Programmable Machines

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.
6.S084 So Far

- Sequential Logic
- Combinational Logic
- CMOS Gates
- Transistors

March 8, 2018
What can you do with these?
- Design special-purpose hardware: a finite state machine that solves a particular problem
• What can you do with these?
  – Design *special-purpose hardware*: a finite state machine that solves a particular problem

• What you’ll be able to do soon:
  – Design a *general-purpose computer*: a machine that can solve any solvable problem, given enough time and memory
The von Neumann Model

- Many ways to build a general-purpose computer
- Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
The von Neumann Model

• Many ways to build a general-purpose computer
• Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
• Components:
The von Neumann Model

• Many ways to build a general-purpose computer
• Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
• Components:

  Main Memory

  - Main memory holds programs and their data
The von Neumann Model

- Many ways to build a general-purpose computer
- Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
- Components:
  - **Main memory** holds programs and their data
  - **Central processing unit** accesses and processes memory values
The von Neumann Model

- Many ways to build a general-purpose computer
- Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
- Components:
  - Main memory holds programs and their data
  - Central processing unit accesses and processes memory values
  - Input/output devices to communicate with the outside world
Main Memory = Random-Access Memory

• Array of bits, organized in $W$ words of $N$ bits each
  - Typically, $W$ is a power of two: $W = 2^k$
  - Example: $W=8$ ($k=3$ bits), $N=32$ bits per word

<table>
<thead>
<tr>
<th>Address</th>
<th>11101000 10110100 01011010 10010101</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>10111010 00000000 11110101 00000000</td>
</tr>
<tr>
<td>001</td>
<td>01101101 00000000 11110101 00000000</td>
</tr>
<tr>
<td>010</td>
<td>01101101 00000000 11110101 00000000</td>
</tr>
<tr>
<td>011</td>
<td>01101101 00000000 11110101 00000000</td>
</tr>
<tr>
<td>100</td>
<td>01101101 00000000 11110101 00000000</td>
</tr>
<tr>
<td>101</td>
<td>01101101 00000000 11110101 00000000</td>
</tr>
<tr>
<td>110</td>
<td>01101101 00000000 11110101 00000000</td>
</tr>
<tr>
<td>111</td>
<td>01101101 00000000 11110101 11011000</td>
</tr>
</tbody>
</table>
Main Memory = Random-Access Memory

- Array of bits, organized in $W$ words of $N$ bits each
  - Typically, $W$ is a power of two: $W = 2^k$
  - Example: $W=8$ ($k=3$ bits), $N=32$ bits per word

- Can read from and write to individual words

<table>
<thead>
<tr>
<th>Address</th>
<th>11101000 10111010 01011010 10010101</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>10111010 00000000 11110101 00000000</td>
</tr>
<tr>
<td>001</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>00000000 00000000 11110101 11011000</td>
</tr>
</tbody>
</table>
Main Memory = Random-Access Memory

- Array of bits, organized in $W$ words of $N$ bits each
  - Typically, $W$ is a power of two: $W = 2^k$
  - Example: $W=8$ ($k=3$ bits), $N=32$ bits per word

- Can read from and write to individual words

- Many possible implementations (later in the course)

<table>
<thead>
<tr>
<th>Address</th>
<th>11101000 10111010 01011010 10010101</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>10111010 00000000 11110101 00000000</td>
</tr>
<tr>
<td>010</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>00000000 00000000 11110101 11011000</td>
</tr>
</tbody>
</table>
Key Idea: Stored-Program Computer

- Express program as a sequence of **coded instructions**
- Memory holds both data and instructions
- CPU fetches, interprets, and executes successive instructions of the program
Key Idea: Stored-Program Computer

- Express program as a sequence of coded instructions
- Memory holds both data and instructions
- CPU fetches, interprets, and executes successive instructions of the program
Key Idea: Stored-Program Computer

- Express program as a sequence of **coded instructions**
- Memory holds both data and instructions
- CPU fetches, interprets, and executes successive instructions of the program

```
rs2 | rs1 | rd | op
---|-----|----|-----
rd ← op(rs1, rs2)
```
Key Idea: Stored-Program Computer

- Express program as a sequence of **coded instructions**
- Memory holds both data and instructions
- CPU fetches, interprets, and executes successive instructions of the program
Key Idea: Stored-Program Computer

- Express program as a sequence of **coded instructions**
- Memory holds both data and instructions
- CPU fetches, interprets, and executes successive instructions of the program

How does CPU distinguish between instructions and data?
Anatomy of a von Neumann Computer
Anatomy of a von Neumann Computer

- Datapath
- Control Unit
- Internal storage
- Main Memory

Control
Status
Address
Data
Instructions
Address

March 8, 2018 L09-6
Anatomy of a von Neumann Computer

Datapath

Control Unit

Main Memory

- Internal storage
- Datapath
- Control
- Status
- Address
- Data
- Instructions
- ALU
- Cc's
- Dest
- Asel
- Bsel
- Fr

March 8, 2018
Anatomy of a von Neumann Computer

Datapath

Control Unit

Main Memory

Internal storage

address

data

control

status

address

instructions

dest

registers

asel

bsel

fr

ALU

Cc’s

March 8, 2018
Anatomy of a von Neumann Computer

Datapath

Control Unit

Main Memory

registers

operations

Internal storage

address

data

control

status

address

instructions

dest

asesel

bsel

fn

ALU

cc’s

March 8, 2018
Anatomy of a von Neumann Computer

- **Instructions** coded as binary data
- **Program Counter** or PC: Address of the instruction to be executed
- Logic to translate instructions into control signals for datapath
Instructions

