CMSC 611: Advanced Computer Architecture

Pipelining & Pipeline Hazards

Some material adapted from Mohamed Younis, UMBC CMSC 611 Spr 2003 course slides
Some material adapted from David Culler, UC Berkeley CS252, Spr 2003 course slides, ©2002 UC Berkeley
Some material adapted from Hennessy & Patterson / © 2003 Elsevier Science
• Washer takes 30 min, Dryer takes 40 min, folding takes 20 min
• Sequential laundry takes 6 hours for 4 loads
• If they learned pipelining, how long would laundry take?
• Pipelining means start work as soon as possible
• Pipelined laundry takes 3.5 hours for 4 loads
Pipelining Lessons

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduce speedup
- Stall for Dependencies
MIPS Instruction Set

- RISC characterized by the following features that simplify implementation:
  - All ALU operations apply only on registers
  - Memory is affected only by load and store
  - Instructions follow very few formats and typically are of the same size

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>target address</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>26 bits</td>
<td></td>
</tr>
</tbody>
</table>
```
MIPS Instruction Formats

- R-type (register)
  - Most operations
    - add $t1, $s3, $s4  \# $t1 = $s3 + $s4
  - rd, rs, rt all registers
  - op always 0, funct gives actual function
### MIPS Instruction Formats

- **I-type (immediate)**
  - ALU with one immediate operand
    - `addi $t1, $s2, 32`  # $t1 = $s2 + 32
  - Load, store within $\pm2^{15}$ of register
    - `lw $t0, 32($s2)`  # $s1 = s2[32]$ or *(32+s2)
  - Load immediate values
    - `lui $t0, 255`  # $t0 = (255<<16)
    - `li $t0, 255`

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>21</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
</tr>
</tbody>
</table>

---

### MIPS Instruction Formats Diagram

![Diagram of MIPS Instruction Formats with fields: opr, rs, rt, immediate]
MIPS Instruction Formats

- **I-type (immediate)**
  - PC-relative conditional branch
  - $\pm 2^{15}$ from PC *after* instruction
    - `beq $s1, $s2, L1`  # goto L1 if ($s1 = $s2)
    - `bne $s1, $s2, L1`  # goto L1 if ($s1 \neq $s2)
MIPS Instruction Formats

• J-type (jump)
  – unconditional jump
    • j L1 # goto L1
  – Address is concatenated to top bits of PC
    • Fixed addressing within $2^{26}$

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>target address</td>
<td>6 bits</td>
</tr>
</tbody>
</table>
Single-cycle Execution

Figure: Dave Patterson
Multi-Cycle Implementation of MIPS

1. **Instruction fetch cycle (IF)**
   
   IR $\leftarrow$ Mem[PC]; \hspace{1em} NPC $\leftarrow$ PC + 4

2. **Instruction decode/register fetch cycle (ID)**
   
   A $\leftarrow$ Regs[IR$_{6..10}$]; \hspace{1em} B $\leftarrow$ Regs[IR$_{11..15}$]; \hspace{1em} Imm $\leftarrow$ ((IR$_{16}$)$^{16}$ ## IR$_{16..31}$)

3. **Execution/effective address cycle (EX)**
   
   Memory ref: \hspace{1em} ALUOutput $\leftarrow$ A + Imm;
   Reg-Reg ALU: \hspace{1em} ALUOutput $\leftarrow$ A $\text{func}$ B;
   Reg-Imm ALU: \hspace{1em} ALUOutput $\leftarrow$ A $\text{op}$ Imm;
   Branch: \hspace{1em} ALUOutput $\leftarrow$ NPC + Imm; \hspace{1em} Cond $\leftarrow$ (A op 0)

4. **Memory access/branch completion cycle (MEM)**
   
   Memory ref: \hspace{1em} LMD $\leftarrow$ Mem[ALUOutput] or Mem(ALUOutput) $\leftarrow$ B;
   Branch: \hspace{1em} if (cond) PC $\leftarrow$ ALUOutput;

5. **Write-back cycle (WB)**
   
   Reg-Reg ALU: \hspace{1em} Regs[IR$_{16..20}$] $\leftarrow$ ALUOutput;
   Reg-Imm ALU: \hspace{1em} Regs[IR$_{11..15}$] $\leftarrow$ ALUOutput;
   Load: \hspace{1em} Regs[IR$_{11..15}$] $\leftarrow$ LMD;
Multi-cycle Execution

Figure: Dave Patterson
The load instruction is the longest

All instructions follow at most the following five steps:

- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory and update PC
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- WB: Write the data back to the register file
Instruction Pipelining

- Start handling next instruction while the current instruction is in progress
- Feasible when different devices at different stages

Program Flow

\[
\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}
\]

Pipelining improves performance by increasing instruction throughput
Example of Instruction Pipelining

<table>
<thead>
<tr>
<th>Instruction Order</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $1$, 100($0$)</td>
<td>2</td>
</tr>
<tr>
<td>lw $2$, 200($0$)</td>
<td>4</td>
</tr>
<tr>
<td>lw $3$, 300($0$)</td>
<td>6</td>
</tr>
</tbody>
</table>

Program execution order (in instructions)

Program execution order (in instructions)

Time between first & fourth instructions
is $3 \times 8 = 24$ ns

Ideal and upper bound for speedup is number of stages in the pipeline
- Cycle time long enough for longest instruction
- Shorter instructions waste time
- No overlap
• Cycle time long enough for longest stage
• Shorter stages waste time
• Shorter instructions can take fewer cycles
• No overlap
• Cycle time long enough for longest stage
• Shorter stages waste time
• No additional benefit from shorter instructions
• Overlap instruction execution
Pipeline Performance

• Pipeline increases the instruction throughput
  – not execution time of an individual instruction
• An individual instruction can be slower:
  – Additional pipeline control
  – Imbalance among pipeline stages
• Suppose we execute 100 instructions:
  – Single Cycle Machine
    • 45 ns/cycle x 1 CPI x 100 inst = 4500 ns
  – Multi-cycle Machine
    • 10 ns/cycle x 4.2 CPI (due to inst mix) x 100 inst = 4200 ns
  – Ideal 5 stages pipelined machine
    • 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns
• Lose performance due to fill and drain
Pipeline Datapath

- Every stage must be completed in one clock cycle to avoid stalls
- Values must be latched to ensure correct execution of instructions
- The PC multiplexer has moved to the IF stage to prevent two instructions from updating the PC simultaneously (in case of branch instruction)
## Pipeline Stage Interface

<table>
<thead>
<tr>
<th>Stage</th>
<th>Any Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>IF/ID.IR ← MEM[PC]; IF/ID.NPC,PC ← (if (EX/MEM.opcode == branch) &amp; EX/MEM.cond) {EX/MEM.ALUOutput} else {PC + 4} ;</td>
</tr>
<tr>
<td>ID</td>
<td>ID/EX.A = Regs[IF/ID.IR_6..10]; ID/EX.B ← Regs[IF/ID.IR_11..15]; ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← (IF/ID.IR_16)_16 ## IF/ID.IR_16..31;</td>
</tr>
<tr>
<td>EX</td>
<td>EX/MEM.IR = ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A func ID/EX.B; Or EX/MEM.ALUOutput ← ID/EX.A op ID/EX.Imm; EX/MEM.cond ← 0;</td>
</tr>
<tr>
<td>ALU</td>
<td>EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm;</td>
</tr>
<tr>
<td>Load or Store</td>
<td>EX/MEM.ALUOutput ← ID/EX.IR;</td>
</tr>
<tr>
<td>Branch</td>
<td>EX/MEM.cond ← 0; EX/MEM.B ← ID/EX.B;</td>
</tr>
<tr>
<td>MEM</td>
<td>MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOutput ← EX/MEM.ALUOutput;</td>
</tr>
<tr>
<td>MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput]; Or Mem[EX/MEM.ALUOutput] ← EX/MEM.B;</td>
<td></td>
</tr>
<tr>
<td>For load only: Regs[MEM/WB.IR_11..15] ← MEM/WB.LMD;</td>
<td></td>
</tr>
</tbody>
</table>
Pipeline Hazards

- Cases that affect instruction execution semantics and thus need to be detected and corrected

- Hazards types
  - **Structural hazard**: attempt to use a resource two different ways at same time
    - Single memory for instruction and data
  - **Data hazard**: attempt to use item before it is ready
