Simple Instruction Pipelining

Krste Asanovic
Laboratory for Computer Science
Massachusetts Institute of Technology
Processor Performance Equation

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

- Instructions per program depends on source code, compiler technology, and ISA
- Microcoded DLX from last lecture had cycles per instruction (CPI) of around 7 minimum
- Time per cycle for microcoded DLX fixed by microcode cycle time
  - mostly ROM access + next \(\mu\)PC select logic
Pipelined DLX

To pipeline DLX:

- First build unpipelined DLX with CPI=1

- Next, add pipeline registers to reduce cycle time while maintaining CPI=1
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

⇒ *the write address, data, and enable must be stable at the clock edge*
Datapath for ALU Instructions

<table>
<thead>
<tr>
<th>opcode</th>
<th>rf1</th>
<th>rf2</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rf3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>func</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

rf3 ← (rf1) func (rf2)
rf2 ← (rf1) op immediate
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
Load/Store Instructions: Harvard-Style Datapath

- RegWrite
- MemWrite
- WBSrc
- ALU / Mem

**Addressing Mode:**
- \((rf1) + \text{displacement}\)

- \(rf1\) is the base register
- \(rf2\) is the destination of a Load or the source for a Store
Our memory model is a good approximation of the hierarchical memory system when we hit in the on-chip cache.
Conditional Branches

PCSrc ( ~j / j )

RegWrite

MemWrite

WBSrc

0x4

Add

addr

inst

Inst. Memory

clk

Pc

clk

alu

control

ALU

OPCode RegDst ExtSel OpSel BSrc zero?

add

rs1 rs2

we ws wd rd2 GPRs

imm Ext

alu

wdata rdata Data Memory wdata

clk
Register-Indirect Jumps

PCSrc ( ~j / j RInd / j PCR ) → RegWrite → MemWrite → WBSrc

0x4 Add → Addr
Addr → Inst. Memory → clk
Inst. Memory → Addr

Inst. Memory → Opcode RegDst

RegWrite → MemWrite
MemWrite → WBSrc

MemWrite → Data Memory wdata → ALU Control
ALU Control → ALU

ALU z → Zero?
Zero? → Jump & Link?

Jump & Link?
PC-Relative Jumps

No new datapath required
Hardwired Control is pure Combinational Logic: 

*Unpipelined DLX*
ALU Control & Immediate Extension

\[ \text{Inst}_{5:0} (\text{Func}) \]
\[ \text{Inst}_{31:26} (\text{Opcode}) \]

\[ \text{ALUop} \]

\[ \text{OpSel} (\text{Func, Op, +, 0?}) \]

\[ \text{ExtSel} (s\text{Ext}_{16}, u\text{Ext}_{16}, s\text{Ext}_{26}, \text{High}_{16}) \]

\[ + \]
\[ 0? \]

Decode Map
Hardwired Control worksheet.

Diagram showing the control flow of a hardwired control processor. The diagram includes blocks for PCSrc, RegWrite, MemWrite, WBSrc, ALU, Mem, adders, and control logic for selecting registers, operands, and memory addresses. The diagram illustrates the flow of signals between the different components, including control signals for register destinations, extenders,选射, and functional units, as well as the interaction with memory and ALU operations.
## Hardwired Control Table

BSrc = Reg / Imm  
WBSrc = ALU / Mem / PC  
PCSrc1 = j / ~j  
PCSrc2 = PCR / RInd  
RegDst = rf2 / rf3 / R31

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Ext Sel</th>
<th>BSrc</th>
<th>Op Sel</th>
<th>Mem Write</th>
<th>Reg Write</th>
<th>WB Src</th>
<th>Reg Dest</th>
<th>PC Src</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>rf3</td>
<td>~j</td>
</tr>
<tr>
<td>ALUu</td>
<td>*</td>
<td>Reg</td>
<td>Func</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rf3</td>
<td>~j</td>
</tr>
<tr>
<td>ALUi</td>
<td>sExt\textsubscript{16}</td>
<td>Imm</td>
<td>Op</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rf2</td>
<td>~j</td>
</tr>
<tr>
<td>ALUiu</td>
<td>uExt\textsubscript{16}</td>
<td>Imm</td>
<td>Op</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rf2</td>
<td>~j</td>
</tr>
<tr>
<td>LW</td>
<td>sExt\textsubscript{16}</td>
<td>Imm</td>
<td>+</td>
<td>no</td>
<td>yes</td>
<td>Mem</td>
<td>rf2</td>
<td>~j</td>
</tr>
<tr>
<td>SW</td>
<td>sExt\textsubscript{16}</td>
<td>Imm</td>
<td>+</td>
<td>yes</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>~j</td>
</tr>
<tr>
<td>BEQZ\textsubscript{z=0}</td>
<td>sExt\textsubscript{16}</td>
<td>*</td>
<td>0?</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>PCR</td>
</tr>
<tr>
<td>BEQZ\textsubscript{z=1}</td>
<td>sExt\textsubscript{16}</td>
<td>*</td>
<td>0?</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>~j</td>
</tr>
<tr>
<td>J</td>
<td>sExt\textsubscript{26}</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>PCR</td>
</tr>
<tr>
<td>JAL</td>
<td>sExt\textsubscript{26}</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>yes</td>
<td>PC</td>
<td>R31</td>
<td>PCR</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>
Hardwired Unpipelined Machine

