Programmable Machines

Silvina Hanono Wachman
Computer Science & Artificial Intelligence Lab
M.I.T.

Quiz 1: next week
Covers L1-L8
Oct 11, 7:30-9:30PM
Walker memorial 50-340
Using Combinational and Sequential Logic, we can build special-purpose hardware: a finite state machine that solves a particular problem (e.g., Modulo 4 Counter, GCD)
Using Combinational and Sequential Logic, we can build **special-purpose hardware**: a finite state machine that solves a particular problem (e.g., Modulo 4 Counter, GCD).
Using Combinational and Sequential Logic, we can build special-purpose hardware: a finite state machine that solves a particular problem (e.g., Modulo 4 Counter, GCD)

Soon we will be able to build 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 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

- Registers and a Register File can only be used to store a small number of data elements. To support all of our storage needs, we use main memory.
Main Memory = Random-Access Memory

- Registers and a Register File can only be used to store a small number of data elements. To support all of our storage needs, we use main memory.

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Example Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>11101000 10111010 01011010 10010101</td>
</tr>
<tr>
<td>001</td>
<td>10111010 00000000 11110101 00000000</td>
</tr>
<tr>
<td>010</td>
<td>00000000 00000000 11110101 11011000</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>...</td>
</tr>
</tbody>
</table>
 Registers and a Register File can only be used to store a small number of data elements. To support all of our storage needs, we use main memory.

Array of bits, organized in $W$ words of $N$ bits each

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

Can read from and write to individual words

<table>
<thead>
<tr>
<th>Address</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11101000 10110101 01010101 10010101</td>
<td>10110101 00000000 11110101 00000000</td>
<td>01111010 00000000 11110101 00000000</td>
<td>...</td>
<td>00000000 00000000 11110101 11011000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Main Memory = Random-Access Memory

- Registers and a Register File can only be used to store a small number of data elements. To support all of our storage needs, we use main memory.
- Array of bits, organized in \( W \) words of \( N \) bits each
  - Typically, \( W \) is a power of two: \( W = 2^k \)
  - Example: \( W=8 \) (\( k=3 \) address 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>10111010 00000000 11110101 00000000</td>
</tr>
<tr>
<td>001</td>
<td>01111010 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>

October 4, 2018
Storage Conventions: Registers vs Memory

- Typically variables live in memory
- Registers hold temporary values or values that we need to use repeatedly
- ALU operations are performed on registers

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>n</td>
</tr>
<tr>
<td>0x1004</td>
<td>r</td>
</tr>
<tr>
<td>0x1008</td>
<td>x</td>
</tr>
<tr>
<td>0x100C</td>
<td>y</td>
</tr>
<tr>
<td>0x1010</td>
<td></td>
</tr>
</tbody>
</table>
Storage Conventions: Registers vs Memory

- Typically variables live in memory
- Registers hold temporary values or values that we need to use repeatedly
- ALU operations are performed on registers
- To operate with memory variables
  - Load them into a register
  - Compute on them
  - Store the results back to memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>n</td>
</tr>
<tr>
<td>0x1004</td>
<td>r</td>
</tr>
<tr>
<td>0x1008</td>
<td>x</td>
</tr>
<tr>
<td>0x100C</td>
<td>y</td>
</tr>
<tr>
<td>0x1010</td>
<td></td>
</tr>
</tbody>
</table>
Storage Conventions: Registers vs Memory

- Typically variables live in memory
- Registers hold temporary values or values that we need to use repeatedly
- ALU operations are performed on registers
- To operate with memory variables
  - Load them into a register
  - Compute on them
  - Store the results back to memory

```
int x, y;
y = x - 37;
R1 ← Mem[0x1008]
R1 ← R1 - 37
Mem[0x100C] ← R1
```
Von Neumann Computer: 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
Von Neumann Computer: 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
Von Neumann Computer: 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

\[
\text{rd} \leftarrow \text{op}(\text{rs}_1, \text{rs}_2)
\]
Von Neumann Computer: 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)
```

0xba5eba11

Central Processing Unit

Main Memory

- instruction
- instruction
- instruction
- data
- data
- data
Von Neumann Computer: 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

<table>
<thead>
<tr>
<th>rs2</th>
<th>rs1</th>
<th>rd</th>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
rd ← op(rs1,rs2)
```

How does CPU distinguish between instructions and data?