    - Instruction depends on result of prior instruction still in the pipeline
  - **Control hazard**: attempt to make a decision before condition is evaluated
    - branch instructions

- Hazards can always be resolved by waiting
Visualizing Pipelining

Time (clock cycles)

Cycle 1  Cycle 2  Cycle 3  Cycle 4  Cycle 5  Cycle 6  Cycle 7

Instr. Order

Slide: David Culler
Example: One Memory Port/Structural Hazard

Time (clock cycles)

Load
Instr 1
Instr 2
Instr 3
Instr 4

Structural Hazard

Slide: David Culler
Resolving Structural Hazards

1. Wait
   - Must detect the hazard
     • Easier with uniform ISA
   - Must have mechanism to stall
     • Easier with uniform pipeline organization

2. Throw more hardware at the problem
   - Use instruction & data cache rather than direct access to memory
Detecting and Resolving Structural Hazard

Time (clock cycles)

Cycle 1: Load
Cycle 2: Instr 1
Cycle 3: Instr 2
Cycle 4: Stall
Cycle 5: Instr 3
Cycle 6
Cycle 7

Detecting and Resolving Structural Hazard

Slide: David Culler
Stalls & Pipeline Performance

Pipelining Speedup = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}
= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \cdot \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}

Ideal CPI pipelined = 1

CPI pipelined = \text{Ideal CPI} + \text{Pipeline stall cycles per instruction}
= 1 + \text{Pipeline stall cycles per instruction}

Speedup = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}} \cdot \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}

Assuming all pipeline stages are balanced

Speedup = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}
### Data Hazards

**Instruction Order**

- **add** $r1, r2, r3$
- **sub** $r4, r1, r3$
- **and** $r6, r1, r7$
- **or** $r8, r1, r9$
- **xor** $r10, r1, r11$
Three Generic Data Hazards

- **Read After Write (RAW)**
  Instr\(_j\) tries to read operand before Instr\(_i\) writes it

  \[
  \begin{align*}
  I: & \quad \text{add } r1,r2,r3 \\
  J: & \quad \text{sub } r4,r1,r3
  \end{align*}
  \]

- **Caused by a “Data Dependence”** (in compiler nomenclature). This hazard results from an actual need for communication.
Three Generic Data Hazards

• Write After Read (WAR)  
  $\text{Instr}_j$ writes operand before $\text{Instr}_i$ reads it

  $I$: sub $r4, r1, r3$
  $J$: add $r1, r2, r3$
  $K$: mul $r6, r1, r7$

• Called an “anti-dependence” in compilers.
  – This results from reuse of the name “r1”.

• Can’t happen in MIPS 5 stage pipeline because:
  – All instructions take 5 stages, and
  – Reads are always in stage 2, and
  – Writes are always in stage 5
Three Generic Data Hazards

- **Write After Write (WAW)**
  Instr$_j$ writes operand before Instr$_i$ writes it.

  I: $\text{mul } r1, r4, r3$
  J: $\text{add } r1, r2, r3$
  K: $\text{sub } r6, r1, r7$

- Called an “output dependence” in compilers
  - This also results from the reuse of name “r1”.

- Can’t happen in MIPS 5 stage pipeline:
  - All instructions take 5 stages, and
  - Writes are always in stage 5

- Do see WAR and WAW in more complicated pipes
Forwarding to Avoid Data Hazard

- add r1, r2, r3
- sub r4, r1, r3
- and r6, r1, r7
- or r8, r1, r9
- xor r10, r1, r11
HW Change for Forwarding
Data Hazard Even with Forwarding

Time (clock cycles)

lw r1, 0(r2)

sub r4, r1, r6

and r6, r1, r7

or r8, r1, r9
Resolving Load Hazards

- Adding hardware? How? Where?
- Detection?
- Compilation techniques?
- What is the cost of load delays?
Resolving the Load Data Hazard

Time (clock cycles)

Iw r1, 0(r2)

sub r4, r1, r6

and r6, r1, r7

or r8, r1, r9

How is this different from the instruction issue stall?
Try producing fast code for

\[
a = b + c; \\
d = e - f;
\]

assuming a, b, c, d, e, and f in memory.

