Control
and Status
Registers
Complete Single-Cycle RV32I Datapath!
Control and Status Registers

- Control and status registers (CSRs) are separate from the register file ($x0-x31$)
  - Used for monitoring the status and performance
  - There can be up to 4096 CSRs
- Not in the base ISA, but almost mandatory in every implementation
  - ISA is modular
  - Necessary for counters and timers, and communication with peripherals
# CSR Instructions

<table>
<thead>
<tr>
<th>Instr.</th>
<th>rd</th>
<th>rs</th>
<th>Read CSR?</th>
<th>Write CSR?</th>
</tr>
</thead>
<tbody>
<tr>
<td>csrrw</td>
<td>x0</td>
<td>-</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>csrrw</td>
<td>!x0</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>csrrs/c</td>
<td>-</td>
<td>x0</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>csrrs/c</td>
<td>-</td>
<td>!x0</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**CSR Instructions**

<table>
<thead>
<tr>
<th>source/dest</th>
<th>source</th>
<th>instr</th>
<th>rd</th>
<th>SYSTEM</th>
<th>uimm[4:0]</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1110011</td>
</tr>
</tbody>
</table>

<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>csr</td>
<td>rsl</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>
**CSR Instructions**

<table>
<thead>
<tr>
<th>Instr.</th>
<th>rd</th>
<th>uimm</th>
<th>Read CSR?</th>
<th>Write CSR?</th>
</tr>
</thead>
<tbody>
<tr>
<td>csrrwi</td>
<td>x0</td>
<td>–</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>csrrwi</td>
<td>!x0</td>
<td>–</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>csrrs/ci</td>
<td></td>
<td>0</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>csrrs/ci</td>
<td></td>
<td>!0</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Source/dest source instr rd SYSTEM

Zero-extended to 32b
The CSRRW (Atomic Read/Write CSR) instruction ‘atomically’ swaps values in the CSRs and integer registers.
- We will see more on ‘atomics’ later

CSRRW reads the previous value of the CSR and writes it to integer register \( rd \). Then writes \( rs1 \) to CSR

Pseudoinstruction \( \text{csrw } csr, rs1 \) is \( \text{csrrw } x0, csr, rs1 \)
- \( rd=x0 \), just writes \( rs1 \) to CSR

Pseudoinstruction \( \text{csrwi } csr, uimm \) is \( \text{csrrwi } x0, csr, uimm \)
- \( rd=x0 \), just writes \( uimm \) to CSR

Hint: Use write enable and clock…
System Instructions

- **ecall** – (I-format) makes requests to supporting execution environment (OS), such as system calls (syscalls)

- **ebreak** – (I-format) used e.g. by debuggers to transfer control to a debugging environment

- **fence** – sequences memory (and I/O) accesses as viewed by other threads or co-processors
Datapath Control
Our Single-Core Processor

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

Memory
- Program
  - Bytes
- Data

Input
Output

Enable?
Read/Write
Address
Write Data
Read Data
Single-Cycle RV32I Datapath and Control

[Diagram of the Single-Cycle RV32I Datapath and Control.]
Example: \textit{sw}
Instruction Timing
Example: add
Add Execution

Clock
PC
PC+4
inst[31:0]
Control logic
Reg[rs1]
Reg[rs2]
alu
wb
Reg[1]

1000
1004
1008
add x1,x2,x3
add x6,x7,x9
add control
Reg[2]
Reg[3]
Example: \texttt{add} timing

Critical path = \( t_{\text{clk-q}} + \max \{ t_{\text{Add}} + t_{\text{mux}}, t_{\text{IMEM}} + t_{\text{Reg}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{mux}} \} + t_{\text{setup}} \)

\begin{align*}
  &= t_{\text{clk-q}} + t_{\text{IMEM}} + t_{\text{Reg}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{mux}} + t_{\text{setup}}
\end{align*}
Example: $1_w$

Critical path = $t_{clk-q} + \max\{t_{Add} + t_{mux}, t_{IMEM} + t_{imm} + t_{mux}, t_{ALU} + t_{DMEM} + t_{mux}, t_{IMEM} + t_{Reg} + t_{mux}, t_{ALU} + t_{DMEM} + t_{mux}\} + t_{setup}$
Instruction Timing

