Single-Cycle Processors: 
Datapath & Control

Arvind
Computer Science & Artificial Intelligence Lab
M.I.T.

Based on the material prepared by
Arvind and Krste Asanovic
Instruction Set Architecture (ISA) versus Implementation

- ISA is the hardware/software interface
  - Defines set of programmer visible state
  - Defines instruction format (bit encoding) and instruction semantics
  - Examples: MIPS, x86, IBM 360, JVM

- Many possible implementations of one ISA
  - 360 implementations: model 30 (c. 1964), z900 (c. 2001)
  - x86 implementations: 8086 (c. 1978), 80186, 286, 386, 486, Pentium, Pentium Pro, Pentium-4 (c. 2000), AMD Athlon, Transmeta Crusoe, SoftPC
  - MIPS implementations: R2000, R4000, R10000, ...
  - JVM: HotSpot, PicoJava, ARM Jazelle, ...

September 26, 2005
Processor Performance

\[
\text{Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}
\]

- Instructions per program depends on source code, compiler technology, and ISA.
- Cycles per instructions (CPI) depends upon the ISA and the microarchitecture.
- Time per cycle depends upon the microarchitecture and the base technology.

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>CPI</th>
<th>cycle time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcoded</td>
<td>&gt;1</td>
<td>short</td>
</tr>
<tr>
<td>Single-cycle unpipelined</td>
<td>1</td>
<td>long</td>
</tr>
<tr>
<td>Pipelined</td>
<td>1</td>
<td>short</td>
</tr>
</tbody>
</table>
Microarchitecture: Implementation of an ISA

**Structure:** How components are connected. **Static**

**Behavior:** How data moves between components. **Dynamic**
Hardware Elements

- Combinational circuits
  - Mux, Demux, Decoder, ALU, ...

- Synchronous state elements
  - Flipflop, Register, Register file, SRAM, DRAM

Edge-triggered: Data is sampled at the rising edge
Register Files

- No timing issues in reading a selected register
- Register files with a large number of ports are difficult to design
  - Intel’s Itanium, GPR File has 128 registers with 8 read ports and 4 write ports!!!
A Simple Memory Model

Reads and writes are always completed in one cycle
- a Read can be done any time (i.e. combinational)
- a Write is performed at the rising clock edge if it is enabled
  \[ \Rightarrow \text{the write address and data must be stable at the clock edge} \]

Later in the course we will present a more realistic model of memory
Implementing MIPS:
Single-cycle per instruction datapath & control logic
The MIPS ISA

Processor State
32 32-bit GPRs, R0 always contains a 0
32 single precision FPRs, may also be viewed as
   16 double precision FPRs
FP status register, used for FP compares & exceptions
PC, the program counter
some other special registers

Data types
8-bit byte, 16-bit half word
32-bit word for integers
32-bit word for single precision floating point
64-bit word for double precision floating point

Load/Store style instruction set
data addressing modes- immediate & indexed
branch addressing modes- PC relative & register indirect
Byte addressable memory- big endian mode

All instructions are 32 bits
Instruction Execution

Execution of an instruction involves

1. instruction fetch
2. decode and register fetch
3. ALU operation
4. memory operation (optional)
5. write back

and the computation of the address of the next instruction
Datapath: Reg-Reg ALU Instructions

RegWrite Timing?

rd ← (rs) func (rt)

September 26, 2005
Datapath: Reg-Imm ALU Instructions

OpCode

PC

0x4

Add

Addr

Inst.

Memory

inst<25:21>

inst<20:16>

inst<15:0>

inst<31:26>

0x4

Add

RegWrite

clk

we

rs1

rs2

rd1

ws

wd

rd2

GPRs

ALU

z

ALU

Control

ExtSel

rt ← (rs) op immediate

September 26, 2005
Conflicts in Merging Datapath

Introduce muxes

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>func</td>
</tr>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Datapath for ALU Instructions

 Opcode | rs | rt | rd | immediate |
---------|----|----|----|-----------|
 0       |    |    |    |           |

rd ← (rs) func (rt)
rt ← (rs) op immediate

September 26, 2005
Datapath for Memory Instructions

Should program and data memory be separate?

*Harvard style: separate* (Aiken and Mark 1 influence)
- read-only program memory
- read/write data memory
  at some level the two memories have
to be the same

*Princeton style: the same* (von Neumann’s influence)
- A Load or Store instruction requires
  accessing the memory more than once
during its execution

September 26, 2005
Load/Store Instructions: Harvard Datapath

rs is the base register
rt is the destination of a Load or the source for a Store

addressing mode (rs) + displacement

September 26, 2005
# MIPS Control Instructions

## Conditional (on GPR) PC-relative branch

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td></td>
<td>offset</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQZ, BNEZ</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Unconditional register-indirect jumps

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JR, JALR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Unconditional absolute jumps

<table>
<thead>
<tr>
<th>6</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>target</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>J, JAL</td>
<td></td>
</tr>
</tbody>
</table>

- PC-relative branches add offset×4 to PC+4 to calculate the target address (offset is in words): ±128 KB range
- Absolute jumps append target×4 to PC<31:28> to calculate the target address: 256 MB range
- Jump-&-link stores PC+4 into the link register (R31)
- All Control Transfers are delayed by 1 instruction

*we will worry about the branch delay slot later*

September 26, 2005
Conditional Branches (BEQZ, BNEZ)
Register-Indirect Jumps (JR)
Register-Indirect Jump-&-Link (JALR)

