RISC-V Processor Design
Machine Structures

CS61C

Application (ex: browser)
Compiler
Assembler
Processor
Memory
I/O system
Datapath & Control
Digital Design
Circuit Design
Transistors
Fabrication

Software
Hardware
Instruction Set Architecture
New-School Machine Structures

Software

Parallel Requests
Assigned to computer
e.g., Search “Cats”

Parallel Threads
Assigned to core e.g., Lookup, Ads

Parallel Instructions
>1 instruction @ one time
e.g., 5 pipelined instructions

Parallel Data
>1 data item @ one time
e.g., Add of 4 pairs of words

Hardware descriptions
All gates work in parallel at same time

Harness Parallelism & Achieve High Performance

Hardware

Warehouse Scale Computer

Computer

Core

Memory (Cache)

Input/Output

Exec. Unit(s)

Functional Block(s)

Main Memory

Logic Gates

Smart Phone

Harness Parallelism & Achieve High Performance

Hardware

Warehouse Scale Computer

Computer

Core

Memory (Cache)

Input/Output

Exec. Unit(s)

Functional Block(s)

Main Memory

Logic Gates

Harness Parallelism & Achieve High Performance

Hardware

Warehouse Scale Computer

Computer

Core

Memory (Cache)

Input/Output

Exec. Unit(s)

Functional Block(s)

Main Memory

Logic Gates

Harness Parallelism & Achieve High Performance

Hardware

Warehouse Scale Computer

Computer

Core

Memory (Cache)

Input/Output

Exec. Unit(s)

Functional Block(s)

Main Memory

Logic Gates

Harness Parallelism & Achieve High Performance

Hardware

Warehouse Scale Computer

Computer

Core

Memory (Cache)

Input/Output

Exec. Unit(s)

Functional Block(s)

Main Memory

Logic Gates

Harness Parallelism & Achieve High Performance

Hardware

Warehouse Scale Computer

Computer

Core

Memory (Cache)

Input/Output

Exec. Unit(s)

Functional Block(s)

Main Memory

Logic Gates

Harness Parallelism & Achieve High Performance

Hardware

Warehouse Scale Computer

Computer

Core

Memory (Cache)

Input/Output

Exec. Unit(s)

Functional Block(s)

Main Memory

Logic Gates

Harness Parallelism & Achieve High Performance

Hardware

Warehouse Scale Computer

Computer

Core

Memory (Cache)

Input/Output

Exec. Unit(s)

Functional Block(s)

Main Memory

Logic Gates
Great Idea #1: Abstraction (Levels of Representation/Interpretation)

- High Level Language Program (e.g., C)
  - Compiler
- Assembly Language Program (e.g., RISC-V)
  - Assembler
- Machine Language Program (RISC-V)
- Hardware Architecture Description (e.g., block diagrams)
- Architecture Implementation
- Logic Circuit Description (Circuit Schematic Diagrams)

```
// C Code
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
# Assembly Code
lw x3, 0(x10)
lw x4, 4(x10)
sw x4, 0(x10)
sw x3, 4(x10)
```

```
0000 1101 1110 0010 0000 0000 0000 0000
0000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