Central Processing Unit

<table>
<thead>
<tr>
<th>Main Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction</td>
</tr>
<tr>
<td>instruction</td>
</tr>
<tr>
<td>instruction</td>
</tr>
<tr>
<td>data</td>
</tr>
<tr>
<td>data</td>
</tr>
<tr>
<td>data</td>
</tr>
</tbody>
</table>

October 4, 2018
Anatomy of a von Neumann Computer
Anatomy of a von Neumann Computer

Datapath

Control Unit

Main Memory

Internal storage

address

data

control

status

address

instructions
Anatomy of a von Neumann Computer

[Diagram showing the components of a computer system, including Datapath, Control Unit, Main Memory, and internal storage, with arrows indicating connections and data flow.]
Anatomy of a von Neumann Computer
Anatomy of a von Neumann Computer

Control Unit

Datapath

Main Memory

Internal storage

address

data

address

instructions

control

status

dest

registers

asell

bsel

fn

ALU

Cc’s

operations
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 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:
    - Fetch instruction
    - Decode instruction
    - Read src operands
    - Execute
    - Write dst operand
    - Compute next PC
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
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
  - 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
  - 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

- Down side: Commercially successful ISAs last for decades
  - Today’s x86 CPUs carry baggage of design decisions from 70’s
RISC-V ISA

In this course we will use the RISC-V ISA

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

In this course we will use the RISC-V ISA

- 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 ISA

In this course we will use the RISC-V ISA

- 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 integer 32-bit variant
# RISC-V ISA: Storage

## CPU State

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>x0-x31</td>
<td>General-Purpose Registers</td>
</tr>
</tbody>
</table>

### General-Purpose Registers

- x0: 000000...0
- x1
- x2
- \[\vdots\]
- x31

**32-bit “words”**
RISC-V ISA: Storage

CPU State

General-Purpose Registers

x0 hardwired to 0

x0 hardwired to 0
RISC-V ISA: Storage

**CPU State**
- PC
- General-Purpose Registers:
  - x0
    - 000000...0
  - x1
  - x2
  - ...
  - x31

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

- x0 hardwired to 0
RISC-V ISA: Storage

**CPU State**

- **PC**: Program Counter
- **General-Purpose Registers**
  - \(x_0\) hardwired to 0
  - \(x_1\)
  - \(x_2\)
  - \(\ldots\)
  - \(x_{31}\)

**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.

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

32-bit “words” (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
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)

  25  
  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)

Why fixed instead of variable-length instructions?

*Simpler to decode & to compute next PC (but larger code)*
Computational Instructions
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 $(\text{funct7, funct3})$
Computational Instructions (R-type)

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

```
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 0 1 1 0 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 \( R[rd] \leftarrow R[rs1] \text{ op } R[rs2] \)
- Operation \( \text{op} \) specified by \( (\text{funct7}, \text{funct3}) \)
- Example: ADD instruction

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

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