• Instructions are the fundamental unit of work
Instructions

- Instructions are the fundamental unit of work
- Each instruction specifies:
  - An operation or opcode to be performed
  - Source and destination operands
Instructions

- Instructions are the fundamental unit of work
- Each instruction specifies:
  - An operation or opcode to be performed
  - Source and destination operands

- A von Neumann machine executes instructions sequentially
  - CPU logically implements this loop:
Instructions

- Instructions are the fundamental unit of work
- Each instruction specifies:
  - An operation or opcode to be performed
  - Source and destination operands

- A von Neumann machine executes instructions sequentially
  - CPU logically implements this loop:
  - By default, the next PC is current PC + size of current instruction unless the instruction says otherwise

March 8, 2018
Instruction Set Architecture (ISA)

- ISA: The contract between software and hardware
  - Functional definition of operations and storage locations
  - Precise description of how software can invoke and access them
Instruction Set Architecture (ISA)

• ISA: The contract between software and hardware
  – Functional definition of operations and storage locations
  – Precise description of how software can invoke and access them

• The ISA is a new layer of abstraction:
  – ISA specifies what hardware provides, not how it’s implemented
Instruction Set Architecture (ISA)

- ISA: The contract between software and hardware
  - Functional definition of operations and storage locations
  - Precise description of how software can invoke and access them

- The ISA is a new layer of abstraction:
  - ISA specifies what hardware provides, not how it’s implemented
  - Hides the complexity of CPU implementation
Instruction Set Architecture (ISA)

• **ISA**: The contract between software and hardware
  - Functional definition of operations and storage locations
  - Precise description of how software can invoke and access them

• The ISA is a new layer of abstraction:
  - ISA specifies what hardware provides, not how it’s implemented
  - Hides the complexity of CPU implementation
  - Enables fast innovation in hardware (no need to change software!)
    - 8086 (1978): 29 thousand transistors, 5 MHz, 0.33 MIPS
    - Pentium 4 (2003): 44 million transistors, 4 GHz, ~5000 MIPS
    - Skylake (2015): 1.75 billion transistors, 4 GHz, ~30k MIPS
    - All implement the x86 ISA
Instruction Set Architecture (ISA)

• ISA: The contract between software and hardware
  – Functional definition of operations and storage locations
  – Precise description of how software can invoke and access them

• The ISA is a new layer of abstraction:
  – ISA specifies what hardware provides, not how it’s implemented
  – Hides the complexity of CPU implementation
  – Enables fast innovation in hardware (no need to change software!)
    • 8086 (1978): 29 thousand transistors, 5 MHz, 0.33 MIPS
    • Pentium 4 (2003): 44 million transistors, 4 GHz, ~5000 MIPS
    • Skylake (2015): 1.75 billion transistors, 4 GHz, ~30k MIPS
    • All implement the x86 ISA
  – Dark side: Commercially successful ISAs last for decades
    • Today’s x86 CPUs carry baggage of design decisions from 70’s
Instruction Set Architecture Design

• Designing an ISA is hard:
  - How many operations?
  - What types of storage, how much?
  - How to encode instructions?
  - How to future-proof?
Designing an ISA is hard:
- How many operations?
- What types of storage, how much?
- How to encode instructions?
- How to future-proof?

How to decide? Take a quantitative approach
- Take a set of representative benchmark programs
- Evaluate versions of your ISA and implementation with and without feature
- Pick what works best overall (performance, energy, area...)
Instruction Set Architecture Design

• Designing an ISA is hard:
  – How many operations?
  – What types of storage, how much?
  – How to encode instructions?
  – How to future-proof?

• How to decide? Take a quantitative approach
  – Take a set of representative benchmark programs
  – Evaluate versions of your ISA and implementation with and without feature
  – Pick what works best overall (performance, energy, area…)

• Corollary: Optimize the common case
Instruction Set Architecture Design

• Designing an ISA is hard:
  – How many operations?
  – What types of storage, how much?
  – How to encode instructions?
  – How to future-proof?

• How to decide? Take a quantitative approach
  – Take a set of representative benchmark programs
  – Evaluate versions of your ISA and implementation with and without feature
  – Pick what works best overall (performance, energy, area...)

• Corollary: Optimize the common case

In this course we will use the RISC-V ISA
RISC-V

- A new, open, free ISA from Berkeley
RISC-V

• A new, open, free ISA from Berkeley

• Several variants
  – RV32, RV64, RV128: Different data widths
  – ‘I’: Base Integer instructions
  – ‘M’: Multiply and Divide
  – ‘F’ and ‘D’: Single- and Double-precision floating point
  – And many other modular extensions
RISC-V

• A new, open, free ISA from Berkeley

• Several variants
  – RV32, RV64, RV128: Different data widths
  – ‘I’: Base Integer instructions
  – ‘M’: Multiply and Divide
  – ‘F’ and ‘D’: Single- and Double-precision floating point
  – And many other modular extensions

• We will design an RV32I processor, which is the base 32-bit variant
## RISC-V ISA: Storage

### CPU State

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td></td>
</tr>
</tbody>
</table>

### General-Purpose Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>32-bit “words”</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>000000...0</td>
</tr>
<tr>
<td>x1</td>
<td></td>
</tr>
<tr>
<td>x2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>x31</td>
<td></td>
</tr>
</tbody>
</table>
RISC-V ISA: Storage

CPU State

PC

General-Purpose Registers

x0 000000...0
x1
x2
:: 32-bit "words"
x31

x0 hardwired to 0
RISC-V ISA: Storage

**CPU State**

- **PC**
  - General-Purpose Registers
    - x0
    - x1
    - x2
    - ...
    - x31
  - x0 hardwired to 0

**Main Memory**

- Up to $2^{32}$ bytes (4GB) of memory, organized as $2^{30}$ 4-byte words

- 32-bit “words”
  - 4 bytes
  - 32-bit “words”

March 8, 2018
RISC-V ISA: Storage

**CPU State**
- **PC**
- General-Purpose Registers:
  - x0
  - x1
  - x2
  - ...
  - x31

**Main Memory**
- Up to $2^{32}$ bytes (4GB) of memory, organized as $2^{30}$ 4-byte words

Each memory word is 32-bits wide, but we use byte memory addresses. Since each word contains 4 bytes, addresses of consecutive words differ by 4.

