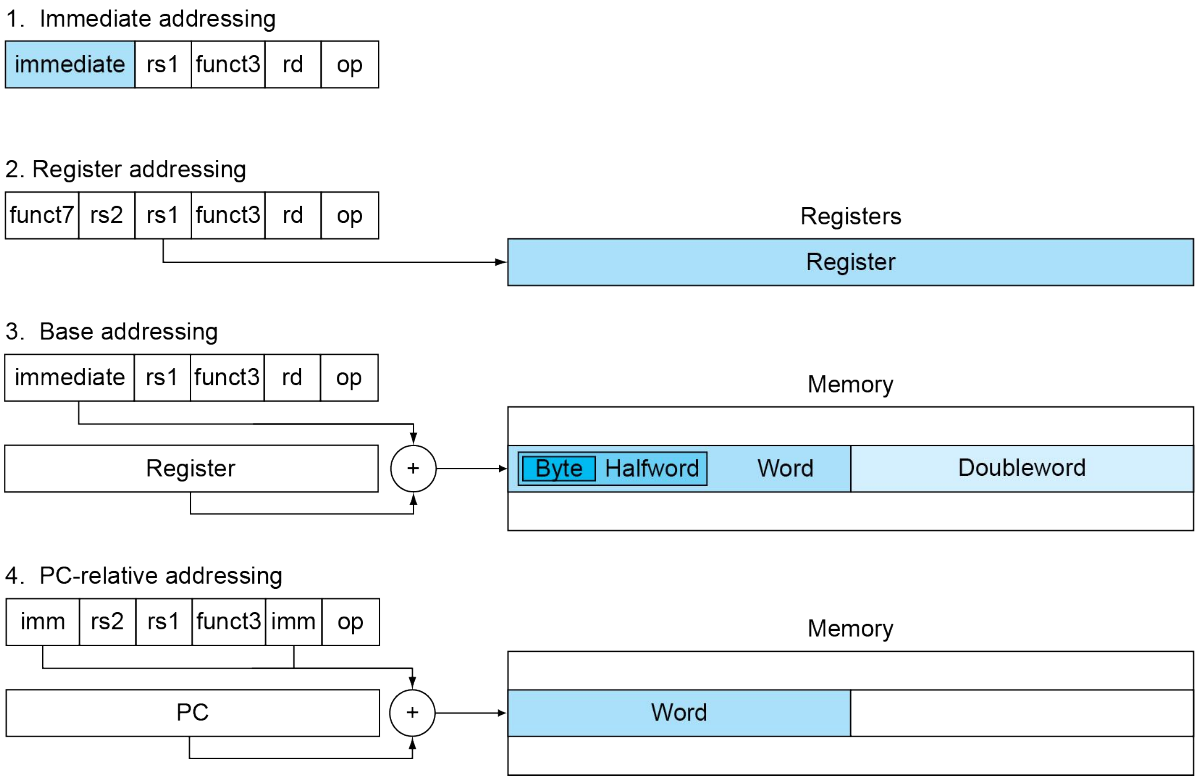


- For long jumps, eg, to 32-bit absolute address
  - lui: load address[31:12] to temp register
  - jalr: add address[11:0] and jump to target

Instruction format Summary

Name (Field size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
R-type	funct7	rs2	rs1	funct3	rd	opcode
I-type	immediate[11:0]		rs1	funct3	rd	opcode
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode
U-type	immediate[31:12]				rd	opcode

RISC-V Addressing Summary



Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Good design demands good compromises
- Make the common case fast
- Layers of software/hardware
  - Compiler, assembler, hardware
- RISC-V: typical of RISC ISAs
  - c.f. x86