Architecture
(Pipelined Implementation)

CSCI 2021: Machine Architecture and Organization

Antonia Zhai
Department Computer Science and Engineering
University of Minnesota
http://www.cs.umn.edu/~zhai

With Slides from Bryant

Overview

• General Principles of Pipelining
  • Goal
  • Difficulties
• Creating a Pipelined Y86 Processor
  • Rearranging SEQ
  • Inserting pipeline registers
  • Problems with data and control hazards

With Slides from Bryant
Real-World Pipelines: Car Washes

• Idea
  • Divide process into independent stages
  • Move objects through stages in sequence
  • At any given times, multiple objects being processed

Computational Example

• System
  • Computation requires total of 300 picoseconds
  • Additional 20 picoseconds to save result in register
  • Must have clock cycle of at least 320 ps
Register Operation

- Stores data bits
- For most of time acts as barrier between input and output
- As clock rises, loads input

Registers

- Stores word of data
- Collection of edge-triggered latches
- Loads input on rising edge of clock
3-Way Pipelined Version

- System
  - Divide combinational logic into 3 blocks of 100 ps each
  - Can begin new operation as soon as previous one passes through stage A.
    - Begin new operation every 120 ps
  - Overall latency increases
    - 360 ps from start to finish

Pipeline Diagrams

Unpipelined
- Cannot start new operation until previous one completes

3-Way Pipelined
- Up to 3 operations in process simultaneously
Limitations: Nonuniform Delays

- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages
Limitations: Register Overhead

- As try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining

SEQ Hardware

- Stages occur in sequence
- One operation in process at a time
SEQ+ Hardware

- Still sequential implementation
- Reorder PC stage to put at beginning

PC Stage
- Task is to select PC for current instruction
- Based on results computed by previous instruction

Processor State
- PC is no longer stored in register
- But, can determine PC based on other stored information

Adding Pipeline Registers
**Pipeline Stages**

- **Fetch**
  - Select current PC
  - Read instruction
  - Compute incremented PC

- **Decode**
  - Read program registers

- **Execute**
  - Operate ALU

- **Memory**
  - Read or write data memory

- **Write Back**
  - Update register file

**PIPE- Hardware**

- Pipeline registers hold intermediate values from instruction execution

- Forward (Upward) Paths
  - Values passed from one stage to next
  - Cannot jump past stages
    - e.g., valC passes through decode
Signal Naming Conventions

- $S_{\text{Field}}$
  - Value of Field held in stage S pipeline register
- $s_{\text{Field}}$
  - Value of Field computed in stage S

Feedback Paths

- Predicted PC
  - Guess value of next PC
- Branch information
  - Jump taken/not-taken
  - Fall-through or target address
- Return point
  - Read from memory
- Register updates
  - To register file write ports
Data Dependencies

- System
  - Each operation depends on result from preceding one
Data Hazards

- Result does not feed back around in time for next operation
- Pipelining has changed behavior of system

Data Dependencies in Processors

- Result from one instruction used as operand for another
  - Read-after-write (RAW) dependency
  - Very common in actual programs
  - Must make sure our pipeline handles these properly
    - Get correct results
    - Minimize performance impact
Data Dependencies: 3 Nop's

0x000: irmovl $10, %edx
0x006: irmovl $3, %eax
0x00c: nop
0x00d: nop
0x00e: nop
0x00f: addl %edx, %eax
0x010: halt

Data Dependencies: 2 Nop's

0x000: irmovl $10, %edx
0x006: irmovl $3, %eax
0x00c: nop
0x00d: nop
0x00e: addl %edx, %eax
0x010: halt
Data Dependencies: 1 Nop

# demo-h1.ys

1 2 3 4 5 6 7 8 9

0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c:nop
0x00d:addl %edx,%eax
0x00f: halt

With Slides from Bryant

Data Dependencies: No Nop

# demo-h0.ys

1 2 3 4 5 6 7 8

0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c:addl %edx,%eax
0x00e: halt

