Control and Status Registers
Complete Single-Cycle RV32I Datapath!

Garcia, Nikolić

RISC-V (78)
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

<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>csr</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

### Register Operand Table

<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>

---

This document is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. You may distribute, remix, transform, and build upon this document provided that you give appropriate credit, do not use the material for commercial purposes, and distribute any adaptations under the same license. For more information, see [Creative Commons](https://creativecommons.org/licenses/by-nc-sa/4.0/).
## CSR Instructions

### CSR Instructions Table

<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>

### CSR Instruction Format

```
31  20  19  15  14  12  11  7  6  5  4  3  2  1  0
   csr  rs1  funct3  rd  opcode
```

- **source/dest**: source, instr, rd, SYSTEM
- **uimm [4:0]**: 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 \( \text{rd} \). Then writes \( \text{rs1} \) to CSR

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

- Pseudoinstruction \textit{csrwi csr, uimm} is \textit{csrrwi x0, csr, uimm}
  \- \( \text{rd}=\text{x0} \), just writes \( \text{uimm} \) 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
  - Data
  - Bytes

Input

Read/Write

Enable?

Address

Write Data

Read Data

Output
Single-Cycle RV32I Datapath and Control
Example: sw

```
ALUSel +4 Add
IMEM addr pc+4 Inst[24:20] ALU
Reg[ ] Inst[19:15] Imm[31:0]
AddrB AddrA DataA DataD
Imm[31:0] R[rs1] R[rs2] ImmSel
PC Sel= pc+4 Control logic
RegWEn =0 Br.Un =* Br.LT =*
Asel =0 ALUSel =add MemRW =Write
WBSel =*

Garcia, Nikolić
```
Example: `beq`

```
beq ALUSel +4
Add

addr
inst
IMEM
addr
DMEM
PC
pc+4
Inst [24:20]
Inst [19:15]
Inst [11:7]
Inst [24:20]

alu
R[rs1]
R[rs2]

Reg

Inst [31:0]

Control logic

RegWEn

ImmSel =B

Imm[31:0]

ImmSel

Inst[31:0]

Inst

IMEM

AddrB

AddrA

AddrD

DataA

DataD

Branch Comp

R[rs1]

R[rs2]

MemRW

clk

0

1

0

1

0

1

0

1

0

1

0

1

=*

=*

=*

=*

BrUn

BrLT

Bsel =1

Aset =1

ALUSel =add

MemRW =Read

WBSel =*

alu

DataB

DMEM
```
Instruction Timing
Example: add

![Diagram of RISC-V instruction execution]
Add Execution

Clock
PC
PC+4
inst[31:0]
Control logic
Reg[rs1]
Reg[rs2]
alu
wb
Reg[11]

1000
1004
1008
add x1,x2,x3
add x6,x7,x9
add control
Reg[2]
Reg[3]

Garcia, Nikolić

RISC-V (91)
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: lw

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

<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></td>
<td>I-MEM</td>
<td>Reg Read</td>
<td>ALU</td>
<td>D-MEM</td>
<td>Reg W</td>
<td>800 ps</td>
</tr>
<tr>
<td></td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td>100 ps</td>
<td></td>
</tr>
</tbody>
</table>
### Maximum clock frequency
- \( f_{\text{max}} = \frac{1}{800\text{ps}} = 1.25 \text{ GHz} \)