Clock: IF | ID | EX | MEM | WB | Total

PC: old | Reg Read | pc | ALU | D-MEM | Reg W | pc+4

Instr. fetch: old | 200 ps

Instr. decode: old | register out

Execute: old | ALU result

Memory Access: old | memory data

$t_{IF}$ | $t_{ID}$ | $t_{EX}$ | $t_{MEM}$ | $t_{WB}$
**Instruction Timing**

<table>
<thead>
<tr>
<th>Instr</th>
<th>IF = 200ps</th>
<th>ID = 100ps</th>
<th>ALU = 200ps</th>
<th>MEM = 200ps</th>
<th>WB = 100ps</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
<tr>
<td>jal</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>600ps</td>
</tr>
<tr>
<td>lw</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>700ps</td>
</tr>
</tbody>
</table>

- **Maximum clock frequency**
  - $f_{\text{max}} = 1/800\text{ps} = 1.25 \text{ GHz}$
- **Most blocks idle most of the time**
  - E.g. $f_{\text{max,ALU}} = 1/200\text{ps} = 5 \text{ GHz}$!
## Control Logic Truth Table

<table>
<thead>
<tr>
<th>Inst[31:0]</th>
<th>BrEq</th>
<th>BrLT</th>
<th>PCSel</th>
<th>ImmSel</th>
<th>BrUn</th>
<th>ASel</th>
<th>BSel</th>
<th>ALUSel</th>
<th>MemRW</th>
<th>RegWEn</th>
<th>WBSel</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>*</td>
<td>*</td>
<td>Reg</td>
<td>Reg</td>
<td>Add</td>
<td>Read</td>
<td>1</td>
<td>ALU</td>
</tr>
<tr>
<td>sub</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>*</td>
<td>*</td>
<td>Reg</td>
<td>Reg</td>
<td>Sub</td>
<td>Read</td>
<td>1</td>
<td>ALU</td>
</tr>
<tr>
<td>(R–R Op)</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>*</td>
<td>*</td>
<td>Reg</td>
<td>Reg</td>
<td>(Op)</td>
<td>Read</td>
<td>1</td>
<td>ALU</td>
</tr>
<tr>
<td>addi</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>I</td>
<td>*</td>
<td>Reg</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>1</td>
<td>ALU</td>
</tr>
<tr>
<td>lw</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>I</td>
<td>*</td>
<td>Reg</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>1</td>
<td>Mem</td>
</tr>
<tr>
<td>sw</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>S</td>
<td>*</td>
<td>Reg</td>
<td>Imm</td>
<td>Add</td>
<td>Write</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>*</td>
<td>+4</td>
<td>B</td>
<td>*</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>beq</td>
<td>1</td>
<td>*</td>
<td>ALU</td>
<td>B</td>
<td>*</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>bne</td>
<td>0</td>
<td>*</td>
<td>ALU</td>
<td>B</td>
<td>*</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>bne</td>
<td>1</td>
<td>*</td>
<td>+4</td>
<td>B</td>
<td>*</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>blt</td>
<td>*</td>
<td>1</td>
<td>ALU</td>
<td>B</td>
<td>0</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>bltu</td>
<td>*</td>
<td>1</td>
<td>ALU</td>
<td>B</td>
<td>1</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>jalr</td>
<td>*</td>
<td>*</td>
<td>ALU</td>
<td>I</td>
<td>*</td>
<td>Reg</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>1</td>
<td>PC+4</td>
</tr>
<tr>
<td>jal</td>
<td>*</td>
<td>*</td>
<td>ALU</td>
<td>J</td>
<td>*</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>1</td>
<td>PC+4</td>
</tr>
<tr>
<td>auipc</td>
<td>*</td>
<td>*</td>
<td>+4</td>
<td>U</td>
<td>*</td>
<td>PC</td>
<td>Imm</td>
<td>Add</td>
<td>Read</td>
<td>1</td>
<td>ALU</td>
</tr>
</tbody>
</table>
Control Realization Options

- **ROM**
  - "Read-Only Memory"
  - Regular structure
  - Can be easily reprogrammed
    - fix errors
    - add instructions
  - Popular when designing control logic manually

- **Combinatorial Logic**
  - Today, chip designers use logic synthesis tools to convert truth tables to networks of gates
RV32I, A Nine-Bit ISA!

