ISA and RISCV

CASS 2018

Lavanya Ramapantulu
Program

• Program = ??

• Algorithm + Data Structures – Niklaus Wirth

• Program (Abstraction) of processor/hardware that executes
Program Abstractions

- You write programs in high level programming languages, e.g., C/C++, Java:

  ```c
  swap(int v[], int k)
  { int temp;
      temp = v[k];
      v[k] = v[k+1];
      v[k+1] = temp;
  }
  ```

  \[ k = k+1 \]

- **Compiler** translates this into assembly language statement:

  ```assembly
  slli x6, x11, 3
  add x6, x10, x6
  ld x5, 0(x6)
  ld x7, 8(x6)
  sd x7, 0(x6)
  sd x5, 8(x6)
  jalr x0, 0(x1)
  ```

- **Assembler** translates this statement into machine language instructions that the processor can execute:

  ```
  0000 0000 0110 0101 0000 0011 0011 0011
  ```
Program Abstractions – Why?

- **Machine code**
  - Instructions are represented in binary
  - `1000110010100000` is an instruction that tells one computer to add two numbers
  - Hard and tedious for programmer

- **Assembly language**
  - Symbolic version of machine code
  - Human readable
  - `add A, B` is equivalent to `1000110010100000`
  - Assembler translates from assembly language to machine code

- How many assembly instructions should you write to read two memory locations, and add them?
Instruction Set Architecture

An abstraction on the interface between the hardware and the low-level software.

Software
(to be translated to the instruction set)

Hardware
(implementing the instruction set)

Does a given ISA have a fixed implementation?

ISA Design Philosophies

• **Complex Instruction Set Computer (CISC)**
  – example: x86-32 (IA32)
  – single instruction performs complex operation
  – smaller assembly code size
    • lesser memory for storing program

• **Reduced Instruction Set Computer (RISC)**
  – example: MIPS, ARM, **RISC-V**
  – multiple instructions to perform complex operation

• Compiler design ?
• Hardware implementation ?
Execution flow of a program

• example of the computer components activated, instructions executed and data flow during an example code execution

```c
#include <stdio.h>

int main()
{
    int i, res = 0;
    for (i = 1; i < 10; i++) {
        res = res + i;
    }
    printf("Res = %d\n", res);
    return 0;
}
```

Recap: Computer Components

- What are the two major components in a computer

![Diagram of computer components](http://www.comp.nus.edu.sg/~cs2100/lect/cs2100-9-MIPS-1-full.pptx)

Recap: Execution flow and Data flow

• The code and data reside in memory
  – Transferred into the processor during execution
Memory access is slow!

- To avoid frequent access of memory
  - Provide temporary storage for values in the processor (known as registers)

![Diagram of processor architecture with memory access and ALU]

Memory instruction

• Need instruction to move data into registers
  – also from registers to memory later

Moving data from memory into a register – load
Moving data from a register into memory – store

Reg-to-Reg Arithmetic
• Arithmetic operation can now work directly on registers only:
  – Much faster!

Reg-to-Reg Arithmetic

- Sometimes, arithmetic operation uses a **constant** value instead of register value

Execution sequence

• Instruction is executed sequentially by default
  – How do we “repeat” or “make a choice”?

```
if r0 < 10,
  r0 <-- r0 + 1
  r1 <-- r1 + r0
```

Control flow instruction

• We need instruction to change the control flow based on condition:
  – Repetition (loop) and Selection (if-else) can both be supported

```
if r0 < 10,
  r0 ← r0 + 1
  r1 ← r1 + r0
if r0 < 10,
```

Looping!

• Since the condition succeeded, execution will repeat from the indicated position

```
if r0 < 10,
r0 ← r0 + 1
r1 ← r1 + r0
```

Looping!

• Execution will continue sequentially:
  – Until we see another control flow instruction!

