Transistors

- MOS - Metal-Oxide Semiconductor
- MOS transistors have three terminals: drain, gate, and source
- A transistor acts as switches:
  - If the voltage on the gate terminal is higher/lower than the source terminal then a conducting path will be established between the drain and source terminals

Transistors

- CMOS - Complementary MOS
- Transistors are the primary components of ICs
- An integrated circuit (IC) or a chip is made up of transistors (these days billions) and other electronic components
  - ICs are the building blocks of computers (CPU, bus interface, memory management unit)
The base RISC-V ISA has four main instruction formats:
- R, I, S and U types.

In the base ISA, there are four core instruction formats (R/I/S/U), as shown in Figure 2.2. All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. An instruction address misaligned exception is generated on a taken branch or unconditional jump if the target address is not four-byte aligned. No instruction fetch misaligned exception is generated for a conditional branch that is not taken.

The alignment constraint for base ISA instructions is relaxed to a two-byte boundary when instruction extensions with 16-bit lengths or other odd multiples of 16-bit lengths are added.

The RISC-V ISA keeps the source (rs1 and rs2) and destination (rd) registers at the same position in all formats to simplify decoding. Except for the 5-bit immediates used in CSR instructions (Section 2.8), immediates are always sign-extended, and are generally packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry.

Decoding register specifiers is usually on the critical paths in implementations, and so the instruction format was chosen to keep all register specifiers at the same position in all formats at the expense of having to move immediate bits across formats (a property shared with RISC-IV aka. SPUR [18]).

In practice, most immediates are either small or require all XLEN bits. We chose an asymmetric immediate split (12 bits in regular instructions plus a special load upper immediate instruction with 20 bits) to increase the opcode space available for regular instructions.

Immediates are sign-extended because we did not observe a benefit to using zero-extension for some immediates as in the MIPS ISA and wanted to keep the ISA as simple as possible.

**Central Processing Unit (CPU)**

- Central Processing Unit (CPU) Organization
  - CPU = Control Unit + ALU + Registers
  - Control Unit: monitors and directs sequences of instructions
  - ALU (Arithmetic-Logic Unit): performs arithmetic and logical operations

- CPU Execution Process
  1. Fetch Instruction
  2. Decode Instruction
  3. Execute Operation
  4. Memory Operation
  5. Register Writeback Operation
Central Processing Unit (CPU)

- Central Processing Unit (CPU) Organization
- CPU Execution Process
  1. Fetch Instruction
     - Read IM[PC]
  2. Decode
  3. Increment PC
  4. Read registers
  5. ALU Operation
     - Or Branch Address
     - Data Memory Operation
  6. Write Back

Execute Operation

- The Arithmetic Logic Unit (ALU) is at the center of the CPU operation execution
- ALU operation is based on instruction type and function code
  - Performs subtraction for branches (beq)
  - Performs no operation for jumps
  - Performs the operation is specified by the function field for R-type instructions
- ALU Control unit will have the following inputs:
  - 3-bit control field called ALUOp
  - Funct3 and funct7 function fields
Memory Operation

- For RISC-V Load and Store are the only two memory instructions

- Recall:
  - RISC-V does not support memory to memory data processing operations
  - Data values must be moved into registers before using them
  - The basic load and store instructions are Load and Store Word or Byte
    - lw/lb rd, offset(rs1)
    - sw/sh rs2, offset(rs1)

CPU Instruction Execution Stages

<table>
<thead>
<tr>
<th>Stage</th>
<th>R-Type</th>
<th>Memory Reference</th>
<th>Branches</th>
<th>Jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td></td>
<td>Instruction[31:0] ● Memory[PC]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction Decode</td>
<td></td>
<td>Result ● Reg. 19:20 ● 10/11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution Operation</td>
<td></td>
<td>Reg. 11:0 ● ALU_Result</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory Access</td>
<td></td>
<td>Load ● Reg. 11:0 ● Instruction[11:0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Writeback</td>
<td></td>
<td>Reg. 11:0 ● Mem[ALU_Result]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5-Stage RISC-V Pipelining

1. Instruction Fetch
2. Instruction Decode
3. Execute
4. Memory
5. WriteBack
Instruction Interactions

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline
  - Structural hazard
- An instruction may depend on something produced by an earlier instruction
  - Dependence may be for a data calculation
  - Data hazard
  - Dependence may be for calculating the next address
  - Control hazard (branches, interrupts)

Resolving Data Hazards

- Strategy 1: Wait for the result to be available by freezing earlier pipeline stages
  - Interlocks
- Strategy 2: Route data as soon as possible after it is calculated to the earlier pipeline stage
  - Bypass
Resolving Data Hazards

- Strategy 3: Speculate on the dependence
  - Two cases:
    - Guessed correctly
    - Do nothing
    - Guessed incorrectly
      - Kill and restart