```
0 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 0 0 1 1
```

$\text{ opcode } = \text{ OP}$

$\text{ op } = \text{ add}$
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>
<tbody>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</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

  opcode = OP
  rd = 1

  Example instruction: ADD
  
  $\text{opcode} = 11100000$
  $\text{funct7} = 000001$
  $\text{funct3} = 000$
  $\text{rs1} = 000011$
  $\text{rs2} = 000100$
  $\text{rd} = 110000$

  $\text{funct7}$ determines the operation mode:
  
  * 000000: load
  * 000001: store
  * 000010: ALU operation
  * 000011: conditional branch
  * 000100: jump
  * 000101: load immediate
  * 001000: illegal
  * 111111: illegal

  $\text{funct3}$ indicates the operation:
  
  * 000: add
  * 001: subtract
  * 010: and
  * 011: or
  * 100: xor
  * 101: shift left
  * 110: shift right
  * 111: illegal
# 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

<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>

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

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

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

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

\[
\text{opcode} = 000001011000000001011000111
\]

\[
\begin{align*}
\text{rs2} &= 4 \\
\text{rs1} &= 3 \\
\text{rd} &= 1
\end{align*}
\]
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

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

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

```
opcode = OP
op = add
rs2=4
rs1=3
rd=1
```


- We prefer a symbolic representation: $\text{add } x1, \ x3, \ x4$
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 $\mathit{op}$ specified by $(\mathit{funct7}, \mathit{funct3})$
- Example: ADD instruction

- We prefer a symbolic representation: 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, SRL, SRA</td>
</tr>
</tbody>
</table>
Computational Instructions (I-type)

- Many programs use small constants frequently (e.g., comparisons, loop indices, etc.)
- Using registers to hold these constants is wasteful!
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 $\text{R}[\text{rd}] \leftarrow \text{R}[\text{rs1}] \text{ op } \text{imm}$
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 $R[rd] \leftarrow R[rs1] \text{ op imm}$
- *Immediate* operand is encoded into instruction (12 bits (instr_imm))
- To use it: $\text{imm} = \text{signExtend(instr_imm[11:0])}$
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 \( R[rd] \leftarrow R[rs1] \text{ op } \text{imm} \)
- *Immediate* operand is encoded into instruction (12 bits (instr_imm))
- To use it: \( \text{imm} = \text{signExtend}(\text{instr_imm}[11:0]) \)
- Example: \( \text{addi } x5, x2, -3 \)
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, SRLI, 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, SRLI, 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, XORI</td>
<td>SLLI, SRLI, SRAI</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, SRLI, SRAI</td>
</tr>
</tbody>
</table>

Why is there no SUBI?

Equivalent to ADDI with a negative immediate

- Shift instructions use lower 5-bits of immediate
  - $R[rd] \leftarrow R[rs1] \text{ op } \text{instr_imm}[4:0]$

Why is the imm 5 bits for shift instructions?
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, SRLI, SRAI</td>
</tr>
</tbody>
</table>

Why is there no SUBI?

*Equivalent to ADDI with a negative immediate*

- Shift instructions use lower 5-bits of immediate
  - \( R[rd] \leftarrow R[rs1] \ op \ instr\_imm[4:0] \)

Why is the imm 5 bits for shift instructions?

*Data is 32 bits wide,*

*5 bits can specify shift between 0-31*
Example 1

- Execute \( a = ((b+3) \gg c) - 1; \)
Example 1

- Execute $a = ((b+3) \gg c) - 1$;
  1. Break up complex expression into basic computations.
Example 1

- Execute $a = ((b+3) \gg c) - 1$
  
  1. Break up complex expression into basic computations.

```plaintext
    t0 = b + 3;
    t1 = t0 \gg c;
    a = t1 - 1;
```
Example 1

- Execute \( a = ((b+3) >> c) - 1; \)
  1. Break up complex expression into **basic computations**.
  2. Assume \( a, b, c, t_0, \) and \( t_1 \) are in registers. \( x_1: a, x_2: b, x_3: c, x_4: t_0, x_5: t_1 \)

