RISC-V Processor Datapath
Great Idea #1: Abstraction
(Levels of Representation/Interpretation)

lw  t0, t2, 0
lw  t1, t2, 4
sw  t1, t2, 0
sw  t0, t2, 4

High Level Language Program (e.g., C)

Assembly Language Program (e.g., RISC-V)

Machine Language Program (RISC-V)

Compiler

Assembler

Machine Language Program (RISC-V)

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

Hardware Architecture Description (e.g., block diagrams)

Register File

ALU

Anything can be represented as a number, i.e., data or instructions

0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111

We are here!
Recap: Complete RV32I ISA

<table>
<thead>
<tr>
<th>imm[31:12]</th>
<th>rd</th>
<th>0110111</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[31:12]</td>
<td>rd</td>
<td>0010111</td>
</tr>
</tbody>
</table>

**Table:**

| imm[20:10|11|19:12] | rd | 1101111 |
|-------------|----|---------|
| imm[11:0] | rs1 | 000 | 1100111 |
| imm[12:10] | rs2 | rd | 1100111 |
| imm[12:10] | rs2 | rd | 1100111 |
| imm[12:10] | rs2 | rd | 1100111 |
| imm[11:0] | rs1 | 000 | rd | 0000111 |
| imm[11:0] | rs1 | 001 | rd | 0000111 |
| imm[11:0] | rs1 | 010 | rd | 0000111 |
| imm[11:0] | rs1 | 011 | rd | 0000111 |
| imm[11:0] | rs2 | 000 | imm[4:0] | 0100011 |
| imm[11:0] | rs1 | 000 | rd | 0010011 |
| imm[11:0] | rs1 | 010 | rd | 0010011 |
| imm[11:0] | rs1 | 011 | rd | 0010011 |
| imm[11:0] | rs1 | 100 | rd | 0010011 |
| imm[11:0] | rs1 | 110 | rd | 0010011 |
| imm[11:0] | rs1 | 111 | rd | 0010011 |

**Instructions:**
- LUI
- AUIPC
- JAL
- JALR
- BEQ
- BNE
- BLT
- BGE
- BLTU
- BGEU
- LB
- LH
- LW
- LBU
- LHU
- SB
- SH
- SW
- ADDI
- SLTI
- SLTUI
- XORI
- ORI
- ANDI
- SLLI
- SRLI
- SRAI
- ADD
- SUB
- SLL
- SLT
- SLTU
- XOR
- SRL
- SRA
- OR
- AND
- FENCE
- FENCE.I
- ECALL
- EBREAK
- CSRWR
- CSRRS
- CSRRC
- CSRRW
- CSRRS
- CSRRC

*Not in CS61C*
“State” Required by RV32I ISA

Each instruction reads and updates this state during execution:

- **Registers (x0 . . x31)**
  - Register file (or regfile) Reg holds 32 registers x 32 bits/register: Reg[0] . . Reg[31]
  - First register read specified by rs1 field in instruction
  - Second register read specified by rs2 field in instruction
  - Write register (destination) specified by rd field in instruction
  - x0 is always 0 (writes to Reg[0] are ignored)

- **Program Counter (PC)**
  - Holds address of current instruction

- **Memory (MEM)**
  - Holds both instructions & data, in one 32-bit byte-addressed memory space
  - We’ll use separate memories for instructions (IMEM) and data (DMEM)
    - Later we’ll replace these with instruction and data caches
  - Instructions are read (fetched) from instruction memory (assume IMEM read-only)
  - Load/store instructions access data memory
1. Current state outputs drive the inputs to the combinational logic, whose outputs settles at the values of the state before the next clock edge.

2. At the rising clock edge, all the state elements are updated with the combinational logic outputs, and execution moves to the next clock cycle.

3. Separate instruction/data memory:
   For simplification, memory is asynchronous read (not clocked), but synchronous write (is clocked).

One-Instruction-Per-Cycle RISC-V Machine

On every tick of the clock, the computer executes one instruction.
Basic Phases of Instruction Execution

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Register Write
Implementing the **add** instruction

**add** rd, rs1, rs2

- Instruction makes two changes to machine’s state:
  
  Reg[rd] = Reg[rs1] + Reg[rs2]
  
  PC = PC + 4
Datapath for **add**

```
+4
IMEM

inst[11:7]
inst[19:15]
inst[24:20]

Reg[]
DataD
AddrD
AddrA
DataA
AddrB
DataB

Reg[rs1]
Reg[rs2]

alu

inst[31:0]
RegWEn
(RegWriteEnable)
1=write, 0=no write

Control Logic
```
Timing Diagram for \texttt{add}

- \texttt{pc+4} to \texttt{IME}
- \texttt{IMEM} with \texttt{inst[31:0]}
- \texttt{Reg[1]}:
- \texttt{Reg[2]} + \texttt{Reg[3]}
- \texttt{alu}:
- \texttt{Reg[2]} + \texttt{Reg[3]}
Implementing the **sub** instruction