Control flow instruction

- The three instructions will be repeated until the condition fails

Memory instruction

• We can now move back the values from register to their “home” in memory
  – Similarly for the "r1" to "res"

![Diagram showing processor and memory interaction]

Summary of observations

• The stored-memory concept:
  – Both instruction and data are stored in memory

• The load-store model:
  – Limit memory operations and relies on registers for storage during execution

• The major types of assembly instruction:
  – Memory: Move values between memory and registers
  – Calculation: Arithmetic and other operations
  – Control flow: Change the sequential execution

ISA Concepts

Concept #1: Data Storage
Concept #2: Memory Addressing Modes
Concept #3: Operations in the Instruction Set
Concept #4: Instruction Formats
Concept #5: Encoding the Instruction Set

Concept #1: **Data Storage**

- Storage Architecture
- General Purpose Register Architecture

Storage Architecture: Definition

- Under Von Neumann Architecture:
  - Data (operands) are stored in memory

- For a processor, **storage architecture** concerns with:
  - Where do we store the operands so that the computation can be performed?
  - Where do we store the computation result afterwards?
  - How do we specify the operands?

- Major storage architectures in the next slide

Operands may be implicit or explicit.

Storage Architecture: **Common Design**

- **Stack architecture:**
  - Operands are implicitly on top of the stack.

- **Accumulator architecture:**
  - One operand is implicitly in the accumulator (a register). Examples: IBM 701, DEC PDP-8.

- **General-purpose register architecture:**
  - Only explicit operands.
  - **Register-memory architecture** (one operand in memory). Examples: Motorola 68000, Intel 80386.
  - **Register-register (or load-store) architecture.**
    Examples: MIPS, DEC Alpha.

- **Memory-memory architecture:**
  - All operands in memory. Example: DEC VAX.

Storage Architecture: Example

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register (load-store)</th>
<th>Memory-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Load R1,A</td>
<td>Add C, A, B</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Load R2,B</td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Add R3,R1,R2</td>
<td></td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td>Store R3,C</td>
<td></td>
</tr>
</tbody>
</table>

\[
C = A + B
\]

Storage Architecture: **GPR Architecture**

- For modern processors:
  - General-Purpose Register (GPR) is the most common choice for storage design

  - **RISC** computers typically use **Register-Register (Load/Store)** design
    - E.g. MIPS, ARM, **RISC-V**

  - **CISC** computers use a mixture of Register-Register and Register-Memory
    - E.g. IA32

Concept #2: Memory & Addressing Mode

- Memory Locations and Addresses
- Addressing Modes

Memory Address and Content

- Given $k$-bit address, the address space is of size $2^k$
- Each memory transfer consists of one word of $n$ bits

Addressing Modes

- **Addressing Mode:**
  - Ways to specify an operand in an assembly language

- In RISC-V, there are only 4 addressing modes:
  - **Register:**
    - Operand is in a register
  - **Immediate:**
    -Operand is specified in the instruction directly
  - **Displacement:**
    -Operand is in memory with address calculated as Base + Offset
  - **PC-Relative:**
    -Operand is in memory with address calculated as PC + Offset
# Addressing Modes: Other

<table>
<thead>
<tr>
<th><strong>Addressing mode</strong></th>
<th><strong>Example</strong></th>
<th><strong>Meaning</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Add R4,R3</td>
<td>R4 ← R4+R3</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R4,#3</td>
<td>R4 ← R4+3</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4,100(R1)</td>
<td>R4 ← R4+Mem[100+R1]</td>
</tr>
<tr>
<td>Register indirect</td>
<td>Add R4,(R1)</td>
<td>R4 ← R4+Mem[R1]</td>
</tr>
<tr>
<td>Indexed / Base</td>
<td>Add R3,(R1+R2)</td>
<td>R3 ← R3+Mem[R1+R2]</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R1,(1001)</td>
<td>R1 ← R1+Mem[1001]</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R1,@(R3)</td>
<td>R1 ← R1+Mem[Mem[R3]]</td>
</tr>
<tr>
<td>Auto-increment</td>
<td>Add R1,(R2)+</td>
<td>R1 ← R1+Mem[R2]; R2 ← R2+d</td>
</tr>
<tr>
<td>Auto-decrement</td>
<td>Add R1,–(R2)</td>
<td>R2 ← R2-d; R1 ← R1+Mem[R2]</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1,100(R2)[R3]</td>
<td>R1 ← R1+Mem[100+R2+R3*d]</td>
</tr>
</tbody>
</table>

Concept #3: Operations in Instruction Set

- Standard Operations in an Instruction Set
- Frequently Used Instructions

<table>
<thead>
<tr>
<th>Concept #1: Data Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concept #2: Memory Addressing Modes</td>
</tr>
<tr>
<td><strong>Concept #3: Operations in the Instruction Set</strong></td>
</tr>
<tr>
<td>Concept #4: Instruction Formats</td>
</tr>
<tr>
<td>Concept #5: Encoding the Instruction Set</td>
</tr>
</tbody>
</table>