```
  t0 = b + 3;
t1 = t0 >> c;
a = t1 - 1;
```
Example 1

- **Execute** $a = ((b+3) \gg c) - 1$;
  1. Break up complex expression into **basic computations**.

2. Assume $a$, $b$, $c$, $t0$, and $t1$ are in registers. $x1: a$, $x2: b$, $x3: c$, $x4: t0$, $x5: t1$

\[
\begin{align*}
  t0 &= b + 3; \\
  t1 &= t0 \gg c; \\
  a &= t1 - 1;
\end{align*}
\]

\[
\begin{align*}
  \text{addi} &\ x4, x2, 3 \\
  \text{srl} &\ x5, x4, x3 \\
  \text{addi} &\ x1, x5, -1;
\end{align*}
\]
Handling Large Constants

- Sometimes we need to use full 32-bit constants
Handling Large Constants

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

\[
\begin{array}{c|c|c}
20 & 5 & 7 \\
\hline
20 \text{ bit immediate (long_imm)} & \text{rd} & \text{opcode} \\
\end{array}
\]

- \( \text{imm} = \{\text{long_imm}[19:0], 12'\text{b}0\} \)

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

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

<table>
<thead>
<tr>
<th>20 bit immediate (long_imm)</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
</table>

- imm = {long_imm[19:0], 12'b0}
- Load upper immediate (LUI): `lui rd, long_imm`
- R[rd] ← imm

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

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

\[ \text{imm} = \{\text{long_imm}[19:0], 12\text{'b}0\} \]

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

- Example: Write code to load constant 0xCAFE0123 into x3
  
  \[ \text{lui x3, 0xCAFE0} \quad // \quad x3 = 0xCAFE0000 \]
Handling Large Constants

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

```
imm = {long_imm[19:0], 12'b0}
```

Load upper immediate (LUI): `lui rd, long_imm`
R[rd] ← imm

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

```
lui x3, 0xCAFE0       // x3 = 0xCAFE0000
addi x3, x3, 0x123    // x3 = 0xCAFE0123
```
Load and Store Instructions
Load Instruction

- Loads move data from main memory into a register
Load Instruction

- Loads move data from main memory into a register
- Load word: `lw rd, inst_imm(rs1)`
- Example: `lw x2, 4(x3)`
Load Instruction

- Loads move data from main memory into a register
- Load word: `lw rd, inst_imm(rs1)`
- Example: `lw x2, 4(x3)`
- Performs `R[rd] ← Mem[R[rs1] + imm]`

Memory address
Load Instruction

- Loads move data from main memory into a register
- Load word: `lw rd, inst_imm(rs1)`
- Example: `lw x2, 4(x3)`
- Performs `R[rd] ← Mem[R[rs1] + imm]`
  - Memory address

- Encode 12 bit immediate offset in instruction
  - `imm = signExtend(inst_imm[11:0])`
Load Instruction

- Loads move data from main memory into a register
- Load word: `lw rd, inst_imm(rs1)`
- Example: `lw x2, 4(x3)`
- Performs \( R[rd] \leftarrow \text{Mem}[R[rs1] + \text{imm}] \)
  
  Memory address

- Encode 12 bit immediate offset in instruction
  - \( \text{imm} = \text{signExtend}(\text{inst_imm}[11:0]) \)

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

- Stores move data from registers into main memory
- Store word: `sw rs2, inst_imm(rs1)`
- Example: `sw x2, 4(x3)`
Store Instruction

- Stores move data from registers into main memory
- Store word: `sw rs2, inst_imm(rs1)`
- Example: `sw x2, 4(x3)`
- Performs $\text{Mem[R[rs1] + imm]} \leftarrow R[rs2]$
Store Instruction

- Stores move data from registers into main memory
- Store word: `sw rs2, inst_imm(rs1)`
- Example: `sw x2, 4(x3)`
- Performs `Mem[R[rs1] + imm] \leftarrow R[rs2]`

- Encode 12 bit immediate offset in instruction
  - `imm = \text{signExtend}(inst\_imm[11:0])`
Store Instruction

- Stores move data from registers into main memory
- Store word: `sw rs2, inst_imm(rs1)`
- Example: `sw x2, 4(x3)`
- Performs `Mem[R[rs1] + imm] \leftarrow R[rs2]`
  
  **Memory address**

- Encode 12 bit immediate offset in instruction
  - `imm = \text{signExtend}(\text{inst}_\text{imm}[11:0])`

- RISC-V has a few other store instructions (store half-word, store byte) that we will not use in this course
### Example 2