- x0 hardwired to 0

Addresses:
- 0x00
- 0x04
- 0x08
- 0x0C
- 0x10
- 0x12

- 32-bit “words” (4 bytes)
Why have separate registers and main memory?
RISC-V ISA: Storage

Why have separate registers and main memory? Small set of registers is much cheaper to access

CPU State

Main Memory

Up to $2^{32}$ bytes (4GB) of memory, organized as $2^{30}$ 4-byte words

Each memory word is 32-bits wide, but we use byte memory addresses. Since each word contains 4 bytes, addresses of consecutive words differ by 4.

PC

General-Purpose Registers

x0

000000...0

x1

x2

... 32-bit “words”

x31

0x00

0x04

0x08

0x0C

0x10

0x12

32-bit “words” (4 bytes)

x0 hardwired to 0

March 8, 2018
RISC-V ISA: Instructions

• Three types of instructions:
  – Computational: Perform operations on general registers
  – Loads and stores: Move data between general registers and main memory
  – Control: Change the program counter
RISC-V ISA: Instructions

• Three types of instructions:
  – Computational: Perform operations on general registers
  – Loads and stores: Move data between general registers and main memory
  – Control: Change the program counter

• All instructions have a fixed 32-bit length (4 bytes)
RISC-V ISA: Instructions

- Three types of instructions:
  - Computational: Perform operations on general registers
  - Loads and stores: Move data between general registers and main memory
  - Control: Change the program counter

- All instructions have a fixed 32-bit length (4 bytes)

  7-bit opcode determines how we interpret the remaining 25 bits
RISC-V ISA: Instructions

- Three types of instructions:
  - Computational: Perform operations on general registers
  - Loads and stores: Move data between general registers and main memory
  - Control: Change the program counter

- All instructions have a fixed 32-bit length (4 bytes)

  7-bit opcode determines how we interpret the remaining 25 bits

  Why fixed instead of variable-length instructions?
RISC-V ISA: Instructions

• Three types of instructions:
  – Computational: Perform operations on general registers
  – Loads and stores: Move data between general registers and main memory
  – Control: Change the program counter

• All instructions have a fixed 32-bit length (4 bytes)

  7-bit opcode determines how we interpret the remaining 25 bits

Why fixed instead of variable-length instructions?

Simpler to decode & to compute next PC (but larger code)
Computational Instructions (R-type)

- **Register-to-register instructions (R-type)**

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

- Performs $R[rd] \leftarrow R[rs1] \ op \ R[rs2]$
- Operation $op$ specified by $(\text{funct7}, \text{funct3})$
Computational Instructions (R-type)

- Register-to-register instructions (R-type)

<table>
<thead>
<tr>
<th>funct7</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
</table>

- Performs $R[rd] \leftarrow R[rs1] \ operation \ R[rs2]$
- Operation $\ operation$ specified by (funct7, funct3)
- Example: ADD instruction
Computational Instructions (R-type)

- Register-to-register instructions (R-type)
  
  \[
  \begin{array}{ccccccc}
  7 & 5 & 5 & 3 & 3 & 7 \\
  \text{funct7} & \text{rs2} & \text{rs1} & \text{funct3} & \text{rd} & \text{opcode} \\
  \end{array}
  \]

  - Performs \( R[rd] \leftarrow R[rs1] \ op \ R[rs2] \)
  - Operation \( op \) specified by \( (\text{funct7}, \text{funct3}) \)
  - Example: ADD instruction

  \[
  \begin{array}{ccccccccccccc}
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1
  \end{array}
  \]
Computational Instructions (R-type)

- Register-to-register instructions (R-type)

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

- Performs \( R[rd] \leftarrow R[rs1] \ op \ R[rs2] \)
- Operation \( op \) specified by (funct7, funct3)
- Example: ADD instruction

```
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 0 1 1 1 1 0 1 1
```
Computational Instructions (R-type)

- Register-to-register instructions (R-type)

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

- Performs $R[rd] \leftarrow R[rs1] \ op \ R[rs2]$
- Operation $\ op$ specified by (funct7, funct3)
- Example: ADD instruction

```
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 0 1 1 1 0 1 1
```

opcode = OP
Register-to-register instructions (R-type)

- Performs $R[rd] \leftarrow R[rs1] \text{ op } R[rs2]$
- Operation $\text{op}$ specified by $(\text{funct7, funct3})$
- Example: ADD 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</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

opcode = OP

$\text{op} = \text{add}$

rd = 1
Computational Instructions (R-type)

- Register-to-register instructions (R-type)

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

- Performs $R[rd] \leftarrow R[rs1] \text{ op } R[rs2]$
- Operation $\text{op}$ specified by (funct7, funct3)
- Example: ADD instruction

opcode = OP

op = add

rs1 = 3

rd = 1

March 8, 2018
## Computational Instructions (R-type)

- **Register-to-register instructions (R-type)**

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

- Performs $R[rd] \leftarrow R[rs1] \text{ op } R[rs2]$
- Operation $\text{op}$ specified by $(\text{funct7}, \text{funct3})$
- Example: ADD instruction