- Instruction type encoded using only 9 bits:
- inst[30], inst[14:12], inst[6:2]
ROM-based Control

Inputs:
- 11-bit address
- \text{Inst}[30,14:12,6:2]
- \text{BrEq}
- \text{BrLT}

Outputs:
- \text{PCSel}
- \text{ImmSel}[2:0]
- \text{BrUn}
- \text{ASel}
- \text{BSel}
- \text{ALUSEl}[3:0]
- \text{MemRW}
- \text{RegWEn}
- \text{WBSel}[1:0]

15 data bits (outputs)
ROM Controller Implementation

Address Decoder

- Inst[
- BrEQ
- BrLT

AND

- add
- sub
- or
- jal

OR

- Control Word for add
- Control Word for sub
- Control Word for or

Controller output (PCSel, ImmSel, ...)

11
Simplest example: \textbf{BrUn}

- How to decode whether BrUn is 1?
  \[
  \text{BrUn} = \text{Inst} \ [13] \ \bullet \ \text{Branch}
  \]
Control Logic to Decode add

\[
\text{add} = i_{30} \cdot i_{14} \cdot i_{13} \cdot i_{12} \cdot \text{R-type}
\]

\[
\text{inst}_{30} \quad \text{inst}_{14:12} \quad \text{inst}_{6:2}
\]

\[
\begin{array}{cccccc}
00000 & \text{shamt} & \text{rs1} & 001 & \text{rd} & 0010011 \\
00000 & \text{shamt} & \text{rs1} & 101 & \text{rd} & 0010011 \\
00000 & \text{shamt} & \text{rs1} & 101 & \text{rd} & 0010011 \\
00000 & \text{rs2} & \text{rs1} & 000 & \text{rd} & 0110011 \\
00000 & \text{rs2} & \text{rs1} & 000 & \text{rd} & 0110011 \\
00000 & \text{rs2} & \text{rs1} & 001 & \text{rd} & 0110011 \\
00000 & \text{rs2} & \text{rs1} & 010 & \text{rd} & 0110011 \\
00000 & \text{rs2} & \text{rs1} & 011 & \text{rd} & 0110011 \\
00000 & \text{rs2} & \text{rs1} & 100 & \text{rd} & 0110011 \\
00000 & \text{rs2} & \text{rs1} & 101 & \text{rd} & 0110011 \\
00000 & \text{rs2} & \text{rs1} & 110 & \text{rd} & 0110011 \\
00000 & \text{rs2} & \text{rs1} & 111 & \text{rd} & 0110011 \\
\end{array}
\]

R-type = i_{6} \cdot i_{5} \cdot i_{4} \cdot i_{3} \cdot i_{2} \cdot \text{RV32I}

RV32I = i_{1} \cdot i_{0}
“And In Conclusion...”
Call home, we’ve made HW/SW contact!

- 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)

Compiler
Assembler

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

Architecture Implementation

```
Out = A*B + C*D
```

```
IMEM
ALU
Imm.
Gen
+4
DMEM
Branch
Comp.
Reg[
][
]
AddrA
AddrB
DataA
AddrD
DataB
DataD
Addr
DataW
DataR
1
0
2
1
pc
0
1
inst[11:7]

addr

pc+4
alu
mem
wb
alu
pc+4

Reg[rs1]

Reg[rs2]

imm[31:0]

wb
“And In conclusion…”

- We have built a processor!
  - Capable of executing all RISC-V instructions in one cycle each
  - Not all units (hardware) used by all instructions
  - Critical path changes

- 5 Phases of execution
  - IF, ID, EX, MEM, WB
  - Not all instructions are active in all phases

- Controller specifies how to execute instructions
  - Implemented as ROM or logic