Source and Destination Registers

```
<table>
<thead>
<tr>
<th>Type</th>
<th>ALU</th>
<th>ALUi</th>
<th>LW</th>
<th>SW</th>
<th>AUIPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALUi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AUIPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- ALU: rd ← (rs1) [func3,func7] (rs2) → rs1, rs2 → rd
- ALUi: rd ← (rs1) [func3] I-imm → rs1 → rd
- LW: rd ← M [rs1] + imm → rs1 → rd
- SW: M [rs1] + imm → (rs2) → rs1, rs2
- AUIPC: rd ← pc + U-imm → rd
### Source and Destination Registers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ALU</th>
<th>ALU</th>
<th>LW</th>
<th>SW</th>
<th>LUI</th>
<th>AUIPC</th>
<th>JAL</th>
<th>JALR</th>
<th>BR</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd</td>
<td>(rs1) [func3, func7] (n2)</td>
<td>rs1, rs2</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
</tr>
<tr>
<td>rd</td>
<td>(rs1) [func3, inst[30:1] imm[4:0]]</td>
<td>rs1</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
</tr>
<tr>
<td>rd</td>
<td>M [rs1] + imm</td>
<td>rs1</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
</tr>
<tr>
<td>rd</td>
<td>M [rs1] + imm</td>
<td>rs1</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
</tr>
<tr>
<td>rd</td>
<td>U- imm</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
</tr>
<tr>
<td>rd</td>
<td>pc + U- imm</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
</tr>
<tr>
<td>rd</td>
<td>pc + 4</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
</tr>
<tr>
<td>rd</td>
<td>pc + 4</td>
<td>rs1</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
</tr>
<tr>
<td>rd</td>
<td>pc + 4</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
</tr>
<tr>
<td>rs1, rs2</td>
<td>rs1, rs2</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
<td>rd</td>
</tr>
</tbody>
</table>

### Types of Data Hazards

- **Consider executing a sequence of** $r_i \leftarrow (r_j) \text{ op } (r_k)$ **type of instructions**

  **Data-dependence**
  
  $r_i \leftarrow r_j \text{ op } r_k$  \text{ Read-after-Write (RAW) hazard}

  $r_i \leftarrow r_j \text{ op } r_k$  \text{ Write-after-Read (WAR) hazard}

  $r_i \leftarrow r_j \text{ op } r_k$  \text{ Write-after-Write (WAW) hazard}

### Data Hazards: An Example

- $I_1$ ADD $x_6, x_6, x_4$
- $I_2$ LW $x_2, 44(x_3)$
- $I_3$ SUB $x_5, x_2, x_4$
- $I_4$ AND $x_8, x_5, x_2$
- $I_5$ SUB $x_{10}, x_5, x_6$
- $I_6$ ADD $x_6, x_8, x_2$

**RAW Hazards**

**WAR Hazards**

**WAW Hazards**
Stalled Stages and Pipeline

- Pipelining was introduced to improve performance

<table>
<thead>
<tr>
<th>Time</th>
<th>Instructions</th>
<th>Cycles</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Program * Instruction * Cycle</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- But with stalling, CPI will go up
- So we need a way to reduce stalling cycles!

Feedback to Resolve Hazards

- Later stages provide dependence information to earlier stages which can stall (or kill) instructions
- Controlling a pipeline in this manner works provided the instruction at stage \( i+1 \) can complete without any interference from instructions in stages 1 to \( i \)
- Otherwise deadlocks may occur

Fully Bypassed Datapath
Why a program may have CPI >1

- Why an Instruction may not be dispatched every cycle (CPI>1)?
  - Full bypassing may be too expensive to implement
  - Typically all frequently used paths are provided
  - Some infrequently used bypass paths may increase cycle time and counteract the benefit of reducing CPI
- Loads have two cycle latency
  - Instruction after load cannot use load result
- Some ISA define load delay slots, a software-visible pipeline hazard (compiler schedules independent instruction or inserts NOP to avoid hazard)

RISC-V CPU Stages

- Single Cycle CPU Instruction Execution

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory Access</th>
<th>Writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td>2ns</td>
<td>3ns</td>
<td>3ns</td>
<td>2ns</td>
<td>1ns</td>
</tr>
</tbody>
</table>

Cycle Time: 2ns + 3ns + 3ns + 2ns + 1ns = 11ns

Total Execution Time: 8 instruction * 11ns = 88 ns
RISC-V CPU Stages

- Multi-Cycle CPU Instruction Execution