- $\text{opcode} = \text{OP}$
- $\text{op} = \text{add}$
- $\text{rs2} = 4$
- $\text{rs1} = 3$
- $\text{rd} = 1$
### Computational Instructions (R-type)

- **Register-to-register instructions (R-type)**
  - Performs $R[rd] \leftarrow R[rs1] \ op \ R[rs2]$
  - Operation $\ op$ specified by $(\text{funct7, funct3})$
  - Example: ADD instruction

\[
\begin{array}{cccccc}
\text{funct7} & \text{rs2} & \text{rs1} & \text{funct3} & \text{rd} & \text{opcode} \\
\hline
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
\end{array}
\]

- $\text{op} = \text{add}$
- $\text{rs2} = 4$
- $\text{rs1} = 3$
- $\text{rd} = 1$

# Computational Instructions (R-type)

- **Register-to-register instructions (R-type)**

  - Performs $R[rd] \leftarrow R[rs1] \text{ op } R[rs2]$
  - Operation $\text{op}$ specified by $(\text{funct7, funct3})$
  - Example: ADD instruction

  - We prefer a **symbolic representation**: `add x1, x3, x4`

---

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
</table>

- $\text{op} = \text{add}$
- $\text{opcode} = \text{OP}$

- $\text{rs2} = 4, \text{rs1} = 3, \text{rd} = 1$

Computational Instructions (R-type)

- **Register-to-register instructions (R-type)**
  - Performs $R[rd] \leftarrow R[rs1] \text{ op } R[rs2]$
  - Operation $\text{op}$ specified by $(\text{funct7, funct3})$
  - Example: ADD instruction
    - We prefer a symbolic representation: $\text{add x1, x3, x4}$
    - Similar instructions for other operations:

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Comparisons</th>
<th>Logical</th>
<th>Shifts</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD, SUB</td>
<td>SLT, SLTU</td>
<td>AND, OR, XOR</td>
<td>SLL, SLR, SRA</td>
</tr>
</tbody>
</table>

March 8, 2018
Computational Instructions (I-type)

- Many programs use small constants frequently (e.g., comparisons, loop indices, etc.)
- Using registers to hold these constants is wasteful!
Computational Instructions (I-type)

- Many programs use small constants frequently (e.g., comparisons, loop indices, etc.)
- Using registers to hold these constants is wasteful!
- Solution: Register-immediate instructions (I-type)

<table>
<thead>
<tr>
<th>12</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
</tr>
</tbody>
</table>

- Performs R[rd] ← R[rs1] op imm
Computational Instructions (I-type)

- Many programs use small constants frequently (e.g., comparisons, loop indices, etc.)
- Using registers to hold these constants is wasteful!
- Solution: Register-immediate instructions (I-type)

<table>
<thead>
<tr>
<th>12</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
</tr>
</tbody>
</table>

- Performs $R[rd] \leftarrow R[rs1] \text{ op } \text{imm}$
- *Immediate* operand $\text{imm} = \text{signExtend}(\text{imm}[11:0])$
Computational Instructions (I-type)

- Many programs use small constants frequently (e.g., comparisons, loop indices, etc.)
- Using registers to hold these constants is wasteful!
- Solution: Register-immediate instructions (I-type)

<table>
<thead>
<tr>
<th></th>
<th>12</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
</tbody>
</table>

- Performs R[rd] ↔ R[rs1] op imm
- *Immediate* operand imm = signExtend(imm[11:0])
- Operation *op* specified by funct3
Computational Instructions (I-type)

- Many programs use small constants frequently (e.g., comparisons, loop indices, etc.)
- Using registers to hold these constants is wasteful!
- Solution: Register-immediate instructions (I-type)

<table>
<thead>
<tr>
<th>imm[11:0]</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

- Performs R[rd] ← R[rs1] op imm
- Immediate operand imm = signExtend(imm[11:0])
- Operation op specified by funct3
- Example: addi x5, x2, -3
Computational Instructions (I-type)

• Many programs use small constants frequently (e.g., comparisons, loop indices, etc.)
• Using registers to hold these constants is wasteful!
• Solution: Register-immediate instructions (I-type)

- Performs \( R[rd] \leftarrow R[rs1] \ op \ imm \)
- \textit{Immediate} operand \( imm = \text{signExtend}(imm[11:0]) \)
- Operation \( op \) specified by \( \text{funct3} \)
- Example: \texttt{addi x5, x2, -3}

\[
\begin{array}{c|c|c|c|c}
12 & 5 & 3 & 5 & 7 \\
\hline
\text{imm}[11:0] & rs1 & \text{funct3} & \text{rd} & \text{opcode} \\
\end{array}
\]

- \( op = \text{add} \)
- \( \text{opcode} = \text{OP-IMM} \)

March 8, 2018
Computational Instructions (I-type)