<table>
<thead>
<tr>
<th></th>
<th>rs2</th>
<th>rs1</th>
<th>000</th>
<th>rd</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>000011</td>
</tr>
<tr>
<td>0100000</td>
<td></td>
<td></td>
<td>000</td>
<td>rd</td>
<td>000011</td>
</tr>
</tbody>
</table>

- Almost the same as add, except now have to subtract operands instead of adding them
- **inst[30]** selects between add and subtract
Datapath for add/sub

**Control Logic**

- **Inst[31:0]**
- **RegWEn** (1=write, 0=no write)
- **ALUSel** (Add=0/Sub=1)

**Datapath Components**
- **pc+4**
- **IMEM**
- **Reg[]**
- **AddrA**
- **DataD**
- **AddrB**
- **DataB**
- **Reg[rs1]**
- **Reg[rs2]**
- **ALU**
- **alu**
Implementing other R-Format instructions

<table>
<thead>
<tr>
<th></th>
<th>rs2</th>
<th>rs1</th>
<th>000</th>
<th>rd</th>
<th>0110011</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADD</td>
</tr>
<tr>
<td>010000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>000000</td>
<td>rs2</td>
<td>rs1</td>
<td>001</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>000000</td>
<td>rs2</td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>000000</td>
<td>rs2</td>
<td>rs1</td>
<td>011</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>000000</td>
<td>rs2</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>000000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>010000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>000000</td>
<td>rs2</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>000000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
<td>0110011</td>
</tr>
</tbody>
</table>
Implementing the `addi` instruction

- RISC-V Assembly Instruction:
  
  ```
  addi x15, x1, -50
  ```

<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `imm=-50`        - `rs1=1`    - ADD           - `rd=15`   - OP-Imm
Datapath for add/sub

Control Logic

RegWEn
(1=write, 0=no write)

ALUSel
(Add=0/Sub=1)
Adding **addi** to datapath

![Diagram of adding addi to datapath]

- **pc+4**
- **IMEM**
- **+4**
- **Inst[11:7]**
- **Inst[19:15]**
- **Inst[24:20]**
- **Inst[31:20]**
- **Imm[31:0]**
- **Reg[]**
- **DataD**
- **AddrD**
- **AddrA**
- **AddrB**
- **Reg[rs1]**
- **Reg[rs2]**
- **Imm. Gen**
- **alu**
- **BSel=1**
- **ALUSel=Add**
- **Control Logic**

**Notes:**
- **ALUSel=Add**
- **ImmSel=I**
- **RegWEn=1**
- **Inst[31:0]**
- **Inst[31:20]**
- **Inst[11:7]**
- **Inst[19:15]**
- **Inst[24:20]**
- **Imm[31:0]**
- **ImmSel=I**
- **RegWEn=1**
- **BSel=1**
- **ALUSel=Add**
I-Format immediates

- High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])
- Immediate is sign-extended by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])
Adding **addi** to datapath

Also works for all other I-format arithmetic instruction (**slti**, **sltiu**, **andi**, **ori**, **xori**, **slli**, **srli**, **srai**) just by changing **ALUSel**

Control Logic
Implementing Load Word instruction

- **RISC-V Assembly Instruction:**
  \[ \text{lw } x14, \ 8(x2) \]

<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>

**Binary Representation:**

```
000000001000 | 00010 | 010 | 01110 | 0000011
```

- **imm** = +8
- **rs1** = 2
- **LW**
- **rd** = 14
- **LOAD**
Adding **addi** to datapath

```
pc+4 -> IMEM -> +4

inst[19:15] -> AddrA
inst[24:20] -> AddrB
inst[31:20] -> Imm.

Inst[31:0] -> Imm. Gen

Reg[]
- DataD
- AddrD
- Inst[11:7]
- Inst[19:15]
- Inst[24:20]

Reg[rs1]
- AddrA
- DataA

Reg[rs2]
- AddrB
- DataB

Imm[31:0] -> BSel=1

alu

RegWEn=1

Inst[31:20] -> ImmSel=I

Control Logic

inst[31:0] -> ImmSel=I
RegWEn=1
BSel=1
ALUSel=Add
```
Adding `lw` to datapath
Adding `lw` to datapath
All RV32 Load Instructions

| imm[11:0] | rs1 | 000 | rd | 0000011 | LB   |
| imm[11:0] | rs1 | 001 | rd | 0000011 | LH   |
| imm[11:0] | rs1 | 010 | rd | 0000011 | LW   |
| imm[11:0] | rs1 | 100 | rd | 0000011 | LBU  |
| imm[11:0] | rs1 | 101 | rd | 0000011 | LHU  |

writing back to register file.

funct3 field encodes size and signedness of load data
Announcements!