### Slow code:

- LW Rb,b
- LW Rc,c
- ADD Ra,Rb,Rc
- SW a,Ra
- LW Re,e
- LW Re,e
- LW Rf,f
- SUB Rd,Re,Rf
- SW d,Rd

### Fast code:

- LW Rb,b
- LW Rc,c
- LW Re,e
- ADD Ra,Rb,Rc
- LW Rf,f
- SW a,Ra
- SUB Rd,Re,Rf
- SW d,Rd
Instruction Set Connection

• What is exposed about this organizational hazard in the instruction set?
• k cycle delay?
  – bad, CPI is not part of ISA
• k instruction slot delay
  – load should not be followed by use of the value in the next k instructions
• Nothing, but code can reduce run-time delays
• MIPS did the transformation in the assembler
Control Hazard on Branches
Three Stage Stall

10: beq r1, r3, 36
14: and r2, r3, r5
18: or r6, r1, r7
22: add r8, r1, r9
36: xor r10, r1, r11
Example: Branch Stall Impact

• If 30% branch, Stall 3 cycles significant
• Two part solution:
  – Determine branch taken or not sooner, AND
  – Compute taken branch address earlier
• MIPS branch tests if register = 0 or ≠ 0
• MIPS Solution:
  – Move Zero test to ID/RF stage
  – Adder to calculate new PC in ID/RF stage
  – 1 clock cycle penalty for branch versus 3
Pipelined MIPS Datapath
Four Branch Hazard

Alternatives

1. Stall until branch direction is clear

2. Predict Branch Not Taken
   - Execute successor instructions in sequence
   - “Squash” instructions in pipeline if branch taken
   - Advantage of late pipeline state update
   - 47% MIPS branches not taken on average
   - PC+4 already calculated, so use it to get next instruction

3. Predict Branch Taken
   - 53% MIPS branches taken on average
   - But haven’t calculated branch target address in MIPS
     • MIPS still incurs 1 cycle branch penalty
     • Other machines: branch target known before outcome
4. Delayed Branch
   - Define branch to take place AFTER a following instruction
     branch instruction
     sequential successor$_1$
     sequential successor$_2$
     ..........  
     sequential successor$_n$

     .......... 
     branch target if taken

   - 1 slot delay allows proper decision and branch target address in 5 stage pipeline
   - MIPS uses this
Delayed Branch

• Where to get branch delay slot instructions?
  – Before branch instruction
  – From the target address
    • only valuable when branch taken
  – From fall through
    • only valuable when branch not taken
  – Canceling branches allow more slots to be filled

• Compiler effectiveness for single delay slot:
  – Fills about 60% of branch delay slots
  – About 80% of instructions executed in branch delay slots useful in computation
  – About 50% (60% x 80%) of slots usefully filled

• Delayed Branch downside: 7-8 stage pipelines, multiple instructions issued per clock (superscalar)
Example: Evaluating Branch Alternatives

Pipeline speedup = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}}

= \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}

Assume:
14% Conditional & Unconditional
65% Change PC

<table>
<thead>
<tr>
<th>Scheduling Scheme</th>
<th>Branch Penalty</th>
<th>CPI</th>
<th>Pipeline Speedup</th>
<th>Speedup vs stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall pipeline</td>
<td>3.0</td>
<td>1.420</td>
<td>3.52</td>
<td>1.00</td>
</tr>
<tr>
<td>Predict taken</td>
<td>1.0</td>
<td>1.140</td>
<td>4.39</td>
<td>1.25</td>
</tr>
<tr>
<td>Predict not taken</td>
<td>1.0</td>
<td>1.091</td>
<td>4.58</td>
<td>1.30</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.5</td>
<td>1.007</td>
<td>4.97</td>
<td>1.41</td>
</tr>
</tbody>
</table>
Scheduling Branch-Delay Slots

(a) From before

ADD R1, R2, R3
if R2 = 0 then
  Delay slot

Becomes

if R2 = 0 then
  ADD R1, R2, R3

(b) From target

SUB R4, R5, R6
ADD R1, R2, R3
if R1 = 0 then
  Delay slot

Becomes

ADD R1, R2, R3
ADD R1, R2, R3
if R1 = 0 then
  SUB R4, R5, R6

(c) From fall through

ADD R1, R2, R3
if R1 = 0 then
  SUB R4, R5, R6

Becomes

Best scenario

Good for loops

Good taken strategy

R4 must be temp reg.