With Slides from Bryant
Pipeline Summary

- Concept
  - Break instruction execution into 5 stages
  - Run instructions through in pipelined mode
- Limitations
  - Can't handle dependencies between instructions when instructions follow too closely
  - Data dependencies
    - One instruction writes register, later one reads it
  - Control dependency
    - Instruction sets PC in way that pipeline did not predict correctly
    - Mispredicted branch and return
- Fixing the Pipeline
  - We'll do that next time

pipe/psim: The Pipeline Simulator

Compilation process:
- `./misc/hcl2c -n seq-std.hcl <seq-std.hcl >seq-std.c`
- `gcc -Wall -O2 -I../misc -o ssim seq-std.c ssim.c ../misc/isa.c -lm`

In pipe/psim.c:
```c
int sim_main(int argc, char **argv) {
    run_tty_sim();
}
Run_tty_sim() {
    icount = sim_run_pipe(instr_limit, 5*instr_limit, &run_status, &result_cc);
}
```
pipe/psim: The Pipeline Simulator

In pipe/psim.c:

```c
int sim_run_pipe(int max_instr, int max_cycle, byte_t *statusp, cc_t *ccp)
{
    while (icount < max_instr && ccount < max_cycle) {
        run_status = sim_step_pipe(max_instr-icount, ccount);
    }
}
```

pipe/psim: The Pipeline Simulator

In pipe/psim.c:

```c
static byte_t sim_step_pipe(int max_instr, int ccount)
{
    /* Update program-visible state */
    update_state(update_mem, update_cc);
    /* Update pipe registers */
    update_pipes();

    do_if_stage();
    do_mem_stage();
    do_ex_stage();
    do_id_wb_stages();
}
```
Make the pipelined processor work!

- Data Hazards
  - Instruction having register R as source follows shortly after instruction having register R as destination
  - Common condition, don’t want to slow down pipeline
- Control Hazards
  - Mispredict conditional branch
    - Our design predicts all branches as being taken
    - Naïve pipeline executes two extra instructions
  - Getting return address for ret instruction
    - Naïve pipeline executes three extra instructions
- Making Sure It Really Works
  - What if multiple special cases happen simultaneously?

Stalling for Data Dependencies

```
# demo-b2.ys
0x000: irmovl $10,%edx
  F D E M W
0x006: irmovl $3,%eax
  F D E M W
0x00c: nop
  F D E M W
  bubble
0x00d: nop
    E M W
0x00e: addl %edx,%eax
    F D E M W
0x010: halt
    F F D E M W
```

- If instruction follows too closely after one that writes register, slow it down
- Hold instruction in decode
- Dynamically inject nop into execute stage
Stall Condition

- **Source Registers**
  - srcA and srcB of current instruction in decode stage

- **Destination Registers**
  - dstE and dstM fields
  - Instructions in execute, memory, and write-back stages

- **Special Case**
  - Don't stall for register ID 15 (0xF)
    - Indicates absence of register operand
  - Don't stall for failed conditional move

Detecting Stall Condition

```c
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: nop
0x00d: nop
  bubble
0x00e: addl %edx,%eax
0x010: halt
```

Cycle 6

- `W_dstE = %eax`
- `W_valE = 3`

1. `srcA = %edx`
2. `srcB = %eax`

With Slides from Bryant
What Happens When Stalling?

- Stalling instruction held back in decode stage
- Following instruction stays in fetch stage
- Bubbles injected into execute stage
  - Like dynamically generated nop’s
  - Move through later stages

With Slides from Bryant
Implementing Stalling

• Pipeline Control
  • Combinational logic detects stall condition
  • Sets mode signals for how pipeline registers should update

Pipeline Register Modes

Normal

- Input = y
- Output = x
- stall = 0
- bubble = 0
- Rising clock
- Output = y

Stall

- Input = y
- Output = x
- stall = 1
- bubble = 0
- Rising clock
- Output = x

Bubble

- Input = y
- Output = x
- stall = 0
- bubble = 1
- Rising clock
- Output = \text{nop}
Data Forwarding

- Naïve Pipeline
  - Register isn’t written until completion of write-back stage
  - Source operands read from register file in decode stage
    - Needs to be in register file at start of stage
- Observation
  - Value generated in execute or memory stage
- Trick
  - Pass value directly from generating instruction to decode stage
  - Needs to be available at end of decode stage

