CS61C
Great Ideas in Computer Architecture (a.k.a. Machine Structures)

RISC-V Processor Design

cs61c.org
Machine Structures

CS61C

Software

- Application (ex: browser)
- Operating System
  - Compiler
  - Assembler

Hardware

- Processor
- Memory
- I/O system
- Datapath & Control
- Digital Design
- Circuit Design
- Transistors
- Fabrication

Instruction Set Architecture

RISC-V (2)
**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
- **Harness Parallelism & Achieve High Performance**

### Hardware descriptions
- All gates work in parallel at same time

---

**Smart Phone Warehouse Scale Computer**

**Computer**
- **Core**
- **Memory**
  - (Cache)
- **Input/Output**
- **Exec. Unit(s)**
- **Functional Block(s)**
- **Main Memory**
- **Logic Gates**

**RISC-V (3)**
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)
- Logic Circuit Description (Circuit Schematic Diagrams)

```
# Great Idea #1: Abstraction
levels = 
  High Level Language Program (e.g., C)
  Assembly Language Program (e.g., RISC-V)
  Machine Language Program (RISC-V)
  Hardware Architecture Description (e.g., block diagrams)
  Logic Circuit Description (Circuit Schematic Diagrams)
```

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
lw x3, 0(x10)
lw x4, 4(x10)
sw x4, 0(x10)
sw x3, 4(x10)
```

```
1000 1101 1110 0010 0000 0000 0000 0000
1000 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
```

```
IMEM
ALU
Imm.
Gen
```

```
DMEM
Branch
Comp.
Reg []
AddrA
AddrB
DataA
AddrD
DataB
DataD

1
0
0
1
2
1
0
```

```
alu
mem
wb
alu
pc+4
Reg [rs1]
imm[31:0]
Reg [rs2]
wb
```
Our Single-Core Processor So Far...

Processor
- Control

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

Memory
- Program
- Data

Enable? Read/Write
Address
Write Data
Read Data

Input
Output
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><strong>Shift Left Logical</strong></td>
<td>R</td>
<td>SLL rd,rs1,rs2</td>
<td><strong>Shift Right Logical</strong></td>
<td>R SRL rd,rs1,rs2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I</td>
<td>SLLI rd,rs1,shamt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Right Log. Imm.</td>
<td>R</td>
<td>SRLI rd,rs1,shamt</td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Right Arithmetic</td>
<td>R</td>
<td>SRA rd,rs1,rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Right Arithmetic</td>
<td>I</td>
<td>SRAI rd,rs1,shamt</td>
<td>Stores</td>
<td>S SB rs1,rs2,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Arithmetic</td>
<td>ADD</td>
<td>ADD rd,rs1,rs2</td>
<td></td>
<td>ADDI rd,rs1,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADD Immediate</td>
<td>I</td>
<td>ADDI rd,rs1,imm</td>
<td></td>
<td>SH rs1,rs2,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADD Immediate</td>
<td></td>
<td>ADDI rd,rs1,imm</td>
<td>Store Word</td>
<td>SW rs1,rs2,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADD Immediate</td>
<td></td>
<td>ADDI rd,rs1,imm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUBtract</td>
<td>R</td>
<td>SUB rd,rs1,rs2</td>
<td>Branches</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></td>
<td>Logical</td>
<td>XOR</td>
<td>XOR rd,rs1,rs2</td>
<td>Branch ≥</td>
<td>B BGE rs1,rs2,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XOR Immediate</td>
<td>I</td>
<td>XORI rd,rs1,imm</td>
<td>Branch &lt; Unsigned</td>
<td>B BLTU rs1,rs2,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OR</td>
<td>R</td>
<td>OR rd,rs1,rs2</td>
<td>Branch ≥ Unsigned</td>
<td>B BGEU 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>Jump &amp; Link</td>
<td>J JAL rd,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AND</td>
<td>R</td>
<td>AND rd,rs1,rs2</td>
<td>Jump &amp; Link Register</td>
<td>I JALR rd,rs1,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AND Immediate</td>
<td>I</td>
<td>ANDI rd,rs1,imm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Compare</td>
<td>R</td>
<td>SLT rd,rs1,rs2</td>
<td>Synch</td>
<td>I FENCE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set &lt; Immediate</td>
<td>I</td>
<td>SLTI rd,rs1,imm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set &lt; Unsigned</td>
<td>R</td>
<td>SLTU rd,rs1,rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set &lt; Imm Unsigned</td>
<td>I</td>
<td>SLTIU rd,rs1,imm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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.
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
  - Adder
  - Multiplexer
  - ALU

- Storage elements + clocking methodology
- Building blocks
- 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 ⇒ 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”
Each instruction during execution reads and updates the state of: (1) Registers, (2) Program counter, (3) Memory

- **Registers (x0..x31)**
  - Register file (regfile) \( \text{Reg} \) holds 32 registers x 32 bits/register: \( \text{Reg}[0]..\text{Reg}[31] \)
  - 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 \( \text{Reg}[0] \) 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)
  - 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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Func3</th>
<th>Rs2</th>
<th>Rs1</th>
<th>Func7</th>
<th>Rd</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td></td>
<td>Rs2</td>
<td>Rs1</td>
<td></td>
<td>000</td>
<td>ADD</td>
</tr>
<tr>
<td>0100000</td>
<td></td>
<td>Rs2</td>
<td>Rs1</td>
<td></td>
<td>000</td>
<td>SUB</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td>Rs2</td>
<td>Rs1</td>
<td></td>
<td>010</td>
<td>SLT</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td>Rs2</td>
<td>Rs1</td>
<td></td>
<td>100</td>
<td>XOR</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td>Rs2</td>
<td>Rs1</td>
<td></td>
<td>101</td>
<td>SRL</td>
</tr>
<tr>
<td>0100000</td>
<td></td>
<td>Rs2</td>
<td>Rs1</td>
<td></td>
<td>101</td>
<td>SRA</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td>Rs2</td>
<td>Rs1</td>
<td></td>
<td>110</td>
<td>OR</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td>Rs2</td>
<td>Rs1</td>
<td></td>
<td>111</td>
<td>AND</td>
</tr>
</tbody>
</table>

- **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:
  - \( \text{Reg}[\text{rd}] = \text{Reg}[\text{rs1}] + \text{Reg}[\text{rs2}] \)
  - \( \text{PC} = \text{PC} + 4 \)
Datapath for add

PC = PC + 4

Reg[rd] = Reg[rs1] + Reg[rs2]

control logic

RegWriteEnable (RegWEn) = 1

IMEM

addr

inst

pc+4

clk

pc

Add

+4

alu

data

Reg[rd] = Reg[rs1] + Reg[rs2]

Reg[rs1]

Reg[rs2]

Reg[rd] = Reg[rs1] + Reg[rs2]

Inst[31:0]

Inst[11:7]

Inst[19:15]

Inst[24:20]

Inst[20:15]

Inst[14:0]

funct7

rs2

rs1

funct3

rd

opcode

RISC-V (22)
Timing Diagram for \texttt{add}

- \texttt{PC}:
  - 1000
  - 1004
  - 1008

- \texttt{inst[31:0]}:
  - \texttt{add x1, x2, x3}
  - \texttt{add x6, x7, x9}

- \texttt{Reg[rs1]}:
  - Reg[2]
  - Reg[7]

- \texttt{Reg[rs2]}:
  - Reg[3]
  - Reg[9]

- \texttt{alu}:

- \texttt{Reg[1]}:
  - ???
Sub Datapath
Implementing the `sub` instruction

- `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

Reg[rd] = Reg[rs1] +/- Reg[rs2]

PC = PC + 4

RegWriteEnable (RegWEn) = 1
Implementing Other R-Format Instructions

<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>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>001</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>011</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</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.
Datapath With Immediates
## Implementing I-Format - `addi` instruction

- **RISC-V Assembly Instruction:**
  ```assembly
caldi  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>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