### Most blocks idle most of the time
- E.g. \( f_{\text{max,ALU}} = \frac{1}{200\text{ps}} = 5 \text{ GHz}! \)
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>RegWE</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]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LUI</td>
<td>ADD</td>
<td>JAL</td>
</tr>
<tr>
<td>AUIPC</td>
<td>ADDI</td>
<td>JALR</td>
</tr>
<tr>
<td>LUI</td>
<td>SLTI</td>
<td>SJAL</td>
</tr>
<tr>
<td>JAIPC</td>
<td>SLTIU</td>
<td>SHL</td>
</tr>
<tr>
<td>BNE</td>
<td>XORI</td>
<td>SRAI</td>
</tr>
<tr>
<td>BGE</td>
<td>ORI</td>
<td>SRM</td>
</tr>
<tr>
<td>BLT</td>
<td>ANDI</td>
<td>SSLR</td>
</tr>
<tr>
<td>BLE</td>
<td>SLTU</td>
<td>SRSL</td>
</tr>
<tr>
<td>BEQ</td>
<td>SRLI</td>
<td>SRL</td>
</tr>
<tr>
<td>LBU</td>
<td>AND</td>
<td>SHL</td>
</tr>
<tr>
<td>LB</td>
<td>OR</td>
<td>SH</td>
</tr>
<tr>
<td>LH</td>
<td>ANDI</td>
<td>SH</td>
</tr>
<tr>
<td>LW</td>
<td>ADDI</td>
<td>SH</td>
</tr>
<tr>
<td>LBU</td>
<td>ADD</td>
<td>SH</td>
</tr>
<tr>
<td>LHU</td>
<td>SLTI</td>
<td>SH</td>
</tr>
<tr>
<td>SB</td>
<td>ADDI</td>
<td>INDEX</td>
</tr>
<tr>
<td>SB</td>
<td>ADDI</td>
<td>INDEX</td>
</tr>
<tr>
<td>SH</td>
<td>ADD</td>
<td>INDEX</td>
</tr>
<tr>
<td>SW</td>
<td>ADD</td>
<td>INDEX</td>
</tr>
<tr>
<td>SB</td>
<td>ADD</td>
<td>INDEX</td>
</tr>
<tr>
<td>SH</td>
<td>ADD</td>
<td>INDEX</td>
</tr>
<tr>
<td>SW</td>
<td>ADD</td>
<td>INDEX</td>
</tr>
</tbody>
</table>

Garcia, Nikolić

RV32I: A Nine-Bit ISA!
ROM-based Control

11-bit address (inputs)

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

9

ROM

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
11

Address Decoder

add
sub
or
jal

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

Controller output (PCSel, ImmSel, ...)

OR

Garcia, Nikolić

RISC-V (101)
Combinational Logic Control

- Simplest example: BrUn

\[
\begin{array}{c|c|c|c|c|c}
\text{imm}[12:0.5] & \text{rs2} & \text{rs1} & 000 & \text{imm}[4:11|1] & 110001 \\
\hline
\text{imm}[12:0.5] & \text{rs2} & \text{rs1} & 001 & \text{imm}[4:11|1] & 110011 \\
\hline
\text{imm}[12:0.5] & \text{rs2} & \text{rs1} & 100 & \text{imm}[4:11|1] & 110011 \\
\hline
\text{imm}[12:0.5] & \text{rs2} & \text{rs1} & 101 & \text{imm}[4:11|1] & 110011 \\
\hline
\text{imm}[12:0.5] & \text{rs2} & \text{rs1} & 110 & \text{imm}[4:11|1] & 110011 \\
\hline
\text{imm}[12:0.5] & \text{rs2} & \text{rs1} & 111 & \text{imm}[4:11|1] & 110011 \\
\end{array}
\]

How to decode whether BrUn is 1?

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

\[
\text{add} = \text{inst}[30] \cdot \text{inst}[14:12] \cdot \text{inst}[6:2] \\
\]

R-type = \text{inst}[6] \cdot \text{inst}[5] \cdot \text{inst}[4] \cdot \text{inst}[3] \cdot \text{inst}[2] \cdot \text{RV32I}

RV32I = \text{inst}[1] \cdot \text{inst}[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)
- Architecture Implementation
- Logic Circuit Description (Circuit Schematic Diagrams)

Compiler

Assembler

Machine Language Program (RISC-V)

Assembly Language Program (e.g., RISC-V)

High Level Language Program (e.g., C)

Machine Language Program (RISC-V)

High Level Language Program (e.g., C)

Assembly Language Program (e.g., RISC-V)