## Standard Operations

<table>
<thead>
<tr>
<th><strong>Data Movement</strong></th>
<th>load (from memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>store (to memory)</td>
</tr>
<tr>
<td></td>
<td>memory-to-memory move</td>
</tr>
<tr>
<td></td>
<td>register-to-register move</td>
</tr>
<tr>
<td></td>
<td>input (from I/O device)</td>
</tr>
<tr>
<td></td>
<td>output (to I/O device)</td>
</tr>
<tr>
<td></td>
<td>push, pop (to/from stack)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Arithmetic</strong></th>
<th>integer (binary + decimal) or FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>add, subtract, multiply, divide</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Shift</strong></th>
<th>shift left/right, rotate left/right</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Logical</strong></th>
<th>not, and, or, set, clear</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Control flow</strong></th>
<th>Jump (unconditional), Branch (conditional)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Subroutine Linkage</strong></th>
<th>call, return</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Interrupt</strong></th>
<th>trap, return</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Synchronization</strong></th>
<th>test &amp; set (atomic r-m-w)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>String</strong></th>
<th>search, move, compare</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Graphics</strong></th>
<th>pixel and vertex operations, compression/decompression</th>
</tr>
</thead>
</table>

Concept #4: Instruction Formats

- Instruction Length
- Instruction Fields
  - Type and Size of Operands

Concept #1: Data Storage
Concept #2: Memory Addressing Modes
Concept #3: Operations in the Instruction Set
Concept #4: Instruction Formats
Concept #5: Encoding the Instruction Set

Instruction Length

- **Variable-length** instructions.
  - Intel 80x86: Instructions vary from 1 to 17 bytes long.
  - Digital VAX: Instructions vary from 1 to 54 bytes long.
  - Require multi-step fetch and decode.
  - Allow for a more flexible (but complex) and compact instruction set.

- **Fixed-length** instructions.
  - Used in most RISC (Reduced Instruction Set Computers)
  - MIPS, PowerPC: Instructions are 4 bytes long.
  - Allow for easy fetch and decode.
  - Simplify pipelining and parallelism.
  - Instruction bits are scarce.

- **Hybrid** instructions: a mix of variable- and fixed-length instructions.

Instruction Fields

- An instruction consists of
  - **opcode**: unique code to specify the desired operation
  - **operands**: zero or more additional information needed for the operation

- The operation designates the type and size of the operands
  - **Typical type and size**: Character (8 bits), half-word (eg: 16 bits), word (eg: 32 bits), single-precision floating point (eg: 1 word), double-precision floating point (eg: 2 words).

- Expectations from any new 32-bit architecture:
  - Support for 8-, 16- and 32-bit integer and 32-bit and 64-bit floating point operations. A 64-bit architecture would need to support 64-bit integers as well.

## Frequently Used Instructions

<table>
<thead>
<tr>
<th>Rank</th>
<th>Integer Instructions</th>
<th>Average Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load</td>
<td>22%</td>
</tr>
<tr>
<td>2</td>
<td>Conditional Branch</td>
<td>20%</td>
</tr>
<tr>
<td>3</td>
<td>Compare</td>
<td>16%</td>
</tr>
<tr>
<td>4</td>
<td>Store</td>
<td>12%</td>
</tr>
<tr>
<td>5</td>
<td>Add</td>
<td>8%</td>
</tr>
<tr>
<td>6</td>
<td>Bitwise AND</td>
<td>6%</td>
</tr>
<tr>
<td>7</td>
<td>Sub</td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>Move register to register</td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>Procedure call</td>
<td>1%</td>
</tr>
<tr>
<td>10</td>
<td>Return</td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td><strong>96%</strong></td>
</tr>
</tbody>
</table>

Make these instructions fast! Amdahl’s law – make the common case fast!

Concept #5: Encoding the Instruction Set

- Instruction Encoding
- Encoding for Fixed-Length Instructions

<table>
<thead>
<tr>
<th>Concept #1: Data Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concept #2: Memory Addressing Modes</td>
</tr>
<tr>
<td>Concept #3: Operations in the Instruction Set</td>
</tr>
<tr>
<td>Concept #4: Instruction Formats</td>
</tr>
<tr>
<td><strong>Concept #5: Encoding the Instruction Set</strong></td>
</tr>
</tbody>
</table>

Instruction Encoding: Overview

- How are instructions represented in binary format for execution by the processor?