```c
int x, y;
y = x - 37;
```

1. **Memory Locations**: 
   - **0x1000**: `n`
   - **0x1004**: `r`
   - **0x1008**: `x`
   - **0x100C**: `y`

2. **Assembly Code**
   - `R1 ← Mem[0x1008]`
   - `R1 ← R1 - 37`
   - `Mem[0x100C] ← R1`
Example 2

```c
int x, y;
y = x - 37;
```

```
R1 ← Mem[0x1008]
R1 ← R1 - 37
Mem[0x100C] ← R1
```

- Load x into x2 register
  - Get value 0x1008 into x1
  - Load Mem[x1] into x2
Example 2

```
int x, y;
y = x - 37;
R1 ← Mem[0x1008]
R1 ← R1 - 37
Mem[0x100C] ← R1
```

- Load x into x2 register
  - Get value 0x1008 into x1
  - Load Mem[x1] into x2

Table:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>n</td>
</tr>
<tr>
<td>0x1004</td>
<td>r</td>
</tr>
<tr>
<td>0x1008</td>
<td>x</td>
</tr>
<tr>
<td>0x100C</td>
<td>y</td>
</tr>
<tr>
<td>0x1010</td>
<td></td>
</tr>
</tbody>
</table>

```c
lui x1, 1       // x1 = 0x1000
addi x1, x1, 8  // x1 = 0x1008
```
Example 2

```
int x, y;
y = x - 37;
```

```
R1 ← Mem[0x1008]
R1 ← R1 - 37
Mem[0x100C] ← R1
```

- Load x into x2 register
  - Get value 0x1008 into x1
  - Load Mem[x1] into x2

```
lui x1, 1
addi x1, x1, 8
lw x2, 0(x1)
```

// x1 = 0x1000
// x1 = 0x1008
// x2 = Mem[0x1008]
Example 2

```
int x, y;
y = x - 37;
```

- Load x into x2 register
  - Get value 0x1008 into x1
  - Load Mem[x1] into x2
  - Subtract 37 from x2

```
R1 ← Mem[0x1008]
R1 ← R1 - 37
Mem[0x100C] ← R1
lui x1, 1       // x1 = 0x1000
addi x1, x1, 8  // x1 = 0x1008
lw x2, 0(x1)    // x2 = Mem[0x1008]`
### Example 2

```plaintext
int x, y;
y = x - 37;

R1 ← Mem[0x1008]
R1 ← R1 - 37
Mem[0x100C] ← R1

lui x1, 1 // x1 = 0x1000
addi x1, x1, 8 // x1 = 0x1008
lw x2, 0(x1) // x2 = Mem[0x1008]
addi x2, x2, -37 // x2 = x2 - 37
```

- Load x into x2 register
  - Get value 0x1008 into x1
  - Load Mem[x1] into x2
- Subtract 37 from x2
Example 2

```
int x, y;
y = x - 37;
```

R1 ← Mem[0x1008]
R1 ← R1 - 37
Mem[0x100C] ← R1

```
lui x1, 1 // x1 = 0x1000
addi x1, x1, 8 // x1 = 0x1008
lw x2, 0(x1) // x2 = Mem[0x1008]
addi x2, x2, -37 // x2 = x2 - 37
```

- Load x into x2 register
  - Get value 0x1008 into x1
  - Load Mem[x1] into x2
  - Subtract 37 from x2
  - Store x2 into y
Example 2

```
int x, y;
y = x - 37;
```