High Level Language Program (e.g., C)
“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
Great Ideas in Computer Architecture (a.k.a. Machine Structures)

Pipelining
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
- Logic Gates
- Core
- Memory
  (Cache)
- Input/Output
- Computer
  Warehouse Scale
  Computer
- Main Memory
- Exec. Unit(s)
  Functional Block(s)
- A0+B0
  A1+B1
- Out = AB + CD

Harness Parallelism & Achieve High Performance

Berkeley
UNIVERSITY OF CALIFORNIA

Garcia, Nikolić
6 Great Ideas in Computer Architecture

1. Abstraction (Layers of Representation/Interpretation)
2. Moore’s Law
3. Principle of Locality/Memory Hierarchy
4. Parallelism
5. Performance Measurement & Improvement
6. Dependability via Redundancy
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></td>
<td>I-MEM</td>
<td>Reg Read</td>
<td>ALU</td>
<td>D-MEM</td>
<td>Reg W</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>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>jal</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>lw</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

- Maximum clock frequency
  - $f_{\text{max}} = 1/800\text{ps} = 1.25\text{ GHz}$
“Our” Single-cycle RISC-V CPU executes instructions at 1.25 GHz
- 1 instruction every 800 ps

Can we improve its performance?
- What do we mean with this statement?
- Not so obvious:
  - Quicker response time, so one job finishes faster?
  - More jobs per unit time (e.g. web server returning pages, spoken words recognized)?
  - Longer battery life?
# Transportation Analogy

## Sports Car vs. Bus

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Sports Car</th>
<th>Bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passenger Capacity</td>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>Travel Speed</td>
<td>200 mph</td>
<td>50 mph</td>
</tr>
<tr>
<td>Gas Mileage</td>
<td>5 mpg</td>
<td>2 mpg</td>
</tr>
</tbody>
</table>

## 50 Mile trip (assume they return instantaneously)

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Sports Car</th>
<th>Bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Travel Time</td>
<td>15 min</td>
<td>60 min</td>
</tr>
<tr>
<td>Time for 100 passengers</td>
<td>750 min (50 2-person trips)</td>
<td>120 min (two 50-person trips)</td>
</tr>
<tr>
<td>Gallons per passenger</td>
<td>5 gallons</td>
<td>0.5 gallons</td>
</tr>
</tbody>
</table>
## Computer Analogy

<table>
<thead>
<tr>
<th>Transportation</th>
<th>Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trip Time</td>
<td>Program execution time: e.g. time to update display</td>
</tr>
<tr>
<td>Time for 100 passengers</td>
<td>Throughput: e.g. number of server requests handled per hour</td>
</tr>
<tr>
<td>Gallons per passenger</td>
<td>Energy per task*: e.g. how many movies you can watch per battery charge or energy bill for datacenter</td>
</tr>
</tbody>
</table>

* **Note**: Power is not a good measure, since low-power CPU might run for a long time to complete one task consuming more energy than faster computer running at higher power for a shorter time.
Processor Performance Iron Law
“Iron Law” of Processor Performance

\[
\frac{\text{Time}}{\text{Program}} = \left(\frac{\text{Instructions}}{\text{Program}}\right) \cdot \left(\frac{\text{Cycles}}{\text{Instruction}}\right) \cdot \left(\frac{\text{Time}}{\text{Cycle}}\right)
\]

\[\text{CPI} = \frac{\text{Cycles}}{\text{Per Instruction}}\]
Instructions per Program

Determined by

- Task
- Algorithm, e.g. $O(N^2) \text{ vs } O(N)$
- Programming language
- Compiler
- Instruction Set Architecture (ISA)
(Average) Clock Cycles per Instruction (CPI)

Determined by

- ISA
- Processor implementation (or microarchitecture)
- E.g. for “our” single-cycle RISC-V design, CPI = 1
- Complex instructions (e.g. `strcpy`), CPI >> 1
- Superscalar processors, CPI < 1 (next lectures)
Determined by

- Processor microarchitecture (determines critical path through logic gates)
- Technology (e.g. 5nm versus 28nm)
- Power budget (lower voltages reduce transistor speed)
Speed Tradeoff Example

- For some task (e.g. image compression) ...

