18-447: Computer Architecture

Lecture 10: Data and Control Dependence Handling in Pipelined Microarchitectures

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2012, 2/20/2012
Reminder: Homeworks

- Homework 3
  - Due Feb 27
  - Out
  - 3 questions
    - LC-3b microcode
    - Adding REP MOVS to LC-3b
    - Pipelining
Reminder: Lab Assignments

- Lab Assignment 2
  - Was due Friday, Feb 17
  - How did it go?

- Lab Assignment 3
  - Due March 2 – Start early.
  - Individual assignment
    - No collaboration; please respect the honor code
  - Extra credit
    - Early check off: 5%
    - Fastest three designs: 5% + prizes
Readings for Today

- Pipelining
  - P&H Chapter 4.5-4.8
  - Pipelined LC-3b Microarchitecture Handout

- Optional
  - Hamacher et al. book, Chapter 6, “Pipelining”
Readings for Next Lecture

- Go over the “pipelining” lecture notes carefully
- We will augment them with supplementary readings
- Start your Lab Assignment 3!
CALCM Seminar Wednesday 4-5pm

- Feb 22, 2012, Wednesday
- Hamerschlag Hall D-210

- Stochastic Computing: Embracing Errors in Architecture and Design of Hardware and Software
- Prof. Rakesh Kumar, University of Illinois

- Hardware is allowed to produce errors that are exposed to the highest layers of software
- Hardware and software are optimized to maximize power savings afforded by relaxed correctness
Review of Last Lecture

- Basics of pipelining
  - Overheads of pipelining (ideal vs. non-ideal pipelines)
  - Pipeline registers
  - Pipelined control

- Issues in pipeline design
  - Balancing work
  - Keeping the pipeline correct, moving, and full

- What is a “stall”? How do you implement “stalling”? 
- What are the advantages/disadvantages of detecting dependences using scoreboarding as opposed to combinational logic?
- What are the advantages/disadvantages of having the compiler reorder code so that hardware does not need to detect data dependences?
Review: Issues in Pipeline Design

- Balancing work in pipeline stages
  - How many stages and what is done in each stage

- Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow
  - Handling dependences
    - Data
    - Control
  - Handling resource contention
  - Handling long-latency (multi-cycle) operations

- Handling exceptions, interrupts

- Advanced: Improving pipeline throughput
  - Minimizing stalls, minimizing CPI, minimizing cycle time
Review: Causes of Pipeline Stalls

- Resource contention

- Dependences (between instructions)
  - Data
  - Control

- Long-latency (multi-cycle) operations
Review: Dependences and Their Types

- Also called “dependency”

- Dependencies dictate ordering requirements between instructions

- Two types
  - Data dependence
  - Control dependence

- Resource contention is sometimes called resource dependence
  - However, this is not fundamental to (dictated by) program semantics, so we will treat it separately
Review: Handling Resource Contention

- Happens when instructions in two pipeline stages need the same resource

- Solution 1: Eliminate the cause of contention
  - Duplicate the resource or increase its throughput
    - E.g., use separate instruction and data memories (caches)
    - E.g., use multiple ports for memory structures

- Solution 2: Detect the resource contention and stall one of the contending stages
  - Which stage do you stall?
  - Example: What if you had a single read and write port for the register file?
Review: Data Dependences

- Types of data dependences
  - Flow dependence (true data dependence – read after write)
  - Output dependence (write after write)
  - Anti dependence (write after read)

- Which ones cause stalls in a pipelined machine?
  - For all of them, we need to ensure semantics of the program are correct
  - Flow dependences always need to be obeyed because they constitute true dependence on a value
  - Anti and output dependences exist due to limited number of architectural registers
    - They are dependence on a name, not a value
    - We will later see what we can do about them
Review: Data Dependence Types

Flow dependence
\[ r_3 \leftarrow r_1 \text{ op } r_2 \]
\[ r_5 \leftarrow r_3 \text{ op } r_4 \]
Read-after-Write (RAW)