- Many programs use small constants frequently (e.g., comparisons, loop indices, etc.)
- Using registers to hold these constants is wasteful!
- Solution: Register-immediate instructions (I-type)

```
<table>
<thead>
<tr>
<th>imm[11:0]</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- Performs R[rd] ← R[rs1] \( op \) imm
- **Immediate** operand \( imm = \text{signExtend}(imm[11:0]) \)
- Operation \( op \) specified by funct3
- Example: \( \text{addi} \) \( x5, x2, -3 \)

\[ \begin{array}{cccccc}
  & 12 & 5 & 3 & 5 & 7 \\
  \text{imm}[11:0] & \text{rs1} & \text{funct3} & \text{rd} & \text{opcode} \\
  \hline
  & 12 & 5 & 3 & 5 & 7 \\
  \end{array} \]

- \( \text{op} = \text{add} \)
- \( \text{opcode} = \text{OP-IMM} \)
Computational Instructions (I-type)

- Many programs use small constants frequently (e.g., comparisons, loop indices, etc.)
- Using registers to hold these constants is wasteful!
- Solution: Register-immediate instructions (I-type)

<table>
<thead>
<tr>
<th>12</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
</tr>
</tbody>
</table>

- Performs $R[rd] \leftarrow R[rs1] \text{ op } \text{imm}$
- *Immediate* operand $\text{imm} = \text{signExtend}(\text{imm}[11:0])$
- Operation $\text{op}$ specified by $\text{funct3}$
- Example: $\text{addi } x5, x2, -3$

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

$\text{op} = \text{add}$  $\text{opcode} = \text{OP-IMM}$
Computational Instructions (I-type)

- Many programs use small constants frequently (e.g., comparisons, loop indices, etc.)
- Using registers to hold these constants is wasteful!
- Solution: Register-immediate instructions (I-type)

- Performs \( R[rd] \leftarrow R[rs1] \ op \ imm \)
- Immediate operand \( imm = \text{signExtend}(imm[11:0]) \)
- Operation \( op \) specified by funct3
- Example: \( \text{addi x5, x2, -3} \)

<table>
<thead>
<tr>
<th>imm[11:0]</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x0</td>
<td>0x0</td>
<td>0x0</td>
<td>0x11</td>
</tr>
</tbody>
</table>

\( \text{opcode} = \text{OP-IMM} \)
\( \text{op} = \text{add} \)
Computational Instructions (I-type)

- Similar set of operations as R-type instructions:

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Comparisons</th>
<th>Logical</th>
<th>Shifts</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI</td>
<td>SLTI, SLTIU</td>
<td>ANDI, ORI, XORI</td>
<td>SLLI, SLRI, SRAI</td>
</tr>
</tbody>
</table>
Computational Instructions (I-type)

- Similar set of operations as R-type instructions:

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Comparisons</th>
<th>Logical</th>
<th>Shifts</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI</td>
<td>SLTI, SLTIU</td>
<td>ANDI, ORI, XORI</td>
<td>SLLI, SLRI, SRAI</td>
</tr>
</tbody>
</table>

Why is there no SUBI?
Computational Instructions (I-type)

- Similar set of operations as R-type instructions:

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Comparisons</th>
<th>Logical</th>
<th>Shifts</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI</td>
<td>SLTI, SLTIU</td>
<td>ANDI, ORI</td>
<td>SLLI, SLRI, SRAI</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XORI</td>
<td></td>
</tr>
</tbody>
</table>

Why is there no SUBI?

*Equivalent to ADDI with a negative immediate*
Computational Instructions (I-type)

- Similar set of operations as R-type instructions:

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Comparisons</th>
<th>Logical</th>
<th>Shifts</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI</td>
<td>SLTI, SLTIU</td>
<td>ANDI, ORI, XORI</td>
<td>SLLI, SLRI, SRAI</td>
</tr>
</tbody>
</table>

Why is there no SUBI?

Equivalent to ADDI with a negative immediate

- Shift instructions use a slightly different encoding
  - R[rd] ← R[rs1] op imm[4:0]
  - Operation op specified by (funct3, inst[30])
Computational Instructions (I-type)

- Similar set of operations as R-type instructions:

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Comparisons</th>
<th>Logical</th>
<th>Shifts</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI</td>
<td>SLTI, SLTIU</td>
<td>ANDI, ORI, XORI</td>
<td>SLLI, SLRI, SRAI</td>
</tr>
</tbody>
</table>

Why is there no SUBI?

Equivalent to ADDI with a negative immediate

- Shift instructions use a slightly different encoding
  - R[rd] ← R[rs1] op imm[4:0]
  - Operation op specified by (funct3, inst[30])
  - This way we can have 9 instructions with 3-bit funct3
Computational Instructions (U-type)

- Sometimes we need to use full 32-bit constants
Computational Instructions (U-type)

- Sometimes we need to use full 32-bit constants
- Solution: Add a format with a 20-bit immediate

\[
\begin{array}{c|c|c}
\text{imm}[31:12] & 5 & 7 \\
\hline
\text{rd} & \text{opcode} & 20 \\
\end{array}
\]

- \( \text{imm} = \{\text{inst}[31:12], 12\text{'}b0\} \)

- Load upper immediate (LUI): \( \text{lui} \ rd, \ \text{imm} : \ R[rd] \leftarrow \text{imm} \)
Computational Instructions (U-type)

- Sometimes we need to use full 32-bit constants
- Solution: Add a format with a 20-bit immediate

\[
\begin{array}{c|c|c}
20 & 5 & 7 \\
\text{imm}[31:12] & \text{rd} & \text{opcode}
\end{array}
\]

- \( \text{imm} = \{\text{inst}[31:12], 12\text{'}b0\} \)

- Load upper immediate (LUI): \( \text{lui } \text{rd}, \text{imm} : \text{R}[\text{rd}] \leftarrow \text{imm} \)

- Example: Write code to load constant 0xCAFE0123 into x3
### Computational Instructions (U-type)

- Sometimes we need to use full 32-bit constants
- Solution: Add a format with a 20-bit immediate

<table>
<thead>
<tr>
<th></th>
<th>20</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[31:12]</td>
<td></td>
<td>rd</td>
<td>opcode</td>
</tr>
</tbody>
</table>

- \( \text{imm} = \{\text{inst}[31:12], 12'b0\} \)

- Load upper immediate (LUI): \( \text{lui rd, imm : } R[rd] \leftarrow \text{imm} \)

- **Example:** Write code to load constant \( 0x\text{CAFE}0123 \) into \( x3 \)

  \[
  \text{lui x3, 0x\text{CAFE}0}\n  \]
Computational Instructions (U-type)

- Sometimes we need to use full 32-bit constants
- Solution: Add a format with a 20-bit immediate

\[
\begin{array}{c|c|c}
20 & 5 & 7 \\ 
\hline 
\text{imm}[31:12] & \text{rd} & \text{opcode} \\
\end{array}
\]

- \( \text{imm} = \{\text{inst}[31:12], 12'b0\} \)

- Load upper immediate (LUI): \( \text{lui } rd, \text{ imm : R}[rd] \leftarrow \text{imm} \)

- Example: Write code to load constant 0xCAFE0123 into x3

\[
\begin{align*}
\text{lui } x3, 0x\text{CAFE0} \\
\text{addi } x3, x3, 0x123 \\
\end{align*}
\]
Load and Store Instructions

- Loads and stores move data between registers and main memory
Load and Store Instructions

• Loads and stores move data between registers and main memory

• Load (I-type):
Load and Store Instructions

- Loads and stores move data between registers and main memory

- Load (I-type):

