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

### CSR Instructions

<table>
<thead>
<tr>
<th>source/dest</th>
<th>source</th>
<th>instr</th>
<th>rd</th>
<th>SYSTEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>csr</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

### Register operand

<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>

<table>
<thead>
<tr>
<th>csr</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
<th>uimm[4:0]</th>
<th>1110011</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>1110011</td>
<td></td>
</tr>
</tbody>
</table>

### System Field

- **csrrw**: Read CSR (no) or write CSR (yes)
- **csrrw**: Read CSR (yes) or write CSR (yes)
- **csrrs/c**: Read CSR (yes) or write CSR (no)
- **csrrs/c**: Read CSR (yes) or write CSR (yes)
## 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>

### Register Access

- **csrrwi**: Read CSR (rd = 0)
- **csrrwi**: Write CSR (rd = 1)

### Immediate Operands

<table>
<thead>
<tr>
<th>source/dest</th>
<th>source</th>
<th>instr</th>
<th>rd</th>
<th>SYSTEM</th>
<th>uimm[4:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>csr</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
</tbody>
</table>

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 `csrw csr, rs1` is `csrrw x0, csr, rs1`

- \( rd=x0 \), just writes \( rs1 \) to CSR

Pseudoinstruction `csrwi csr, uimm` is `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
Our Single-Core Processor

Processor

Control

Datapath

Program Counter (PC)

Registers

Arithmetic-Logic Unit (ALU)

Memory

Enable?
Read/Write

Address

Write Data

Read Data

Program

Bytes

Data

Input

Output
Single-Cycle RV32I Datapath and Control
Example: \texttt{sw}

```plaintext
Example: \texttt{sw}

\begin{itemize}
\item \texttt{ALUSel}
\item \texttt{Add}
\item \texttt{clk}
\item \texttt{addr}
\item \texttt{inst}
\item \texttt{IMEM}
\item \texttt{addr}
\item \texttt{DMEM}
\item \texttt{PC}
\item \texttt{pc+4}
\item \texttt{Inst[24:20]}
\item \texttt{Inst[19:15]}
\item \texttt{Inst[11:7]}
\item \texttt{Inst[10:0]}
\item \texttt{Inst[31:20]}
\item \texttt{ImmSel[31:0]}
\item \texttt{RegWEn}
\item \texttt{MemRW}
\item \texttt{Branch}
\item \texttt{Comp}
\item \texttt{WBSel}
\item \texttt{Alu}
\item \texttt{Reg[rs1]}
\item \texttt{Reg[rs2]}
\item \texttt{DataA}
\item \texttt{DataB}
\item \texttt{DataD}
\item \texttt{AddrB}
\item \texttt{AddrA}
\item \texttt{AddrD}
\item \texttt{Imm[31:0]}
\item \texttt{ImmSel}
\item \texttt{BrUn}
\item \texttt{BrEq}
\item \texttt{BrLT}
\end{itemize}

\begin{itemize}
\item \texttt{ PCSel = pc+4 }
\item \texttt{Inst[31:0]}
\item \texttt{ImmSel = S}
\item \texttt{RegWEn = 0}
\item \texttt{BrUn = *}
\item \texttt{BrEq = *}
\item \texttt{BrLT = *}
\item \texttt{Asel = 0}
\item \texttt{ALUSel = add}
\item \texttt{MemRW = Write}
\item \texttt{WBSel = *}
\end{itemize}

\text{}
Example: `beq`
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]
Reg[7]
Reg[9]

Garcia, Nikolić
**Example:** add timing

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

= $t_{\text{clk-q}} + t_{\text{IMEM}} + t_{\text{Reg}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{mux}} + t_{\text{setup}}$
Example: 1w

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

RISC-V (93)
### Instruction Timing

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-MEM</td>
<td>Reg Read</td>
<td>ALU</td>
<td>D-MEM</td>
<td>Reg W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td>100 ps</td>
<td>800 ps</td>
<td></td>
</tr>
</tbody>
</table>
### 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>X</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>X</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 Design
## 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!

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LUI</td>
<td>ADDI</td>
<td>SLL</td>
</tr>
<tr>
<td>AUipc</td>
<td>SRLI</td>
<td>SLT</td>
</tr>
<tr>
<td>JAL</td>
<td>ADD</td>
<td>SRAI</td>
</tr>
<tr>
<td>JALR</td>
<td>SUB</td>
<td>SH</td>
</tr>
<tr>
<td>BEQ</td>
<td>SRL</td>
<td>XOR</td>
</tr>
<tr>
<td>BLT</td>
<td>SLTIU</td>
<td>ORI</td>
</tr>
<tr>
<td>BGE</td>
<td>ORI</td>
<td>AND</td>
</tr>
<tr>
<td>BLTU</td>
<td>ANDI</td>
<td>JR</td>
</tr>
<tr>
<td>BGEU</td>
<td>JR</td>
<td>BE</td>
</tr>
<tr>
<td>LB</td>
<td>BE</td>
<td>ECALL</td>
</tr>
<tr>
<td>LH</td>
<td>ECALL</td>
<td>EBREAK</td>
</tr>
<tr>
<td>LW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LHU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SH</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

11-bit address (inputs)

Inst[30,14:12,6:2]  BrEq  BrLT

9

PCSel
ImmSel[2:0]
BrUn
ASel
BSel
ALUSel[3:0]
MemRW
RegWEn
WBSel[1:0]

15 data bits (outputs)
ROM Controller Implementation

AND

Inst[], BrEQ, BrLT

add, sub, or

OR

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

Controller output (PCSel, ImmSel, ...)

Address Decoder
Combinational Logic Control

- Simplest example: **BrUn**

How to decode whether BrUn is 1?

\[ \text{BrUn} = \text{Inst}[13] \cdot \text{Branch} \]
Control Logic to Decode \texttt{add}

\begin{align*}
\text{add} & = \text{i}[30] \cdot \text{i}[14] \cdot \text{i}[13] \cdot \text{i}[12] \cdot \text{R-type} \\
\text{R-type} & = \text{i}[6] \cdot \text{i}[5] \cdot \text{i}[4] \cdot \text{i}[3] \cdot \text{i}[2] \cdot \text{RV32I} \\
\text{RV32I} & = \text{i}[1] \cdot \text{i}[0]
\end{align*}
“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)
- Compiler
- Assembler
- Hardware Architecture Description (e.g., block diagrams)
- Architecture Implementation
- 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
```

---

Garcia, Nikolić

RISC-V (105)
“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