Anti dependence
\[ r_3 \leftarrow r_1 \text{ op } r_2 \]
\[ r_1 \leftarrow r_4 \text{ op } r_5 \]
Write-after-Read (WAR)

Output-dependence
\[ r_3 \leftarrow r_1 \text{ op } r_2 \]
\[ r_5 \leftarrow r_3 \text{ op } r_4 \]
\[ r_3 \leftarrow r_6 \text{ op } r_7 \]
Write-after-Write (WAW)
Review: How to Handle Data Dependences

- Anti and output dependences are easier to handle
  - write to the destination in one stage and in program order

- Flow dependences are more interesting

- Five fundamental ways of handling flow dependences
  - Detect and stall
  - Detect and forward/bypass data to dependent instruction
  - Eliminate the need for detecting dependence at the software level
    - Software-based interlocking: No need to detect
  - Do something else (fine-grained multithreading)
    - No need to detect
  - Predict the needed values and execute “speculatively”
Review: Interlocking

- Detection of dependence between instructions in a pipelined processor to guarantee correct execution

- Software based interlocking vs.
- Hardware based interlocking

- MIPS acronym?
Review: Approaches to Dependence Detection (I)

- **Scoreboarding**
  - Each register in register file has a Valid bit associated with it
  - An instruction that is writing to the register resets the Valid bit
  - An instruction in Decode stage checks if all its source and destination registers are Valid
    - Yes: No need to stall... No dependence
    - No: Stall the instruction

- **Advantage:**
  - Simple. 1 bit per register

- **Disadvantage:**
  - Need to stall for all types of dependences, not only flow dep.
Review: Approaches to Dependence Detection (II)

- Combinational dependence check logic
  - Special logic that checks if any instruction in later stages is supposed to write to any source register of the instruction that is being decoded
  - Yes: stall the instruction/pipeline
  - No: no need to stall... no flow dependence

- Advantage:
  - No need to stall on anti and output dependences

- Disadvantage:
  - Logic is more complex than a scoreboard
  - Logic becomes more complex as we make the pipeline deeper and wider (superscalar)
A Special Case of Data Dependence

- Control dependence
  - Data dependence on the Instruction Pointer / Program Counter
Control Dependence

Question: What should the fetch PC be in the next cycle?
Answer: The address of the next instruction
- All instructions are control dependent on previous ones. Why?

If the fetched instruction is a non-control-flow instruction:
- Next Fetch PC is the address of the next-sequential instruction
- Easy to determine if we know the size of the fetched instruction

If the instruction that is fetched is a control-flow instruction:
- How do we determine the next Fetch PC?

In fact, how do we know whether or not the fetched instruction is a control-flow instruction?
Data Dependence Handling

in More Depth
Data Dependence Types

Flow dependence
\[ r_3 \leftarrow r_1 \text{ op } r_2 \text{ Read-after-Write (RAW)} \]
\[ r_5 \leftarrow r_3 \text{ op } r_4 \n\]

Anti dependence
\[ r_3 \leftarrow r_1 \text{ op } r_2 \text{ Write-after-Read (WAR)} \]
\[ r_1 \leftarrow r_4 \text{ op } r_5 \n\]

Output-dependence
\[ r_3 \leftarrow r_1 \text{ op } r_2 \text{ Write-after-Write (WAW)} \]
\[ r_5 \leftarrow r_3 \text{ op } r_4 \n\]
\[ r_3 \leftarrow r_6 \text{ op } r_7 \n\]
Remember: How to Handle Data Dependences

- Anti and output dependences are easier to handle
  - write to the destination in one stage and in program order

- Flow dependences are more interesting

- Five fundamental ways of handling flow dependences
  - Detect and stall
  - Detect and forward/bypass data to dependent instruction
  - Eliminate the need for detecting dependence at the software level
    - Software-based interlocking: No need to detect
  - Do something else (fine-grained multithreading)
    - No need to detect
  - Predict the needed values and execute “speculatively”
Following flow dependences lead to conflicts in the 5-stage pipeline.
Register Data Dependence Analysis

For a given pipeline, when is there a potential conflict between 2 data dependent instructions?
- dependence type: RAW, WAR, WAW?
- instruction types involved?
- distance between the two instructions?