Branch-Delay Scheduling
Requirements

<table>
<thead>
<tr>
<th>Scheduling Strategy</th>
<th>Requirements</th>
<th>Improves performance when?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) From before</td>
<td>Branch must not depend on the rescheduled instructions</td>
<td>Always</td>
</tr>
<tr>
<td>(b) From target</td>
<td>Must be OK to execute rescheduled instructions if branch is not taken. May need to duplicate instructions.</td>
<td>When branch is taken. May enlarge programs if instructions are duplicated.</td>
</tr>
<tr>
<td>(c) From fall through</td>
<td>Must be okay to execute instructions if branch is taken.</td>
<td>When branch is not taken.</td>
</tr>
</tbody>
</table>

- Limitation on delayed-branch scheduling arise from:
  - Restrictions on instructions scheduled into the delay slots
  - Ability to predict at compile-time whether a branch is likely to be taken
- May have to fill with a no-op instruction
  - Average 30% wasted
- Additional PC is needed to allow safe operation in case of interrupts (more on this later)
Static Branch Prediction

• Examination of program behavior
  – Assume branch is usually taken based on statistics but misprediction rate still 9%-59%

• Predict on branch direction forward/backward based on statistics and code generation convention
  – Profile information from earlier program runs
Exception Types

- I/O device request
- Breakpoint
- Integer arithmetic overflow
- FP arithmetic anomaly
- Page fault
- Misaligned memory accesses
- Memory-protection violation
- Undefined instruction
- Privilege violation
- Hardware and power failure
Exception Requirements

• Synchronous vs. asynchronous
  – I/O exceptions: Asynchronous
    • Allow completion of current instruction
  – Exceptions within instruction: Synchronous
    • Harder to deal with

• User requested vs. coerced
  – Requested predictable and easier to handle

• User maskable vs. unmaskable

• Resume vs. terminate
  – Easier to implement exceptions that terminate program execution
Stopping & Restarting Execution

• Some exceptions require restart of instruction
  – e.g. Page fault in MEM stage

• When exception occurs, pipeline control can:
  – Force a trap instruction into the pipeline next IF
  – Until the trap is taken, turn off all writes for the faulting (and later) instructions
  – OS exception-handling routine saves faulting instruction PC
Stopping & Restarting Execution

- Precise exceptions
  - Instructions before the faulting one complete
  - Instructions after it restart
  - As if execution were serial
- Exception handling complex if faulting instruction can change state before exception occurs
- Precise exceptions simplifies OS
- Required for demand paging
### Exceptions in MIPS

<table>
<thead>
<tr>
<th>Pipeline Stage</th>
<th>Problem exceptions occurring</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Page fault on instruction fetch; misaligned memory access; memory protection violation</td>
</tr>
<tr>
<td>ID</td>
<td>Undefined or illegal opcode</td>
</tr>
<tr>
<td>EX</td>
<td>Arithmetic exception</td>
</tr>
<tr>
<td>MEM</td>
<td>Page fault on data fetch; misaligned memory access; memory protection violation</td>
</tr>
<tr>
<td>WB</td>
<td>None</td>
</tr>
</tbody>
</table>

- Multiple exceptions might occur since multiple instructions are executing
  - (LW followed by DIV might cause page fault and an arith. exceptions in same cycle)
- Exceptions can even occur out of order
  - IF page fault before preceding MEM page fault

**Pipeline exceptions must follow order of execution of faulting instructions not according to the time they occur**
Precise Exception Handling

• The MIPS Approach:
  – Hardware posts all exceptions caused by a given instruction in a status vector associated with the instruction
  – The exception status vector is carried along as the instruction goes down the pipeline
  – Once an exception indication is set in the exception status vector, any control signal that may cause a data value to be written is turned off
  – Upon entering the WB stage the exception status vector is checked and the exceptions, if any, will be handled according the time they occurred
  – Allowing an instruction to continue execution till the WB stage is not a problem since all write operations for that instruction will be disallowed
Instruction Set Complications

• Early-Write Instructions
  – MIPS only writes late in pipeline
  – Machines with multiple writes usually require capability to rollback the effect of an instruction
    • e.g. VAX auto-increment,
  – Instructions that update memory state during execution, e.g. string copy, may need to save & restore temporary registers

• Branching mechanisms
  – Complications from condition codes, predictive execution for exceptions prior to branch

• Variable, multi-cycle operations
  – Instruction can make multiple writes