- Load x into x2 register
  - Get value 0x1008 into x1
  - Load Mem[x1] into x2
  - Subtract 37 from x2
  - Store x2 into y

```
R1 ← Mem[0x1008]
R1 ← R1 - 37
Mem[0x100C] ← R1
```

```
lui x1, 1  // x1 = 0x1000
addi x1, x1, 8  // x1 = 0x1008
lw x2, 0(x1)  // x2 = Mem[0x1008]
addi x2, x2, -37  // x2 = x2 - 37
sw x2, 4(x1)  // Mem[0x100C] = x2
                // 0x100C = 0x1008 + 4
```
Control Instructions
Control Instructions: Branches

Conditional branches: Example: `blt x1, x2, label`

- 4KB range
- Supported comparison operators
Control Instructions: Branches

Conditional branches: Example: `blt x1, x2, label`
- First performs comparison to determine if branch is taken or not: `R[rs1] comp R[rs2]`

- 4KB range
- Supported comparison operators
Conditional branches: Example: `blt x1, x2, label`

- First performs comparison to determine if branch is taken or not: `R[rs1] comp R[rs2]`
- If comparison returns True, then branch is taken:
  - Performs `pc ← pc + imm`
- Else:
  - Performs `pc ← pc + 4`

- 4KB range
- Supported comparison operators
Control Instructions: Branches

4KB range

Conditional branches: Example: `blt x1, x2, label`

- First performs comparison to determine if branch is taken or not: `R[rs1] comp R[rs2]`
- If comparison returns True, then branch is taken:
  - Performs `pc ← pc + imm`
- Else:
  - Performs `pc ← pc + 4`
- Offset to label is encoded in 12 bit immediate in instruction
  - `imm = signExtend(inst_imm[11:0], 1'b0)`
  - Enforces half word alignment
  - Can change PC only within a ±4KB range
- Supported comparison operators
**Control Instructions: Branches**

4KB range

**Conditional branches: Example:** \texttt{blt x1, x2, label}

- First performs comparison to determine if branch is taken or not: \texttt{R[rs1] comp R[rs2]}
- If comparison returns True, then branch is taken:
  - Performs \texttt{pc \leftarrow pc + imm}
- Else:
  - Performs \texttt{pc \leftarrow pc + 4}
- Offset to label is encoded in 12 bit immediate in instruction
  - \texttt{imm = signExtend(inst_imm[11:0],1'b0)}
    - Enforces half word alignment

- Supported comparison operators
- Supported comparison operators

<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>\texttt{&lt;}</td>
<td>\texttt{\ge}</td>
<td>\texttt{&lt;}</td>
<td>\texttt{\ge}</td>
</tr>
</tbody>
</table>
Unconditional Control Instructions: Jumps

JAL: Unconditional jump and link
- Example: jal rd, label

- 1MB range of current PC

JALR: Unconditional jump via register and link
- Example: jalr rd, 0(rs1)
- Performs R[rd] ← pc + 4; pc ← (R[rs1] + imm) & ~0x01
  - ~0x01 forces LSB to 0 (half word alignment)
- Encodes 12 bit immediate in instruction
  - imm = signExtend(inst_imm[11:0])
- Can jump to any 32 bit address – supports long jumps
Unconditional Control Instructions: Jumps

**JAL:** Unconditional jump and link
- Example: `jal rd, label`
- Performs $R[rd] \leftarrow pc + 4$; $pc \leftarrow pc + imm$

- 1MB range of current PC

**JALR:** Unconditional jump via register and link
- Example: `jalr rd, 0(rs1)`
- Performs $R[rd] \leftarrow pc + 4$; $pc \leftarrow R[rs1] + imm$ & $\sim0x01$
  - $\sim0x01$ forces LSB to 0 (half word alignment)
- Encodes 12 bit immediate in instruction
  - $imm = \text{signExtend}(\text{inst_imm}[11:0])$
- Can jump to any 32 bit address – supports long jumps
Unconditional Control Instructions: Jumps