<table>
<thead>
<tr>
<th>R/I-Type</th>
<th>LW</th>
<th>SW</th>
<th>Br</th>
<th>J</th>
<th>Jr</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>read RF</td>
<td>read RF</td>
<td>read RF</td>
<td>read RF</td>
<td>read RF</td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>write RF</td>
<td>write RF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

24
Safe and Unsafe Movement of Pipeline

- **RAW Dependence**
  - Stage X
  - \( j: r_k \leftarrow \) Reg Read
  - \( i: r_k \leftarrow \) Reg Write
  - \( i: O j \)
  - \( \text{dist}(i,j) \leq \text{dist}(X,Y) \Rightarrow \text{Unsafe to keep } j \text{ moving} \)
  - \( \text{dist}(i,j) > \text{dist}(X,Y) \Rightarrow \text{Safe} \)

- **WAR Dependence**
  - Stage Y
  - \( j: r_k \leftarrow \) Reg Write
  - \( i: r_k \leftarrow \) Reg Read
  - \( i: A j \)

- **WAW Dependence**
  - \( j: r_k \leftarrow \) Reg Write
  - \( i: r_k \leftarrow \) Reg Write
  - \( i: D j \)
RAW Dependence Analysis Example

<table>
<thead>
<tr>
<th></th>
<th>R/I-Type</th>
<th>LW</th>
<th>SW</th>
<th>Br</th>
<th>J</th>
<th>Jr</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>read RF</td>
<td>read RF</td>
<td>read RF</td>
<td>read RF</td>
<td>read RF</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>write RF</td>
<td>write RF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Instructions $I_A$ and $I_B$ (where $I_A$ comes before $I_B$) have RAW dependence iff
  - $I_B$ (R/I, LW, SW, Br or JR) reads a register written by $I_A$ (R/I or LW)
  - $\text{dist}(I_A, I_B) \leq \text{dist(ID, WB)} = 3$

What about WAW and WAR dependence?
What about memory data dependence?
Pipeline Stall: Resolving Data Dependence

Stall==make the dependent instruction wait until its source data value is available

1. stop all up-stream stages
2. drain all down-stream stages

Inst_i
Inst_j
Inst_k
Inst_l
Inst_h

\[
\begin{align*}
  i: r_x & \leftarrow _ \_ \\
  j: _ & \leftarrow r_x \\
  \text{dist}(i,j) &= 4
\end{align*}
\]

\[
\begin{array}{ccccccc}
  t_0 & t_1 & t_2 & t_3 & t_4 & t_5 \\
  \text{Inst}_h & \text{IF} & \text{ID} & \text{ALU} & \text{MEM} & \text{WB} \\
  \text{Inst}_i & i & \text{IF} & \text{ID} & \text{ALU} & \text{MEM} & \text{WB} \\
  \text{Inst}_j & j & \text{IF} & \text{ID} & \text{ID} & \text{ID} & \text{ID} \\
  \text{Inst}_k & \text{IF} & \text{IF} & \text{IF} & \text{IF} & \text{IF} \\
  \text{Inst}_l & \text{IF} & \text{IF} & \text{IF} & \text{IF} & \text{IF} \\
\end{array}
\]
How to Implement Stalling

- Stall
  - disable **PC** and **IR** latching; ensure stalled instruction stays in its stage
  - Insert “invalid” instructions/nops into the stage following the stalled one

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

28
Stall Conditions

- Instructions $I_A$ and $I_B$ (where $I_A$ comes before $I_B$) have RAW dependence iff
  - $I_B$ (R/I, LW, SW, Br or JR) reads a register written by $I_A$ (R/I or LW)
  - $\text{dist}(I_A, I_B) \leq \text{dist(ID, WB)} = 3$