• Project 2B due Friday!
• Remote Lab Checkoffs...
  • A separate queue that you have to be in by 10AM
    • Timing needed to prevent overloading later labs
    • Submit a Google meeting link, have your lab partner join
  • TA/AI will join the meeting and do the checkoff process remotely
• Prototyping this starting Monday for those who feel a need to self-isolate due to COVID-19...
Implementing Store Word instruction

- RISC-V Assembly Instruction:
  \[ \text{sw } x14, \ 8(\times2) \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>00000000</td>
<td>01110</td>
<td>00010</td>
<td>010</td>
<td>010000</td>
<td>0100011</td>
</tr>
</tbody>
</table>

- Combined 12-bit offset = 8
Adding \textit{lw} to datapath
Adding *sw* to datapath

```
IMEM
+4
IMEM

Reg[rs1]
inst[11:7]
AddrD

Reg[rs2]
inst[19:15]
AddrA

imm[31:0]
inst[24:20]
AddrB

Imm.Gen
inst[31:7]

alu

ALU

DMEM

imm[31:0]

RegWEn=0

Bsel=1

ALUSel=Add

MemRW=Write

WBSel=*  

* = “Don’t Care”
```
Adding $sw$ to datapath

- pc + 4
- IMEM
- Inst[11:7]
- Inst[19:15]
- Inst[24:20]
- Inst[31:20]
- Imm. Gen
- Inst[31:0]
- ImmSel = S
- RegWEn
- BSel = 1
- ALUSel = Add
- MemRW = Write
- WBSel = *

* = “Don’t Care”
I-Format immediates

- High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])
- Immediate is sign-extended by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])
I & S Immediate Generator

- Just need a 5-bit mux to select between two positions where low five bits of immediate can reside in instruction
- Other bits in immediate are wired to fixed positions in instruction
## Implementing Branches

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

- B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate.
- But now immediate represents values -4096 to +4094 in 2-byte increments.
- The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it).
Adding \texttt{sw} to datapath

![Diagram showing the datapath with \texttt{sw} added]

- **PC**: Program Counter
- **IMEM**: Instruction Memory
- **DMEM**: Data Memory
- **Reg[]**: Register File
- **ALU**: Arithmetic Logic Unit
- **Mem**: Memory Unit
- **WBSel**: Write-Back Select
- **Wb**: Write-Back
- **alu**: ALU
- **mem**: Memory
- **inst[31:0]**: Instruction
- **ImmSel**: Immediate Selection
- **RegWEn**: Register Write Enable
- **Reg[rs1]**, **Reg[rs2]**: Register Select
- **Addr[31:0]**, **Data[31:0]**: Address/Data

The diagram illustrates the flow of data and control signals through the datapath, including the addition of instructions and immediate values.
Adding branches to datapath
Adding branches to datapath

PCSel=taken/not-taken  inst[31:0]  ImmSel=BR  RegWEn=0  BrUnBrEqBrLT  Bsel=1ASel=1  MemRW=Read  WBSel=*  ALUSel=Add

CS 61c
Branch Comparator

- \( \text{BrEq} = 1 \), if \( A = B \)
- \( \text{BrLT} = 1 \), if \( A < B \)
- \( \text{BrUn} = 1 \) selects unsigned comparison for \( \text{BrLT} \), 0=signed
- \( \text{BGE branch}: A \geq B \), if \( !(A < B) \)
Implementing **JALR** Instruction (I-Format)

- **JALR rd, rs, immediate**
- Writes PC+4 to Reg[rd] (return address)
- Sets PC = Reg[rs1] + immediate
- Uses same immediates as arithmetic and loads
  - *no* multiplication by 2 bytes
Adding branches to datapath
Adding jalr to datapath
Adding \texttt{jalr} to datapath

![Diagram of adding jalr to datapath]

- **PC** (Program Counter)
- **IMEM** (Instruction Memory)
- **Reg[]** (Register File)
- **AddrD**
- **AddrA**
- **AddrB**
- **DataA**
- **DataB**
- **DataD**
- **Addr**
- **DataW**
- **DataR**
- **alu** (Arithmetic Logic Unit)
- **wb** (Write-Back Buffer)
- **ALU**
- **Branch Comp**
- **DMEM** (Data Memory)
- **wb**

### Control Signals
- \texttt{ALUSel} = Add
- \texttt{MemRW} = Read
- \texttt{ALUSel} = Add

### Execution
- \texttt{inst[31:0]}
- \texttt{imm[31:0]}
- \texttt{Reg[rs1]}
- \texttt{Reg[rs2]}

### Arithmetic Logic Unit (ALU)
- \texttt{ALU} operations

---

CS 61c
Implementing jal Instruction

- JAL saves PC+4 in Reg[rd] (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
  - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost
Adding `jal` to datapath
Adding `jal` to datapath

Diagram showing the datapath components and their connections, including ALU, IMEM, ALU, DMEM, and WB stages. The diagram includes labels for PC, IMEM, IM Gen, Reg[], IMME, and DMEM, among others, with arrows indicating data flow and control signals.
Single-Cycle RISC-V RV32I Datapath
And in Conclusion, …

- Universal datapath
  - Capable of executing all RISC-V instructions in one cycle each
  - Not all units (hardware) used by all instructions
- 5 Phases of execution
  - IF, ID, EX, MEM, WB
  - Not all instructions are active in all phases
- Controller specifies how to execute instructions
  - what new instructions can be added with just most control?