1MB range of current PC

JAL: Unconditional jump and link
- Example: jal rd, label
- Performs R[rd] ← pc + 4; pc ← pc + imm
- Encodes 20 bit immediate in instruction
  - imm = signExtend(inst_imm[19:0], 1'b0)
  - Can jump within a ±1MB range of current PC

JALR: Unconditional jump via register and link
- Example: jalr rd, 0(rs1)
- Performs R[rd] ← pc + 4; pc ← (R[rs1] + imm) & ~0x01
  - ~0x01 forces LSB to 0 (half word alignment)
- Encodes 12 bit immediate in instruction
  - imm = signExtend(inst_imm[11:0])
- Can jump to any 32 bit address – supports long jumps
Unconditional Control Instructions: Jumps

1MB range of current PC

JAL: Unconditional jump and link
- Example: `jal rd, label`
- Performs `R[rd] ← pc + 4; pc ← pc + imm`
- Encodes 20 bit immediate in instruction
  - `imm = signExtend(inst_imm[19:0],1’b0)`
  - Can jump within a $\pm 1$MB range of current PC

JALR: Unconditional jump via register and link
- Example: `jalr rd, 0(rs1)`
- Example: `jalr rd, 0(rs1)`
- Performs `R[rd] ← pc + 4; pc ← (R[rs1] + imm) & ~0x01`
  - $\sim 0x01$ forces LSB to 0 (half word alignment)
- Encodes 12 bit immediate in instruction
  - `imm = signExtend(inst_imm[11:0])`
- Can jump to any 32 bit address – supports long jumps
Unconditional Control Instructions: Jumps

1MB range of current PC

**JAL:** Unconditional jump and **link**
- Example: `jal rd, label`
- Performs `R[rd] ← pc + 4; pc ← pc + imm`
- Encodes 20 bit immediate in instruction
  - `imm = signExtend(inst_imm[19:0],1'b0)`
  - Can jump within a ±1MB range of current PC

**JALR:** Unconditional jump via register and **link**
- Example: `jalr rd, 0(rs1)`
- Performs `R[rd] ← pc + 4; pc ← (R[rs1] + imm) & ~0x01`
  - `~0x01` forces LSB to 0 (half word alignment)
  - `~0x01` forces LSB to 0 (half word alignment)
- Encodes 12 bit immediate in instruction
  - `imm = signExtend(inst_imm[11:0])`
- Can jump to **any 32 bit address** — supports long jumps
Unconditional Control Instructions: Jumps

1MB range of current PC

**JAL**: Unconditional jump and link

- Example: `jal rd, label`
- Performs `R[rd] ← pc + 4; pc ← pc + imm`
- Encodes 20 bit immediate in instruction
  - `imm = signExtend(inst_imm[19:0], 1'b0)`
  - Can jump within a ±1MB range of current PC

**JALR**: Unconditional jump via register and link

- Example: `jalr rd, 0(rs1)`
- Performs `R[rd] ← pc + 4; pc ← (R[rs1] + imm) & ~0x01`
  - `~0x01` forces LSB to 0 (half word alignment)
- Encodes 12 bit immediate in instruction
  - `imm = signExtend(inst_imm[11:0])`
- Can jump to any 32 bit address – supports long jumps

Can jump to any 32 bit address – supports long jumps
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`
  - 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`
  - Symbols for program locations and data
    - `bneq x1, x2, loop_begin`
    - `lw x1, flag`
  - Pseudoinstructions
    - `mv x1, x2` // short for `addi x1, x2, 0`
    - `j label` // short for `jal x0, label`
    - `beqz x1, dest` // short for `beq x1, x0, dest`
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`
  - Symbols for program locations and data
    - `bneq x1, x2, loop_begin`
    - `lw x1, flag`
  - Pseudoinstructions
    - `mv x1, x2`  // short for `addi x1, x2, 0`
    - `j label`  // short for `jal x0, label`
    - `beqz x1, dest`  // short for `beq x1, x0, dest`