- Issues:
  - Code size, speed/performance, design complexity.

- Things to be decided:
  - Number of registers
  - Number of addressing modes
  - Number of operands in an instruction

- The different competing forces:
  - Have many registers and addressing modes
  - Reduce code size
  - Have instruction length that is easy to handle (fixed-length instructions are easy to handle)

# Encoding Choices

- Three encoding choices: variable, fixed, hybrid.

<table>
<thead>
<tr>
<th>Operation and no. of operands</th>
<th>Address specifier 1</th>
<th>Address field 1</th>
<th>(\ldots)</th>
<th>Address specifier</th>
<th>Address field</th>
</tr>
</thead>
</table>

(a) Variable (e.g., VAX, Intel 80x86)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address field 1</th>
<th>Address field 2</th>
<th>Address field 3</th>
</tr>
</thead>
</table>

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier</th>
<th>Address field</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier 1</th>
<th>Address specifier 2</th>
<th>Address field</th>
</tr>
</thead>
</table>

(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

Fixed Length Instruction: Encoding (1/4)

• Fixed length instruction presents a much more interesting challenge:
  – Q: How to fit multiple sets of instruction types into same number of bits?
  – A: Work with the most constrained instruction types first

• Expanding Opcode scheme:
  – The opcode has variable lengths for different instructions.
  – A good way to maximizes the instruction bits.

Fixed Length Instruction: Encoding (2/4)

• Example:
  – 16-bit fixed length instructions, with 2 types of instructions
  – Type-A: 2 operands, each operand is 5-bit
  – Type-B: 1 operand of 5-bit

<table>
<thead>
<tr>
<th>Type</th>
<th>opcode</th>
<th>operand</th>
<th>operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-A</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
</tr>
<tr>
<td>Type-B</td>
<td>6 bits</td>
<td>5 bits</td>
<td>unused</td>
</tr>
</tbody>
</table>

Problem:
- Wasted bits in Type-B instruction
- Maximum total number of instructions is $2^6$ or 64.

Fixed Length Instruction: Encoding (3/4)

- Use **expanding opcode** scheme:
  - Extend the opcode for type-B instructions to **11 bits**
  - No wasted bits and result in a larger instruction set

Questions:
- How do we distinguish between Type-A and Type-B?
- How many different instructions do we really have?

Fixed Length Instruction: Encoding (4/4)

- What is the maximum number of instructions?

**Answer:**

\[ 1 + (2^6 - 1) \times 2^5 = 1 + 63 \times 32 = 2017 \]

**Reasoning:**

1. For every 6-bit prefix (front-part) given to Type-B, we get \(2^5\) unique patterns, e.g. \([111111]xxxxx\)

2. So, we should minimize Type-A instruction and give as many 6-bit prefixes as possible to Type-B

   \(\Rightarrow\) 1 Type-A instruction, \(2^6 - 1\) prefixes for Type-B

Why RISC-V

• Open and free

• Not domain-specific

• To keep things simple, flexible and extensible

• No baggage of legacy
RISC-V ISA manuals

• User level – Volume 1

• Privileged level – Volume 2
RISC-V ISA Design Principle-1

• *Design Principle 1*: Simplicity favours regularity
  – Regularity makes implementation simpler
  – Simplicity enables higher performance at lower cost

• E.g. All arithmetic operations have same form
  – Two sources and one destination

  ```plaintext
  add a, b, c  // a gets b + c
  ```
RISC-V ISA Design Principle-2

• *Design Principle 2*: Smaller is faster
  – memory is larger than no. of registers, use register operands

• E.g. Arithmetic operations use register operands and not direct memory

• most implementations have decoding the operands on the critical path so only 32 registers
RISC-V ISA Design Principle-3

• *Design Principle 3:* Make the common case fast
  – Small constants are common
  – Immediate operand avoids a load instruction

• support for immediate operands,
• e.g. `addi x22, x22, 4`
RISC-V ISA Design Principle-4

• *Design Principle 4*: Good design demands good compromises
  – Different formats complicate decoding, but allow 32-bit instructions uniformly
  – Keep formats as similar as possible