```
Out = AB + CD
```

```
// Machine Code
```

```
// Logic Circuit Diagrams
```
Our Single-Core Processor So Far…

Processor
- Control

Datapath
- Program Counter (PC)
- Registers
- Arithmetic-Logic Unit (ALU)

Memory
- Program
- Data

Input
- Enable?
- Read/Write
- Address
- Write Data

Output
- Read Data
The CPU

- **Processor (CPU):** the active part of the computer that does all the work (data manipulation and decision-making)

- **Datapath:** portion of the processor that contains hardware necessary to perform operations required by the processor (the brawn)

- **Control:** portion of the processor (also in hardware) that tells the datapath what needs to be done (the brain)
### Base Integer Instructions: RV32I

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Fmt</th>
<th>RV32I Base</th>
<th>Category</th>
<th>Name</th>
<th>Fmt</th>
<th>RV32I Base</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Shifts</strong></td>
<td>Shift Left Logical</td>
<td>R</td>
<td>SLL rd,rs1,rs2</td>
<td>Loads</td>
<td>Load Byte</td>
<td>I</td>
<td>LB rd,rs1,imm</td>
</tr>
<tr>
<td></td>
<td>Shift Left Log. Imm.</td>
<td>I</td>
<td>SLLI rd,rs1,shamt</td>
<td>Load Word</td>
<td>LW rd,rs1,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Right Logical</td>
<td>R</td>
<td>SRL rd,rs1,rs2</td>
<td>Load Byte Signed</td>
<td>LBUS rd,rs1,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Right Log. Imm.</td>
<td>I</td>
<td>SRLI rd,rs1,shamt</td>
<td>Load Word Signed</td>
<td>LHSU rd,rs1,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Right Arithmetic</td>
<td>R</td>
<td>SRA rd,rs1,rs2</td>
<td>Load Word</td>
<td>LW rd,rs1,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Right Arith. Imm.</td>
<td>I</td>
<td>SRAI rd,rs1,shamt</td>
<td>Store Byte</td>
<td>SB rs1,rs2,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic</strong></td>
<td>ADD Immediate</td>
<td>I</td>
<td>ADD rd,rs1,rs2</td>
<td>Store Word</td>
<td>SH rs1,rs2,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUBtract</td>
<td>R</td>
<td>SUB rd,rs1,rs2</td>
<td>Branch =</td>
<td>B  BEQ rs1,rs2,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Load Upper Imm</td>
<td>U</td>
<td>LUI rd,imm</td>
<td>Branch =</td>
<td>B  BNE rs1,rs2,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Add Upper Imm to PC</td>
<td>U</td>
<td>AUIPC rd,imm</td>
<td>Branch &lt;</td>
<td>B  BLT rs1,rs2,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Logical</strong></td>
<td>XOR Immediate</td>
<td>I</td>
<td>XORI rd,rs1,imm</td>
<td>Branch &lt; Signed</td>
<td>B  BLTU rs1,rs2,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OR Immediate</td>
<td>I</td>
<td>ORI rd,rs1,imm</td>
<td>Branch ≥ Signed</td>
<td>B  BGEU rs1,rs2,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AND Immediate</td>
<td>I</td>
<td>ANDI rd,rs1,imm</td>
<td>Jump &amp; Link Register</td>
<td>JAL rd,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Compare</td>
<td>R</td>
<td>SLT rd,rs1,rs2</td>
<td>Synch Thread</td>
<td>I</td>
<td>FENCE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I</td>
<td>SLTI rd,rs1,imm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R</td>
<td>SLTU rd,rs1,rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I</td>
<td>SLTIU rd,rs1,imm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Environment**
- CALL
- BREAK
- EBREAK

*Not in 61C*
Building a RISC-V Processor
On every tick of the clock, the computer executes one instruction.

Current state outputs drive the inputs to the combinational logic, whose outputs settles at the values of the state before the next clock edge.

At the rising clock edge, all the state elements are updated with the combinational logic outputs, and execution moves to the next clock cycle.
Stages of the Datapath: Overview

• Problem: a single, “monolithic” block that “executes an instruction” (performs all necessary operations beginning with fetching the instruction) would be too bulky and inefficient

• Solution: break up the process of “executing an instruction” into stages, and then connect the stages to create the whole datapath
  – smaller stages are easier to design
  – easy to optimize (change) one stage without touching the others (modularity)
Five Stages of the Datapath

- Stage 1: Instruction Fetch (IF)
- Stage 2: Instruction Decode (ID)
- Stage 3: Execute (EX) - ALU (Arithmetic-Logic Unit)
- Stage 4: Memory Access (MEM)
- Stage 5: Write Back to Register (WB)
Basic Phases of Instruction Execution

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory Access
5. Register Write
Datapath Components: Combinational

- Combinational elements

- Storage elements + clocking methodology

- Building blocks
Datapath Elements: State and Sequencing (1/3)

- **Register**
- **Write Enable:**
  - Low (or deasserted) (0): Data Out will not change
  - Asserted (1): Data Out will become Data In on positive edge of clock
Register file (regfile, RF) consists of 32 registers:
- Two 32-bit output busses: busA and busB
- One 32-bit input bus: busW

Register is selected by:
- RA (number) selects the register to put on busA (data)
- RB (number) selects the register to put on busB (data)
- RW (number) selects the register to be written via busW (data) when Write Enable is 1

Clock input (Clk)
- Clk input is a factor ONLY during write operation
- During read operation, behaves as a combinational logic block:
  - RA or RB valid $\Rightarrow$ busA or busB valid after “access time.”
“Magic” Memory
- One input bus: Data In
- One output bus: Data Out

Memory word is found by:
- For Read: Address selects the word to put on Data Out
- For Write: Set Write Enable = 1: address selects the memory word to be written via the Data In bus

Clock input (CLK)
- CLK input is a factor ONLY during write operation
- During read operation, behaves as a combinational logic block: Address valid ⇒ Data Out valid after “access time”
State Required by RV32I ISA (1/2)

Each instruction during execution reads and updates the state of: (1) Registers, (2) Program counter, (3) Memory

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

- Program Counter (\textit{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)
  - These are placeholders for instruction and data caches
- Instructions are read (fetched) from instruction memory (assume IMEM read-only)
- Load/store instructions access data memory
R-Type Add Datapath
### Review: R-Type Instructions

#### R-format: ALU

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0|

- **func7**: 7
- **rs2**: 5
- **rs1**: 5
- **func3**: 3
- **rd**: 5
- **opcode**: 7

---

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0000000 | rs2 | rs1 | 000 : ADD | rd | 0110011:OP-R |
| 0100000 | rs2 | rs1 | 000 : SUB | rd | 0110011:OP-R |
| 0000000 | rs2 | rs1 | 001 : SLL | rd | 0110011:OP-R |
| 0000000 | rs2 | rs1 | 010 : SLT | rd | 0110011:OP-R |
| 0000000 | rs2 | rs1 | 011 : SLTU | rd | 0110011:OP-R |
| 0000000 | rs2 | rs1 | 100 : XOR | rd | 0110011:OP-R |
| 0000000 | rs2 | rs1 | 101 : SRL | rd | 0110011:OP-R |
| 0100000 | rs2 | rs1 | 101 : SRA | rd | 0110011:OP-R |
| 0000000 | rs2 | rs1 | 110 : OR | rd | 0110011:OP-R |
| 0000000 | rs2 | rs1 | 111 : AND | rd | 0110011:OP-R |

---

- **E.g. Addition/subtraction**

  - **add rd, rs1, rs2**
    
    $$R[rd] = R[rs1] + R[rs2]$$

  - **sub rd, rs1, rs2**
    
    $$R[rd] = R[rs1] - R[rs2]$$
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

\[ \text{Reg}[rd] = \text{Reg}[rs1] + \text{Reg}[rs2] \]
Timing Diagram for add

PC | 1000 | 1004
---|---|---
PC+4 | 1004 | 1008
inst[31:0] | add x1,x2,x3 | add x6,x7,x9

time
Sub Datapath
Implementing the `sub` instruction

<table>
<thead>
<tr>
<th></th>
<th>rs2</th>
<th>rs1</th>
<th>000</th>
<th>rd</th>
<th></th>
<th>add</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>rs2</th>
<th>rs1</th>
<th>000</th>
<th>rd</th>
<th></th>
<th>sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
<td></td>
</tr>
</tbody>
</table>