<table>
<thead>
<tr>
<th></th>
<th>Processor A</th>
<th>Processor B</th>
</tr>
</thead>
<tbody>
<tr>
<td># Instructions</td>
<td>1 Million</td>
<td>1.5 Million</td>
</tr>
<tr>
<td>Average CPI</td>
<td>2.5</td>
<td>1</td>
</tr>
<tr>
<td>Clock rate $f$</td>
<td>2.5 GHz</td>
<td>2 GHz</td>
</tr>
<tr>
<td>Execution time</td>
<td>1 ms</td>
<td>0.75 ms</td>
</tr>
</tbody>
</table>

Processor B is faster for this task, despite executing more instructions and having a slower clock rate!
Energy Efficiency
Where Does Energy Go in CMOS?

Symbol (INV)

\[ V_{DD} \]

A \quad \text{Out}

Schematic

A \quad \text{Out}

- Leakage: (30%)
- Charging capacitors \((CV^2)\): (70%)
Energy per Task

\[ \text{Energy} = \text{Instructions}_{\text{Program}} \times \text{Energy}_{\text{Instruction}} \]

\[ \text{Energy} \propto \text{Instructions}_{\text{Program}} \times C \times V^2 \]

“Capacitance” depends on technology, processor features e.g. # of cores

Supply voltage, e.g. 1V

Want to reduce capacitance and voltage to reduce energy/task
Energy Tradeoff Example

- “Next-generation” processor
  - C (Moore’s Law): -15%
  - Supply voltage, $V_{sup}$: -15%
  - Energy consumption: $0 - (1 - 0.85^3) = -39\%$

- Significantly improved energy efficiency thanks to
  - Moore’s Law AND
  - Reduced supply voltage
Performance/Power Trends

48 Years of Microprocessor Trend Data

Transistors (thousands)
Single-Thread Performance (SpecINT x 10^3)
Frequency (MHz)
Typical Power (Watts)
Number of Logical Cores

Year


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten.
New plot and data collected for 2010-2019 by K. Rupp.

Garcia, Nikolić

Berkeley UNIVERSITY OF CALIFORNIA

RISC-V (19)
In recent years, industry has not been able to reduce supply voltage much, as reducing it further would mean increasing “leakage power” where transistor switches don’t fully turn off (more like dimmer switch than on-off switch).

Also, size of transistors and hence capacitance, not shrinking as much as before between transistor generations. Need to go to 3D.

Power becomes a growing concern – the “power wall”
Energy efficiency (e.g., instructions/Joule) is key metric in all computing devices.

For power-constrained systems (e.g., 20MW datacenter), need better energy efficiency to get more performance at same power.

For energy-constrained systems (e.g., 1W phone), need better energy efficiency to prolong battery life.

\[
\text{Performance} = \frac{\text{Power}}{\text{Energy Efficiency}} = \frac{(\text{Tasks/Second})}{(\text{Joules/Second})} = (\text{Tasks/Joule})
\]
Introduction to Pipelining
Gotta Do Laundry

- Avi, Bora, Caroline, Dan each have one load of clothes to wash, dry, fold, and put away
  - Washer takes 30 minutes
  - Dryer takes 30 minutes
  - “Folder” takes 30 minutes
  - “Stasher” takes 30 minutes to put clothes into drawers
Sequential laundry takes 8 hours for 4 loads!
Pipelined Laundry

*What happens sequentially?*

*What happens simultaneously?*

Pipelined laundry takes 3.5 hours for 4 loads!
- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number of pipe stages
- Time to “fill” pipeline and time to “drain” it reduces speedup: 2.3X v. 4X in this example
Suppose:
- new Washer takes 20 minutes
- new Stasher takes 20 minutes.

How much faster is pipeline?

Pipeline rate limited by slowest pipeline stage

Unbalanced lengths of pipe stages reduce speedup