6.823 L5-20
Arvind

September 26, 2005
Absolute Jumps (J, JAL)
Harvard-Style Datapath for MIPS
Five-minute break to stretch your legs
Single-Cycle Hardwired Control: Harvard architecture

We will assume

- clock period is sufficiently long for all of the following steps to be “completed”:

1. instruction fetch
2. decode and register fetch
3. ALU operation
4. data fetch if required
5. register write-back setup time

⇒ \[ t_C > t_{IFetch} + t_{RFetch} + t_{ALU} + t_{DMem} + t_{RWB} \]

- At the rising edge of the following clock, the PC, the register file and the memory are updated
Hardwired Control is pure Combinational Logic

- op code
- zero?

combinational logic

- ExtSel
- BSrc
- OpSel
- MemWrite
- WBSrc
- RegDst
- RegWrite
- PCSrc

September 26, 2005
ALU Control & Immediate Extension

Inst<5:0> (Func)
Inst<31:26> (Opcode)

ALUop

+ 0?

Decode Map

OpSel (Func, Op, +, 0?)

ExtSel (sExt_{16}, uExt_{16}, High_{16})
# Hardwired Control Table

<table>
<thead>
<tr>
<th>Opcode</th>
<th>ExtSel</th>
<th>BSrc</th>
<th>OpSel</th>
<th>MemW</th>
<th>RegW</th>
<th>WBSrc</th>
<th>RegDst</th>
<th>PCSrc</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>*</td>
<td>Reg</td>
<td>Func</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rd</td>
<td>pc+4</td>
</tr>
<tr>
<td>ALUi</td>
<td>sExt_{16}</td>
<td>Imm</td>
<td>Op</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rt</td>
<td>pc+4</td>
</tr>
<tr>
<td>ALUiu</td>
<td>uExt_{16}</td>
<td>Imm</td>
<td>Op</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rt</td>
<td>pc+4</td>
</tr>
<tr>
<td>LW</td>
<td>sExt_{16}</td>
<td>Imm</td>
<td>+</td>
<td>no</td>
<td>yes</td>
<td>Mem</td>
<td>rt</td>
<td>pc+4</td>
</tr>
<tr>
<td>SW</td>
<td>sExt_{16}</td>
<td>Imm</td>
<td>+</td>
<td>yes</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>pc+4</td>
</tr>
<tr>
<td>BEQZ_{z=0}</td>
<td>sExt_{16}</td>
<td>*</td>
<td>0?</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>br</td>
</tr>
<tr>
<td>BEQZ_{z=1}</td>
<td>sExt_{16}</td>
<td>*</td>
<td>0?</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>pc+4</td>
</tr>
<tr>
<td>J</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>jabs</td>
</tr>
<tr>
<td>JAL</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>yes</td>
<td>PC</td>
<td>R31</td>
<td>jabs</td>
</tr>
<tr>
<td>JR</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>rind</td>
</tr>
<tr>
<td>JALR</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>yes</td>
<td>PC</td>
<td>R31</td>
<td>rind</td>
</tr>
</tbody>
</table>

BSrc = Reg / Imm
RegDst = rt / rd / R31
WBSrc = ALU / Mem / PC
PCSrc = pc+4 / br / rind / jabs

September 26, 2005
Pipelined MIPS

To pipeline MIPS:

• First build MIPS without pipelining with CPI=1

• Next, add pipeline registers to reduce cycle time while maintaining CPI=1
Pipelined Datapath

Clock period can be reduced by dividing the execution of an instruction into multiple cycles

\[ t_C > \max\{t_{IM}, t_{RF}, t_{ALU}, t_{DM}, t_{RW}\} \ (= t_{DM} \text{ probably}) \]

However, CPI will increase unless instructions are pipelined
An Ideal Pipeline

- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- The scheduling of an object entering the pipeline is not affected by the objects in other stages

*These conditions generally hold for industrial assembly lines.*

*But can an instruction pipeline satisfy the last condition?*

September 26, 2005
How to divide the datapath into stages

Suppose memory is significantly slower than other stages. In particular, suppose

\[ t_{IM} = 10 \text{ units} \]
\[ t_{DM} = 10 \text{ units} \]
\[ t_{ALU} = 5 \text{ units} \]
\[ t_{RF} = 1 \text{ unit} \]
\[ t_{RW} = 1 \text{ unit} \]

Since the slowest stage determines the clock, it may be possible to combine some stages without any loss of performance
Alternative Pipelining

\[
t_C > \max \{t_{IM}, t_{RF} + t_{ALU}, t_{DM} + t_{RW}\} = t_{DM} + t_{RW}
\]

⇒ increase the critical path by 10%

Write-back stage takes much less time than other stages. Suppose we combined it with the memory phase.
## Maximum Speedup by Pipelining

<table>
<thead>
<tr>
<th>Assumptions</th>
<th>Unpipelined Speedup</th>
<th>Pipelined Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $t_{IM} = t_{DM} = 10$, $t_{ALU} = 5$, $t_{RF} = t_{RW} = 1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-stage pipeline</td>
<td>27 10 2.7</td>
<td></td>
</tr>
<tr>
<td>2. $t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-stage pipeline</td>
<td>25 10 2.5</td>
<td></td>
</tr>
<tr>
<td>3. $t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5-stage pipeline</td>
<td>25 5 5.0</td>
<td></td>
</tr>
</tbody>
</table>

It is possible to achieve higher speedup with more stages in the pipeline.
Thank you!