```
sub rd, rs1, rs2
```

- 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

\[
PC = PC + 4
\]

\[
Reg[rd] = Reg[rs1] +/− Reg[rs2]
\]
## 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>0000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>001</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>010</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>011</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>100</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0100000</td>
<td></td>
<td></td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>110</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>111</td>
<td>rd</td>
<td>0110011</td>
</tr>
</tbody>
</table>

All implemented by decoding funct3 and funct7 fields and selecting appropriate ALU function:
- `add`
- `sub`
- `sll`
- `slt`
- `sltu`
- `xor`
- `srl`
- `sra`
- `or`
- `and`
Datapath With Immediates
Implementing I-Format - addi instruction

- RISC-V Assembly Instruction:
  \[ \text{addi } x15, x1, -50 \]

<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>111111001110</td>
<td>00001</td>
<td>000</td>
<td>01111</td>
<td>0010011</td>
</tr>
</tbody>
</table>

- \( \text{imm} = -50 \)
- \( \text{rs1} = 1 \)
- \( \text{add} \)
- \( \text{rd} = 15 \)
- OP-Imm
Datapath for add/sub

\[ PC = PC + 4 \]

\[ Reg[rd] = Reg[rs1] + Imm \]

Control logic

RegWriteEnable (RegWEn) = 1

ALU Sel (add=0/sub=1)

Immediate should be here
Adding addi to Datapath

\[ \text{PC} = \text{PC} + 4 \]

\[ \text{Reg}[\text{rd}] = \text{Reg}[\text{rs1}] + \text{Imm} \]

Control logic:
- RegWriteEnable (RegWEn)=1
- BSel (rs2=0/Imm=1)
- ALUSel (add=0/sub=1)

<table>
<thead>
<tr>
<th>Imm[11:0]</th>
<th>rs1</th>
<th>000</th>
<th>rd</th>
<th>0010011</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>
Adding `addi` to Datapath

\[
PC = PC + 4
\]

\[
\text{Reg}[rd] = \text{Reg}[rs1] + \text{Imm}
\]

- **PC** = Program Counter
- **IMEM** = Instruction Memory
- **Addr** = Address
- **Inst** = Instruction
- **Reg** = Register
- **alu** = Arithmetic Logic Unit

**Control logic**:
- **ImmSel** = 1
- **RegWriteEnable** (RegWEn) = 1
- **BSel** (rs2=0/Imm=1)
- **ALUSel** (add=0/sub=1)
Adding `addi` to Datapath

\[ \text{PC} = \text{PC} + 4 \]

\[ \text{Reg[rd]} = \text{Reg[rs1]} + \text{Imm} \]

- \( \text{Inst}[31:0] \)
- \( \text{AddrA} \)
- \( \text{AddrB} \)
- \( \text{DataA} \)
- \( \text{DataB} \)
- \( \text{Reg[rs1]} \)
- \( \text{Reg[rs2]} \)
- \( \text{Imm[31:0]} \)
- \( \text{RegWriteEnable} = \text{RegWEn} = 1 \)
- \( \text{BSel} = \text{rs2} = 0/\text{Imm} = 1 \)
- \( \text{ALUSel} = \text{add} = 0/\text{sub} = 1 \)

**Control logic:**
- \( \text{ImmSel} = 1 \)
- \( \text{RegWriteEnable} = \text{RegWEn} = 1 \)
- \( \text{BSel} = \text{rs2} = 0/\text{Imm} = 1 \)
- \( \text{ALUSel} = \text{add} = 0/\text{sub} = 1 \)
I-Format Immediates

- inst[31] -

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

Works for all other I-format arithmetic instructions (`slti, sltiu, andi, ori, xori, slli, srli, srai`) just by changing ALUSel.