- In other words, must stall when $I_B$ in ID stage wants to read a register to be written by $I_A$ in EX, MEM or WB stage
Stall Conditions

- **Helper functions**
  - $rs(I)$ returns the $rs$ field of $I$
  - $use_{rs}(I)$ returns true if $I$ requires $RF[rs]$ and $rs!=r0$

- **Stall when**
  - $(rs(I_{id})==dest_{EX} \&\& use_{rs}(I_{id}) \&\& RegWrite_{EX})$ or
  - $(rs(I_{id})==dest_{MEM} \&\& use_{rs}(I_{id}) \&\& RegWrite_{MEM})$ or
  - $(rs(I_{id})==dest_{WB} \&\& use_{rs}(I_{id}) \&\& RegWrite_{WB})$ or
  - $(rt(I_{id})==dest_{EX} \&\& use_{rt}(I_{id}) \&\& RegWrite_{EX})$ or
  - $(rt(I_{id})==dest_{MEM} \&\& use_{rt}(I_{id}) \&\& RegWrite_{MEM})$ or
  - $(rt(I_{id})==dest_{WB} \&\& use_{rt}(I_{id}) \&\& RegWrite_{WB})$

- It is crucial that the EX, MEM and WB stages continue to advance normally during stall cycles.
Impact of Stall on Performance

- Each stall cycle corresponds to 1 lost ALU cycle

- For a program with $N$ instructions and $S$ stall cycles,
  Average CPI=$\frac{(N+S)}{N}$

- $S$ depends on
  - frequency of RAW dependences
  - exact distance between the dependent instructions
  - distance between dependences

  suppose $i_1, i_2$ and $i_3$ all depend on $i_0$, once $i_1$’s dependence is resolved, $i_2$ and $i_3$ must be okay too
Sample Assembly (P&H)

- for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { ...... }

```assembly
for2tst:
  addi $s1, $s0, -1  ; 3 stalls
  slti $t0, $s1, 0    ; 3 stalls
  bne $t0, $zero, exit2
  sll  $t1, $s1, 2    ; 3 stalls
  add  $t2, $a0, $t1   ; 3 stalls
  lw   $t3, 0($t2)     
  lw   $t4, 4($t2)     ; 3 stalls
  slt  $t0, $t4, $t3   ; 3 stalls
  beq  $t0, $zero, exit2

..........
  addi $s1, $s1, -1
  j    for2tst

exit2:
```
Data Forwarding (or Data Bypassing)

- It is intuitive to think of RF as state
  - “add rx ry rz” literally means get values from RF[ry] and RF[rz] respectively and put result in RF[rx]

- But, RF is just a part of a computing abstraction
  - “add rx ry rz” means 1. get the results of the last instructions to define the values of RF[ry] and RF[rz], respectively, and 2. until another instruction redefines RF[rx], younger instructions that refers to RF[rx] should use this instruction’s result

- What matters is to maintain the correct “dataflow” between operations, thus

```
add     ra r- r-
addi    r- ra r-
```

```
                   IF  ID  EX  MEM  WB
          if not hit:  ID  EX  MEM  WB
```
Resolving RAW Dependence with Forwarding

- Instructions $I_A$ and $I_B$ (where $I_A$ comes before $I_B$) have RAW dependence iff
  - $I_B$ (R/I, LW, SW, Br or JR) reads a register written by $I_A$ (R/I or LW)
  - $\text{dist}(I_A, I_B) \leq \text{dist}(\text{ID, WB}) = 3$

- In other words, if $I_B$ in ID stage reads a register written by $I_A$ in EX, MEM or WB stage, then the operand required by $I_B$ is not yet in RF
  - Retrieve operand from datapath instead of the RF
  - Retrieve operand from the youngest definition if multiple definitions are outstanding
Data Forwarding Paths (v1)