| 111111001110 | 00001 | 000 | 01111 | 0010011 |

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

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

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

Control logic

RegWriteEnable (RegWEn) = 1

ALUSel (add=0/sub=1)

Immediate should be here

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

\[
\text{PC} = \text{PC} + 4
\]
Adding addi to Datapath

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

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

- **ALUSel**: (add=0/sub=1)
- **RegWriteEnable**: (RegWEn)=1

**Control logic**

- **Reg\[\text{rd}\]** = Reg\[\text{rs1}\] + Imm
- **PC** = PC + 4

**IMEM**

- Inst[31:0]
- Inst[11:7]
- Inst[19:15]
- Inst[24:20]

**Add**

- AddrA
- AddrB

**ALU**

- DataA
- DataB
- DataD

- Imm[31:0]
- BSel (rs2=0/Imm=1)
- ALUSel (add=0/sub=1)

**Register**

- Reg\[\text{rs1}\]
- Reg\[\text{rs2}\]

**Multiplexer**

- AddrD

**Routing**

- Imm[31:0]
- ALUSel (add=0/sub=1)
Adding addi to Datapath

Reg[rd] = Reg[rs1] + Imm

PC = PC + 4

Control logic

RegWriteEnable (RegWEn)=1
BSel (rs2=0/Imm=1)
ALUSel (add=0/sub=1)

ImmSel = I

Inst[31:0]

Inst[31:20]

Inst[24:20]

Inst[19:15]

Inst[11:7]

IMEM

PC

addr

inst

Add

Reg[rd] = Reg[rs1] + Imm

DataD

AddrD

DataA

AddrA

DataB

AddrB

Reg[1]

ALU

alu

AddrD

AddrA

AddrB

DataD

DataA

DataB

Imm[31:0]

Reg[rs1]

Reg[rs2]
Adding `addi` to Datapath

\[ PC = PC + 4 \]

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

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

**Ports**
- **Inst** [31:0]
- **Addr** [24:20]
- **Addr** [19:15]
- **Addr** [11:7]
- **Data** [31:0]
- **Reg** [31:0]
- **Imm** [31:0]
- **Bsel** = 1

**Logic Gates**
- **Add**
- **Imm. Gen**
- **IMEM**
- **alu**
- **clk**
- **pc+4**
I-Format Immediates

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

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