```
<table>
<thead>
<tr>
<th>12</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
</tr>
</tbody>
</table>
```
Load and Store Instructions

- Loads and stores move data between registers and main memory

- Load (I-type):

  \[
  \text{imm} = \text{signExtend}(\text{inst}[31:20])
  \]
Load and Store Instructions

- Loads and stores move data between registers and main memory

- **Load (I-type):**

<table>
<thead>
<tr>
<th>12</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
</tr>
</tbody>
</table>

  - \(\text{imm} = \text{signExtend}(\text{inst}[31:20])\)
  
  - Load word: \(\text{lw \ rd, imm(rs1)}: \text{R}[\text{rd}] \leftarrow \text{Mem}[\text{R}[\text{rs1}] + \text{imm}]\)
Load and Store Instructions

- Loads and stores move data between registers and main memory

- Load (I-type):
  - imm = signExtend(inst[31:20])
  - Load word: lw rd, imm(rs1) : R[rd] ← Mem[R[rs1] + imm]

<table>
<thead>
<tr>
<th>imm[11:0]</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

Memory address
Load and Store Instructions

- Loads and stores move data between registers and main memory

- Load (I-type):

<table>
<thead>
<tr>
<th>12</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
</tr>
</tbody>
</table>

  - imm = signExtend(inst[31:20])

  - Load word: lw rd, imm(rs1) : R[rd] ← Mem[R[rs1] + imm]

  - Example: lw x2, 4(x3)

Memory address
Load and Store Instructions

- Loads and stores move data between registers and main memory

- Load (I-type):

  \[
  \text{imm} = \text{signExtend}((\text{inst}[31:20]))
  \]

  \[
  \text{Load word: } \text{lw } \text{rd, imm(rs1)} : \text{R[rd]} \leftarrow \text{Mem[R[rs1] + imm]}
  \]

  - Example: lw x2, 4(x3)

  - RISC-V has a few other load instructions (load half-word, load byte, etc.) that we will not use in this course
Load and Store Instructions

- **Store (S-type):**

  \[
  \text{imm} = \text{signExtend}(\{\text{inst}[31:25], \text{inst}[11:7]\})
  \]

<table>
<thead>
<tr>
<th>7</th>
<th>5</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
</table>
Load and Store Instructions

- **Store (S-type):**

  - `imm = signExtend({inst[31:25], inst[11:7]})`

  - **Store word:**
    
    ```
    sw imm(rs1), rs2 : Mem[R[rs1] + imm] ← R[rs2]
    ```
Load and Store Instructions

- **Store (S-type):**
  
  \[
  \begin{array}{cccccc}
  7 & 5 & 5 & 3 & 5 & 7 \\
  \text{imm}[11:5] & rs2 & rs1 & \text{funct3} & \text{imm}[4:0] & \text{opcode} \\
  \end{array}
  \]

  - \( \text{imm} = \text{signExtend}\{\text{inst}[31:25], \text{inst}[11:7]\} \)
  
  - Store word:
    \[
    \text{sw } \text{imm}(\text{rs1}), \text{rs2} : \text{Mem}[\text{R}[\text{rs1}] + \text{imm}] \leftarrow \text{R}[\text{rs2}]
    \]

  Memory address
Load and Store Instructions

• Store (S-type):

- imm = signExtend({inst[31:25], inst[11:7]})

- Store word:
  \[ sw \text{ imm}(rs1), \text{ rs2} : \text{Mem[R}[rs1] + \text{imm}] \leftarrow \text{R}[\text{rs2}] \]

  Memory address

- RISC-V has a few other store instructions (store half-word, store byte) that we will not use in this course
Note on encodings

- RISC-V encodings are carefully chosen so that rd, rs1, and rs2 always fall on the same bits.

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>7</td>
<td>funct7, rs2, rs1, funct3, rd, opcode</td>
</tr>
<tr>
<td>I</td>
<td>12</td>
<td>imm[11:0], rs1, funct3, rd, opcode</td>
</tr>
<tr>
<td>U</td>
<td>20</td>
<td>imm[31:12], rd, opcode</td>
</tr>
<tr>
<td>S</td>
<td>7</td>
<td>imm[11:5], rs2, rs1, funct3, imm[4:0], opcode</td>
</tr>
</tbody>
</table>
Note on encodings

- RISC-V encodings are carefully chosen so that rd, rs1, and rs2 always fall on the same bits

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>imm[31:12]</td>
<td></td>
<td></td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

March 8, 2018
### Note on encodings

- RISC-V encodings are carefully chosen so that `rd`, `rs1`, and `rs2` always fall on the same bits.

<table>
<thead>
<tr>
<th>R</th>
<th>7</th>
<th>5</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
</tr>
<tr>
<td>I</td>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>20</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>imm[31:12]</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>
**Note on encodings**

- RISC-V encodings are carefully chosen so that rd, rs1, and rs2 always fall on the same bits

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>5</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R</strong></td>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
</tr>
<tr>
<td><strong>I</strong></td>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td><strong>U</strong></td>
<td>imm[31:12]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td><strong>S</strong></td>
<td>imm[11:5]</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>imm[4:0]</td>
<td>opcode</td>
</tr>
</tbody>
</table>
Note on encodings

- RISC-V encodings are carefully chosen so that rd, rs1, and rs2 always fall on the same bits.