• E.g. R-format and I-format, I-format versus S-format
Instruction Encoding

- Variable length encoding supported
- Base-ISA: 32-bits
R-format Instruction

<table>
<thead>
<tr>
<th>funct7</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>3 bits</td>
<td>5 bits</td>
<td>7 bits</td>
</tr>
</tbody>
</table>

- **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

<table>
<thead>
<tr>
<th>funct7</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>3 bits</td>
<td>5 bits</td>
<td>7 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>21</th>
<th>20</th>
<th>0</th>
<th>9</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000000</td>
<td>10101</td>
<td>10100</td>
<td>000</td>
<td>01001</td>
<td>0110011</td>
</tr>
</tbody>
</table>

0000 0001 0101 1010 0000 0100 1011 0011_{two} = 015A04B3_{16}
R-format Instructions

• Shift operations (logical and arithmetic)
  – SLL, SRL, SRA (why no SLA ?)

• Arithmetic operations
  – ADD, SUB

• Logical operations
  – XOR, OR, AND (missing NOT ?)

• Compare operations
  – SLT, SLTU (what is a good implementation of SLTU?)
I-format Instruction

<table>
<thead>
<tr>
<th>immediate</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 bits</td>
<td>5 bits</td>
<td>3 bits</td>
<td>5 bits</td>
<td>7 bits</td>
</tr>
</tbody>
</table>

- 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
I-format Instructions

- Loads: LB, LH, LW, LBU, LHU *(why not stores ?)*
- Shifts: SLLI
- Arithmetic: ADDI *(why not sub ?)*
- Logical: XORI, ORI, ANDI
- Compare: SLTI, SLTIU
- System call and break, Sync threads, Counters

3-Jul-18 CASS18 - ISA and RISCV 55
S-format Instruction

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bits</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

• 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

• Stores: SB, SH, SW
U-format Instruction

<table>
<thead>
<tr>
<th>immediate</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 bits</td>
<td>5 bits</td>
<td>7 bits</td>
</tr>
</tbody>
</table>

- Why is this separate format needed
- How to load a 32 bit constant into a register?
  - Rd [31:12] == immediate[19:0]
  - Rd [11:0] == 12'b0
- Load upper immediate (LUI)
- Add upper immediate to PC (AUIPC)
Other instruction formats

• What is missing?

• NOP?

• Is the above list complete?

• Control flow instructions
## SB-format Instruction

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>3 bits</td>
<td>4 bits</td>
<td>1 bit</td>
<td>7 bits</td>
</tr>
</tbody>
</table>

- Why different immediate format for branch instructions
- What about imm[0]?
- Branches: BEQ, BNE, BLT, BGE, BLTU, BGEU
- What about overflows?
Why different immediate format for jump?
What about imm[0]?
JAL – jump and link
What about JALR (jump and link return?)
   – I-type format, Why?
Addressing Modes

1. Immediate addressing
   immediate rs1 funct3 rd op

2. Register addressing
   funct7 rs2 rs1 funct3 rd op

3. Base addressing
   immediate rs1 funct3 rd op

4. PC-relative addressing
   imm rs2 rs1 funct3 imm op

Registers
- Register
- Memory
  - [Byte]
  - Halfword
  - Word
  - Doubleword

3-Jul-18 CASS18 - ISA and RISCV 61
## Types of Immediate

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>10</th>
<th>5</th>
<th>4</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>20</td>
<td>19</td>
<td>12</td>
<td>11</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>-----------------</td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>20</td>
<td>19</td>
<td>12</td>
<td>11</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>-----------------</td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>20</td>
<td>19</td>
<td>12</td>
<td>11</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>-----------------</td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>20</td>
<td>19</td>
<td>12</td>
<td>11</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>-----------------</td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>20</td>
<td>19</td>
<td>12</td>
<td>11</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
32-bit Constants

• Most constants are small
  – 12-bit immediate is sufficient

• For the occasional 32-bit constant
  \texttt{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

\begin{verbatim}
lui x19, 976  // 0x003D0
\end{verbatim}

\begin{verbatim}
addi x19, x19, 128  // 0x500
\end{verbatim}
Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward
- SB format:
  - PC-relative addressing
    - Target address = PC + immediate × 2
Jump Addressing

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

- UJ format:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>5 bits</td>
<td>7 bits</td>
</tr>
</tbody>
</table>

- 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
References

• RISC-V User-level ISA specification
  https://riscv.org/specifications/