Data Forwarding Example

```
# demo-h2.py
0x000: irmovl $10, %edx
0x004: irmovl $3, %eax
0x008: nop
0x009: nop
0x00a: addl %edx, %eax
0x010: halt
```

- `irmovl` in write-back stage
- Destination value in W pipeline register
- Forward as `valB` for decode stage

With Slides from Bryant
Bypass Paths

- Decode Stage
  - Forwarding logic selects valA and valB
  - Normally from register file
  - Forwarding: get valA or valB from later pipeline stage

- Forwarding Sources
  - Execute: valE
  - Memory: valE, valM
  - Write back: valE, valM

Data Forwarding Example #2

```
# demo-h0.ys
0x000: irmovl $10, %edx
0x006: irmovl $3, %eax
0x00c: addl %edx, %eax
0x00e: halt
```

- Register %edx
  - Generated by ALU during previous cycle
  - Forward from memory as valA

- Register %eax
  - Value just generated by ALU
  - Forward from execute as valB
With Slides from Bryant

Forwarding Priority

- # demo-priority.ys
- 0x000: irmovl $1, %eax
- 0x006: irmovl $2, %eax
- 0x00c: irmovl $3, %eax
- 0x012: rrmovl %eax, %edx
- 0x014: halt

- Multiple Forwarding Choices
  - Which one should have priority
  - Match serial semantics
  - Use matching value from earliest pipeline stage

Implementing Forwarding

- Add additional feedback paths from E, M, and W pipeline registers into decode stage
- Create logic blocks to select from multiple sources for valA and valB in decode stage
Limitation of Forwarding

• Load-use dependency
  • Value needed by end of decode stage in cycle 7
  • Value read from memory in memory stage of cycle 8

Avoiding Load/Use Hazard

• Stall using instruction for one cycle
  • Can then pick up loaded value by forwarding from memory stage
### Detecting Load/Use Hazard

<table>
<thead>
<tr>
<th>Condition</th>
<th>Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/Use Hazard</td>
<td>$E_{code}$ in { IMRMOV, IPOPL } &amp;&amp; $E_{dstM}$ in { d$<em>{srcA}$, d$</em>{srcB}$ }</td>
</tr>
</tbody>
</table>

### Control for Load/Use Hazard

```plaintext
# demo-luh.ys

0x000: irmovl$128,edx
0x006: irmovl $3,ecx
0x00c: rmmovl$ecx, 0(%edx)
0x012: irmovl$10,ebx
0x018: mrmovl (%edx),%eax # Load %eax
    bubble
0x01e: addl %ebx,%eax # Use %eax
0x020: halt
```

- Stall instructions in fetch and decode stages
- Inject bubble into execute stage

<table>
<thead>
<tr>
<th>Condition</th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/Use Hazard</td>
<td>stall</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
</tr>
</tbody>
</table>
Predicting the PC

• Start fetch of new instruction after current one has completed fetch stage
  - Not enough time to reliably determine next instruction
• Guess which instruction will follow
  - Recover if prediction was incorrect

Our Prediction Strategy

• Instructions that Don’t Transfer Control
  • Predict next PC to be valP
  • Always reliable
• Call and Unconditional Jumps
  • Predict next PC to be valC (destination)
  • Always reliable
• Conditional Jumps
  • Predict next PC to be valC (destination)
  • Only correct if branch is taken
    • Typically right 60% of time
• Return Instruction
  • Don’t try to predict
Recovering from PC Misprediction

- Mispredicted Jump
  - Will see branch condition flag once instruction reaches memory stage
  - Can get fall-through PC from valA (value $M_{valA}$)
- Return Instruction
  - Will get return PC when $ret$ reaches write-back stage ($W_{valM}$)

Branch Misprediction Example

demo-j.js

0x000: xorl %eax,%eax  # Not taken
0x002: jne t            # Not taken
0x007: irmovl $1, %eax  # Fall through
0x00d: nop
0x00e: nop
0x00f: nop
0x010: halt
0x011: t: irmovl $3, %edx  # Target (Should not execute)
0x017: irmovl $4, %ecx  # Should not execute
0x01d: irmovl $5, %edx  # Should not execute