```
+--------+-------+-------+-------+-------+-------+
<table>
<thead>
<tr>
<th>R</th>
<th>I</th>
<th>U</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>opc</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>imm[31:12]</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>imm[11:5]</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>imm[4:0]</td>
<td>opcode</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- This results in strange encodings for the immediate field, but simplifies the implementation.
## Control Instructions: Branches

### Conditional branches (SB-type)

<table>
<thead>
<tr>
<th>1</th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>7</th>
</tr>
</thead>
</table>
Control Instructions: Branches

Conditional branches (SB-type)

<table>
<thead>
<tr>
<th>1</th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>7</th>
</tr>
</thead>
</table>

- Comparison operator \(comp\) specified by funct3
Control Instructions: Branches

### Conditional branches (SB-type)

|---------|-----------|-----|-----|--------|----------|---------|--------|

- Comparison operator $comp$ specified by $funct3$

<table>
<thead>
<tr>
<th>Instruction</th>
<th>BEQ</th>
<th>BNE</th>
<th>BLT</th>
<th>BGE</th>
<th>BLTU</th>
<th>BGEU</th>
</tr>
</thead>
<tbody>
<tr>
<td>$comp$</td>
<td>$==$</td>
<td>$!=$</td>
<td>$&lt;$</td>
<td>$\geq$</td>
<td>$&lt;$</td>
<td>$\geq$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>signed</th>
<th>unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;$</td>
<td>$\geq$</td>
</tr>
</tbody>
</table>
Control Instructions: Branches

Conditional branches (SB-type 4KB range)

- Comparison operator \textit{comp} specified by funct3

<table>
<thead>
<tr>
<th>Instruction</th>
<th>BEQ</th>
<th>BNE</th>
<th>BLT</th>
<th>BGE</th>
<th>BLTU</th>
<th>BGEU</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{comp}</td>
<td>\texttt{==}</td>
<td>\texttt{!=}</td>
<td>&lt;</td>
<td>\geq</td>
<td>&lt;</td>
<td>\geq</td>
</tr>
</tbody>
</table>

- Can change PC only within a \(\pm4\)KB range
Control Instructions: Jumps

JAL: Unconditional jump and link (UJ-type)

- imm = signExtend({inst[31], inst[19:12], inst[20], inst[30:21], 1'b0})
- R[rd] ← pc + 4; pc ← pc + imm
- 1MB range of current PC

JALR: Unconditional jump via register and link (I-type)

- imm = signExtend(inst[31:20])
- R[rd] ← pc + 4; pc ← (R[rs1] + imm) & ~0x01
Control Instructions: Jumps

1MB range of current PC

**JAL**: Unconditional jump and link (UJ-type)

- $\text{imm} = \text{signExtend} \{ \text{inst}[31], \text{inst}[19:12], \text{inst}[20], \text{inst}[30:21], 1'b0 \}$
- $R[rd] \leftarrow \text{pc} + 4$; $\text{pc} \leftarrow \text{pc} + \text{imm}$
- Can jump within a $\pm 1$MB range of current PC

**JALR**: Unconditional jump via register and link (I-type)

- $\text{imm} = \text{signExtend}(\text{inst}[31:20])$
- $R[rd] \leftarrow \text{pc} + 4$; $\text{pc} \leftarrow (R[rs1] + \text{imm}) \& \sim 0x01$
Control Instructions: Jumps

1MB range of current PC

### JAL: Unconditional jump and link (UJ-type)

- \[
\text{imm} = \text{signExtend}\{\text{inst}[31], \text{inst}[19:12], \text{inst}[20], \text{inst}[30:21], 1'\text{b0}\}\]
- \[
\text{R}[\text{rd}] \leftarrow \text{pc} + 4; \text{pc} \leftarrow \text{pc} + \text{imm}
\]
- Can jump within a $\pm$1MB range of current PC

### JALR: Unconditional jump via register and link (I-type)

- \[
\text{imm} = \text{signExtend}(\text{inst}[31:20])
\]
- \[
\text{R}[\text{rd}] \leftarrow \text{pc} + 4; \text{pc} \leftarrow (\text{R}[\text{rs1}] + \text{imm}) \& \sim0x01
\]
- \[
\text{R}[\text{rd}] \leftarrow \text{pc} + 4; \text{pc} \leftarrow (\text{R}[\text{rs1}] + \text{imm}) \& \sim0x01
\]
Assembly Code vs. Binary

• It's too tedious to write programs directly in binary

• To simplify writing programs, assemblers provide:
  – Mnemonics for instructions
    • `add x1, x2, x3`
Assembly Code vs. Binary

- It's too tedious to write programs directly in binary.
- To simplify writing programs, assemblers provide:
  - Mnemonics for instructions
    - `add x1, x2, x3`
  - Pseudoinstructions
    - `mv x1, x2` // short for `add x1, x2, x0`
    - `li x1, 6175` // short for `lui x1, 2 ; addi x1, x1, -2017`
      (exact sequence depends on immediate value)
Assembly Code vs. Binary

- It's too tedious to write programs directly in binary.