Cycle Time: \( \text{Max}(2\text{ns}, 3\text{ns}, 3\text{ns}, 2\text{ns}, 1\text{ns}) = 3\text{ns} \)

Idle Stages

Cycle Time: \( \text{Max}(2\text{ns}, 3\text{ns}, 3\text{ns}, 2\text{ns}, 1\text{ns}) = 3\text{ns} \)
RISC-V CPU Stages

- Multi-Cycle CPU Instruction Execution

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>CPU stages used by the instruction class</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>Instruction fetch, Decode, Execute, Writeback</td>
</tr>
<tr>
<td>Load word</td>
<td>Instruction fetch, Decode, Execute, Memory Access, Writeback</td>
</tr>
<tr>
<td>Store word</td>
<td>Instruction fetch, Decode, Execute, Memory Access</td>
</tr>
<tr>
<td>Branch</td>
<td>Instruction fetch, Decode, Execute</td>
</tr>
<tr>
<td>Jump</td>
<td>Instruction fetch, Decode</td>
</tr>
</tbody>
</table>

Cycle Time: Max(2ns, 3ns, 3ns, 2ns, 1ns) = 3ns

- Idle Stages

<table>
<thead>
<tr>
<th>I</th>
<th>1st Instruction</th>
<th>2nd Instruction</th>
<th>3rd Instruction</th>
<th>4th Instruction</th>
<th>5th Instruction</th>
<th>6th Instruction</th>
<th>7th Instruction</th>
<th>8th Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>3ns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3ns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3ns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3ns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total Execution Time: 90 ns
Pipelined Multi-Cycle CPU Instruction Execution

RISC-V CPU Stages

Instruction Fetch
Instruction Decode
Execute
Memory Access
Writeback

3ns
3ns
3ns
3ns
3ns

Cycle Time: Max(2ns, 3ns, 3ns, 2ns, 1ns) = 3ns

I_1
addi x3,x0,12

I_2
lw x1,4(x2)

I_3
slli x5,x1,2

I_4
addi x3,x0,12

I_5
lw x1,4(x2)

I_6
slli x5,x1,2

I_7
add x3,x3,x3

I_8
bge x6,x5,Loop

I_9
addi x5,x5,-1

I_10
j Exit

Loop:

I_11
lw x1,4(x2)

I_12
slli x5,x1,2

I_13
add x3,x3,x3

I_14
bge x6,x5,Loop

I_15
addi x5,x5,-1

I_16
j Exit

Exit:

I_17
addi x5,x0,8

Total Execution Time: 15 time steps * 3ns = 45 ns
RISC-V CPU Stages

- Pipelined With Bypass Multi-Cycle CPU Instruction Execution

Instructions:
- Instruction Fetch
- Instruction Decode
- Execute
- Memory Access
- Writeback

Time:
- $t_0$
- $t_1$
- $t_2$
- $t_3$
- $t_4$
- $t_5$
- $t_6$
- $t_7$
- $t_8$
- $t_9$
- $t_{10}$
- $t_{11}$
- $t_{12}$
- $t_{13}$
- $t_{14}$

Loop:
- addi $x3$, $x0$, 12

Exit:
- addi $x5$, $x0$, 8

Total Execution Time: 13 time steps * 3ns = 39ns
Performance Measurement

- Processor performance:
  - Execution time
  - Area
  - Logic complexity
  - Power

<table>
<thead>
<tr>
<th>Time</th>
<th>Instructions</th>
<th>Cycles</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Program</td>
<td>Instruction</td>
<td>Cycle</td>
</tr>
</tbody>
</table>

- In this class we will focus on Execution time

Amdahl's Law

- By Gene Amdahl
- This law answers the critical question:
  - How much of a speedup one can get for a given architectural improvement/enhancement?
  - The performance enhancement possible due to a given design improvement is limited by the amount that the improved feature is used

- Performance improvement or speedup due to enhancement $E$

$$\text{Speedup}(E) = \frac{\text{Execution Time without } E}{\text{Execution Time with } E} = \frac{\text{Performance without } E}{\text{Performance with } E}$$

- Suppose that enhancement $E$ accelerates a fraction $F$ of the execution time by a factor $S$ and the remainder of the time is unaffected then:
  - Execution Time with $E = (1 - F) + \frac{F}{S} \times$ Execution Time without $E$
  - Hence speedup is given by:

$$\text{Speedup}(E) = \frac{1}{(1 - F) + \frac{F}{S} \times \text{Execution Time without } E} = \frac{1}{(1 - F) + \frac{F}{S}}$$
CPU Performance

- CPU performance factors
  - Instruction count
  - Determined by ISA and compiler
  - CPI and Cycle time
  - Determined by CPU hardware
  - Longest delay determines clock period
    - Critical path: load instruction

- Longest delay determines clock period
  - Critical path: load instruction
    1. Instruction memory
    2. Register file read
    3. ALU operation
    4. Data memory access
    5. Register file writeback

- Performance can be improved by pipelining

Next Class

- Memory Organization