- Should only execute first 8 instructions
Branch Misprediction Trace

Incorrectly execute two instructions at branch target

Return Example

demo-ret.yc

0x000: irmovl Stack,%esp # Initialize stack pointer
0x006: nop # Avoid hazard on %esp
0x007: nop
0x008: nop
0x009: call p # Procedure call
0x00e: irmovl $5,%esi # Return point
0x014: halt
0x020: .pos 0x20
0x020: p: nop # procedure
0x021: nop
0x022: nop
0x023: ret
0x024: irmovl $1,%eax # Should not be executed
0x02a: irmovl $2,%ecx # Should not be executed
0x030: irmovl $3,%edx # Should not be executed
0x036: irmovl $4,%ebx # Should not be executed
0x100: .pos 0x100
0x100: Stack: # Stack; Stack pointer
• Require lots of nops to avoid data hazards
Incorrect Return Example

```
0x023:    ret
0x024:    irmovl $1, %eax # Oops!
0x02a:    irmovl $2, %ecx # Oops!
0x030:    irmovl $3, %edx # Oops!
0x00e:    irmovl $5, %esi # Return
```

Incorrectly execute 3 instructions following `ret`.

Handling Misprediction

```
0x000:    xorl %eax, %eax
0x002:    jne target # Not taken
0x011:    t: irmovl $2, %edx # Target
0x017:    irmovl $3, %ebx # Target+1
0x007:    t: irmovl $1, %eax # Fall through
0x00d:    nop
```

**Predict branch as taken**
- Fetch 2 instructions at target

**Cancel when mispredicted**
- Detect branch not-taken in execute stage
- On following cycle, replace instructions in execute and decode by bubbles
- No side effects have occurred yet

With Slides from Bryant
Detecting Mispredicted Branch

Control for Misprediction

With Slides from Bryant
Return Example
demo-retb.ys

0x000:  irmovl Stack,%esp  # Initialize stack pointer
0x006:  call p               # Procedure call
0x00b:  irmovl $5,%esi      # Return point
0x011:  halt
0x020:  .pos 0x20
0x020:  p:  irmovl $-1,%edi  # procedure
0x026:  ret
0x027:  irmovl $1,%eax      # Should not be executed
0x02d:  irmovl $2,%ecx      # Should not be executed
0x033:  irmovl $3,%edx      # Should not be executed
0x039:  irmovl $4,%ebx      # Should not be executed
0x100:  .pos 0x100
0x100:  Stack:               # Stack: Stack pointer

- Previously executed three additional instructions

Correct Return Example
demo-retb

# demo-retb
0x026:  ret
       bubble
       bubble
       bubble
       0x00b:  irmovl $5,%esi  # Return

- As ret passes through pipeline, stall at fetch stage
  - While in decode, execute, and memory stage
- Inject bubble into decode stage
- Release stall when reach write-back stage

With Slides from Bryant
Detecting Return

<table>
<thead>
<tr>
<th>Condition</th>
<th>Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing ret</td>
<td>IRET in { D_icode, E_icode, M_icode }</td>
</tr>
</tbody>
</table>

Control for Return

# demo-retb
0x026:    ret
bubble
bubble
bubble
0x00b:  irmovl$5,%esi  # Return

<table>
<thead>
<tr>
<th>Condition</th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing ret</td>
<td>stall</td>
<td>bubble</td>
<td>normal</td>
<td>normal</td>
<td>normal</td>
</tr>
</tbody>
</table>
Implementing Pipeline Control

• Combinational logic generates pipeline control signals
• Action occurs at start of following cycle

Pipeline Summary

• Data Hazards
  • Most handled by forwarding
    • No performance penalty
  • Load/use hazard requires one cycle stall
• Control Hazards
  • Cancel instructions when detect mispredicted branch
    • Two clock cycles wasted
  • Stall fetch stage while ret passes through pipeline
    • Three clock cycles wasted

With Slides from Bryant