- Simple
- One instruction per cycle
- Why wasn’t this a popular machine style?
Unpipelined DLX

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

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

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. An instruction pipeline, however, cannot satisfy the last condition. Why?
Pipelining History

- Some very early machines had limited pipelined execution (e.g., Zuse Z4, WWII)
  - Usually overlap fetch of next instruction with current execution

- IBM Stretch first major “supercomputer” incorporating extensive pipelining, result bypassing, and branch prediction
  - Project started in 1954, delivered in 1961
  - Didn’t meet initial performance goal of 100x faster with 10x faster circuits
  - Up to 11 macroinstructions in pipeline at same time
  - Microcode engine highly pipelined also (up to 6 microinstructions in pipeline at same time)
  - Stretch was origin of 8-bit byte and lower case characters, carried on into IBM 360
How to divide the datapath into stages

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

\[
\begin{align*}
    t_{IM} &= t_{DM} = 10 \text{ units} \\
    t_{ALU} &= 5 \text{ units} \\
    t_{RF} &= t_{RW} = 1 \text{ unit}
\end{align*}
\]

Since the slowest stage determines the clock, it may be possible to combine some stages without any loss of performance.
Minimizing Critical Path

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

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

⇒ increase the critical path by 10%
Maximum Speedup by Pipelining

For the 4-stage pipeline, given
\[ t_{IM} = t_{DM} = 10 \text{ units}, \quad t_{ALU} = 5 \text{ units}, \quad t_{RF} = t_{RW} = 1 \text{ unit} \]
\( t_C \) could be reduced from 27 units to 10 units
\[ \Rightarrow \text{speedup} = 2.7 \]

However, if \( t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5 \text{ units} \)
The same 4-stage pipeline can reduce \( t_C \) from 25 units to 10 units
\[ \Rightarrow \text{speedup} = 2.5 \]

But, since \( t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} \), it is possible to achieve higher speedup with more stages in the pipeline.
A 5-stage pipeline can reduce \( t_C \) from 25 units to 5 units
\[ \Rightarrow \text{speedup} = 5 \]
Technology Assumptions

We will assume

- A small amount of very fast memory (caches) backed up by a large, slower memory
- Fast ALU (at least for integers)
- Multiported Register files (slower!).

It makes the following timing assumption valid

\[ t_{IM} \approx t_{RF} \approx t_{ALU} \approx t_{DM} \approx t_{RW} \]

A 5-stage pipelined Harvard-style architecture will be the focus of our detailed design
5-Stage Pipelined Execution

fetch phase (IF)

decode & Reg-fetch phase (ID)

execute phase (EX)

memory phase (MA)

write-back phase (WB)

<table>
<thead>
<tr>
<th>time</th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
<th>....</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF_1</td>
<td>ID_1</td>
<td>EX_1</td>
<td>MA_1</td>
<td>WB_1</td>
<td>IF_2</td>
<td>ID_2</td>
<td>EX_2</td>
<td>MA_2</td>
<td>WB_2</td>
</tr>
<tr>
<td>IF_3</td>
<td>ID_3</td>
<td>EX_3</td>
<td>MA_3</td>
<td>WB_3</td>
<td>IF_4</td>
<td>ID_4</td>
<td>EX_4</td>
<td>MA_4</td>
<td>WB_4</td>
</tr>
<tr>
<td>IF_5</td>
<td>ID_5</td>
<td>EX_5</td>
<td>MA_5</td>
<td>WB_5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5-Stage Pipelined Execution

Resource Usage Diagram

- **Fetch Phase (IF)**
- **Decode & Reg-Fetch Phase (ID)**
- **Execute Phase (EX)**
- **Memory Phase (MA)**
- **Write-Back Phase (WB)**

**Resources**

- **IF**
- **ID**
- **EX**
- **MA**
- **WB**

**Time**

- **t0**: I₁, I₂, I₃, I₄, I₅
- **t1**: I₂, I₃, I₄, I₅
- **t2**: I₃, I₄, I₅
- **t3**: I₄, I₅
- **t4**: I₅
- **t5**: I₅
- **t6**: I₅, I₆, I₇
- **t7**: I₅

...
Pipelined Execution: ALU Instructions

not quite correct!
Pipelined Execution: Need for Several IR’s
IRs and Control points

Are control points connected properly?
- Load/Store instructions
- ALU instructions
Pipelined DLX Datapath
without jumps
How Instructions can Interact with each other in a Pipeline

• An instruction in the pipeline may need a resource being used by another instruction in the pipeline structural hazard

• An instruction may produce data that is needed by a later instruction data hazard

• In the extreme case, an instruction may determine the next instruction to be executed control hazard (branches, interrupts,...)
Controlling 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).

Feedback to previous stages is used to stall or kill instructions.