- Assemblers translate programs into machine code for the processor to execute.
Example 3

GCD Computation:

```c
int a, b;
while (a != 0) {
    if (a >= b) {
        a = a - b;
    } else {
        // Swap a and b
        int t = a;
        a = b;
        b = t;
    }
}
```

Assume x1: a, x2: b, x3: t

```c
while:
    beqz x1, done  // if(a==0)
    // goto done

b_bigger:

done:  // now x2: b
    // holds the gcd
```

GCD Computation:

Assume x1: a, x2: b, x3: t

```c
int a, b;
while (a != 0) {
    if (a >= b) {
        a = a - b;
    } else {
        // Swap a and b
        int t = a;
        a = b;
        b = t;
    }
}
```
Example 3

GCD Computation:

```c
int a, b;
while (a != 0) {
    if (a >= b) {
        a = a - b;
    } else {
        // Swap a and b
        int t = a;
        a = b;
        b = t;
    }
}
```

Assume x1: a, x2: b, x3: t

```assembly
while:
    beqz x1, done       // if(a==0)
    // goto done
    blt x1, x2, b_bigger // if(a < b)
    // goto b_bigger

b_bigger:

done:                   // now x2: b
    // holds the gcd
```

GCD Computation:
Assume x1: a, x2: b, x3: t
Example 3

GCD Computation:

```c
int a, b;
while (a != 0) {
    if (a >= b) {
        a = a - b;
    } else {
        // Swap a and b
        int t = a;
        a = b;
        b = t;
    }
}
```

Assume x1: a, x2: b, x3: t

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

done:  // now x2: b
    // holds the gcd
```

GCD Computation:
Assume x1: a, x2: b, x3: t
Example 3

GCD Computation:

```c
int a, b;
while (a != 0) {
    if (a >= b) {
        a = a - b;
    } else {
        // Swap a and b
        int t = a;
        a = b;
        b = t;
    }
}
```

Assume x1: a, x2: b, x3: t

```asm
while:
    beqz x1, done // if(a==0)
    // goto done
    blt x1, x2, b_bigger // if(a < b)
    // goto b_bigger
    sub x1, x1, x2 // a a - b
    j while // goto while
b_bigger:

done: // now x2: b
    // holds the gcd
```
Example 3

GCD Computation:

```
int a, b;
while (a != 0) {
    if (a >= b) {
        a = a - b;
    } else {
        // Swap a and b
        int t = a;
        a = b;
        b = t;
    }
}
```

Assume $x_1$: $a$, $x_2$: $b$, $x_3$: $t$

```
while:
    beqz x1, done  // if(a==0)
    // goto done
    blt x1, x2, b_bigger  // if(a < b)
    // goto b_bigger
    sub x1, x1, x2  // a = a - b
    j while  // goto while
b_bigger:
    mv x3, x1  // t = a
    mv x1, x2  // a = b
    mv x2, x3  // b = t
done:  // now x2: $b$
    // holds the gcd
```
Example 3

GCD Computation:

```c
int a, b;
while (a != 0) {
    if (a >= b) {
        a = a - b;
    } else {
        // Swap a and b
        int t = a;
        a = b;
        b = t;
    }
}
```

Assume x1: a, x2: b, x3: t

```assembly
while:
    beqz x1, done  // if(a==0)
        // goto done
    blt x1, x2, b_bigger  // if(a < b)
        // goto b_bigger
    sub x1, x1, x2  // a a - b
    j while  // goto while
b_bigger:
    mv x3, x1  // t a
    mv x1, x2  // a b
    mv x2, x3  // b t
    j while  // goto while
done:
    // now x2: b
    // holds the gcd
```
Take-home problem: Write RISC-V assembly code

You are told that an array $A$ of 10 32-bit values are stored in memory beginning at address 0x100.

1. $A[0]$ is stored at address 0x100. At what address is $A[3]$ stored?

Thank you!

Next lecture: Procedures and stacks