To simplify writing programs, assemblers provide:

- Mnemonics for instructions
  - `add x1, x2, x3`
- Pseudoinstructions
  - `mv x1, x2` // short for `add x1, x2, x0`
  - `li x1, 6175` // short for `lui x1, 2 ; addi x1, x1, -2017`
    (exact sequence depends on immediate value)
- Symbols for program locations and data
  - `bneq x1, x2, loop_begin`
  - `lw x1, flag`
Assembly Code vs. Binary

- It's too tedious to write programs directly in binary.
- To simplify writing programs, assemblers provide:
  - Mnemonics for instructions
    - `add x1, x2, x3`
  - Pseudoinstructions
    - `mv x1, x2` // short for `add x1, x2, x0`
    - `li x1, 6175` // short for `lui x1, 2 ; addi x1, x1, -2017`
      (exact sequence depends on immediate value)
  - Symbols for program locations and data
    - `bneq x1, x2, loop_begin`
    - `lw x1, flag`

- Assemblers translate programs into machine code for the processor to execute.
// assume x >= 0 && y > 0
int gcd(int a, int b) {
    while (a != 0) {
        if (a >= b) {
            a = a - b;
        } else {
            // Swap a and b
            int t = a;
            a = b;
            b = t;
        }
    }
    return b;
}
GCD in C

// assume x >= 0 && y > 0
int gcd(int a, int b) {
    while (a != 0) {
        if (a >= b) {
            a = a - b;
        } else {
            // Swap a and b
            int t = a;
            a = b;
            b = t;
        }
    }
    return b;
}
Example: GCD

GCD in C

```c
// assume x >= 0 && y > 0
int gcd(int a, int b) {
    while (a != 0) {
        if (a >= b) {
            a = a - b;
        } else {
            // Swap a and b
            int t = a;
            a = b;
            b = t;
        }
    }
    return b;
}
```

GCD in RISC-V Assembler

```assembly
// a: x1, b: x2, t: x3
begin:
    beqz x1, done // if(x1==0) goto done

b_bigger:

done: // now x2 contains the gcd
```
Example: GCD

GCD in C

```c
// assume x >= 0 && y > 0
int gcd(int a, int b) {
    while (a != 0) {
        if (a >= b) {
            a = a - b;
        } else {
            // Swap a and b
            int t = a;
            a = b;
            b = t;
        }
    }
    return b;
}
```

GCD in RISC-V Assembler

```riscv
// a: x1, b: x2, t: x3
begin:
    beqz x1, done // if(x1==0) goto done
    blt x1, x2, b_bigger // if(x1 < x2) // goto b_bigger
b_bigger:
    b_bigger: 

done: // now x2 contains the gcd
```
Example: GCD

GCD in C

// assume x >= 0 && y > 0
int gcd(int a, int b) {
    while (a != 0) {
        if (a >= b) {
            a = a - b;
        } else {
            // Swap a and b
            int t = a;
            a = b;
            b = t;
        }
    }
    return b;
}

GCD in RISC-V Assembler

// a: x1, b: x2, t: x3
begin:
    beqz x1, done // if(x1==0) goto done
    blt x1, x2, b_bigger // if(x1 < x2)
        // goto b_bigger
    sub x1, x1, x2 // x1 ^= x1 - x2
b_bigger:

done: // now x2 contains the gcd
Example: GCD

GCD in C

```c
// assume x >= 0 && y > 0
int gcd(int a, int b) {
    while (a != 0) {
        if (a >= b) {
            a = a - b;
        } else {
            // Swap a and b
            int t = a;
            a = b;
            b = t;
        }
    }
    return b;
}
```

GCD in RISC-V Assembler

```assembly
// a: x1, b: x2, t: x3
begin:
    beqz x1, done // if(x1==0) goto done
    blt x1, x2, b_bigger // if(x1 < x2)
        // goto b_bigger
    sub x1, x1, x2 // x1 = x1 - x2
    j begin // goto begin
b_bigger:

done: // now x2 contains the gcd
```
Example: GCD

GCD in C

```c
// assume x >= 0 && y > 0
int gcd(int a, int b) {
    while (a != 0) {
        if (a >= b) {
            a = a - b;
        } else {
            // Swap a and b
            int t = a;
            a = b;
            b = t;
        }
    }
    return b;
}
```

GCD in RISC-V Assembler

```asm
// a: x1, b: x2, t: x3
begin:
    beqz x1, done // if(x1==0) goto done
    blt x1, x2, b_bigger // if(x1 < x2)
        // goto b_bigger
    sub x1, x1, x2 // x1 ← x1 - x2
    j begin // goto begin
b_bigger:
    mv x3, x1 // x3 ← x1
    mv x1, x2 // x1 ← x2
    mv x2, x3 // x2 ← x3

done: // now x2 contains the gcd
```
Example: GCD

GCD in C

```c
// assume x >= 0 && y > 0
int gcd(int a, int b) {
    while (a != 0) {
        if (a >= b) {
            a = a - b;
        } else {
            // Swap a and b
            int t = a;
            a = b;
            b = t;
        }
    }
    return b;
}
```

GCD in RISC-V Assembler

```assembly
// a: x1, b: x2, t: x3
begin:
    beqz x1, done // if(x1==0) goto done
    blt x1, x2, b_bigger // if(x1 < x2)
        // goto b_bigger
    sub x1, x1, x2 // x1 -= x1 - x2
    j begin // goto begin
b_bigger:
    mv x3, x1 // x3 = x1
    mv x1, x2 // x1 = x2
    mv x2, x3 // x2 = x3
    j begin // goto begin
done:  // now x2 contains the gcd
```
Thank you!

Next lecture: Procedures and stacks