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Data Forwarding Paths (v2)

Assumes RF forwards internally

b. With forwarding

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
if \( rs_{EX} \neq 0 \) && \( rs_{EX} = dest_{MEM} \) && \( RegWrite_{MEM} \) then
forward operand from MEM stage  // dist=1
else if \( rs_{EX} \neq 0 \) && \( rs_{EX} = dest_{WB} \) && \( RegWrite_{WB} \) then
forward operand from WB stage  // dist=2
else
use \( A_{EX} \) (operand from register file)  // dist >= 3

Ordering matters!! Must check youngest match first

Why doesn’t use\_rs( ) appear in the forwarding logic?
Data Forwarding (Dependence Analysis)

<table>
<thead>
<tr>
<th>R/I-Type</th>
<th>LW</th>
<th>SW</th>
<th>Br</th>
<th>J</th>
<th>Jr</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>use</td>
</tr>
<tr>
<td>EX</td>
<td>use</td>
<td>use</td>
<td>use</td>
<td>use</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>produce</td>
<td>(use)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Even with data-forwarding, RAW dependence on an immediately preceding LW instruction requires a stall
Sample Assembly, Revisited (P&H)

- for (j=i-1; j>=0 && v[j] > v[j+1]; j=1) { ...... }
  
  ```assembly
  addi $s1, $s0, -1
  for2tst:
  slti $t0, $s1, 0
  bne $t0, $zero, exit2
  sll $t1, $s1, 2
  add $t2, $a0, $t1
  lw $t3, 0($t2)
  lw $t4, 4($t2)
  nop
  slt $t0, $t4, $t3
  beq $t0, $zero, exit2
  ........
  addi $s1, $s1, -1
  j for2tst
  exit2:
  ```
Pipelining the LC-3b
Pipelining the LC-3b

- Let’s remember the single-bus datapath

- We’ll divide it into 5 stages
  - Fetch
  - Decode/RF Access
  - Address Generation/Execute
  - Memory
  - Store Result

- Conservative handling of data and control dependences
  - Stall on branch
  - Stall on flow dependence
An Example LC-3b Pipeline
Control of the LC-3b Pipeline

- Three types of control signals

- Datapath Control Signals
  - Control signals that control the operation of the datapath

- Control Store Signals
  - Control signals (microinstructions) stored in control store to be used in pipelined datapath (can be propagated to later stages than decode)

- Stall Signals
  - Ensure the pipeline operates correctly in the presence of dependencies
<table>
<thead>
<tr>
<th>Stage</th>
<th>Signal Name</th>
<th>Signal Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>FETCH</td>
<td>MEM.PCMUX/2;</td>
<td>PC+2 ;select pc=2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TARGET.PC ;select MEM.TARGET.PC (branch target)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TRAP.PC ;select MEM.TRAP.PC</td>
</tr>
<tr>
<td></td>
<td>LD.PC/1;</td>
<td>NO(0), LOAD(1) ;destination IR[11:9]</td>
</tr>
<tr>
<td></td>
<td>LD.DE/1;</td>
<td>NO(0), LOAD(1) ;destination R7</td>
</tr>
<tr>
<td>DECODE</td>
<td>DRMUX/1;</td>
<td>11.9 ;asserted if instruction needs SR1</td>
</tr>
<tr>
<td></td>
<td>SR1.NEEDED/1;</td>
<td>NO(0), YES(1) ;asserted if instruction needs SR2</td>
</tr>
<tr>
<td></td>
<td>SR2.NEEDED/1;</td>
<td>NO(0), YES(1) ;BR Opcode</td>
</tr>
<tr>
<td></td>
<td>DE.BR.OP/1;</td>
<td>NO(0), BR(1) ;source IR[2:0]</td>
</tr>
<tr>
<td></td>
<td>SR2.IDMUX/1;</td>
<td>2.0 ;source IR[11:9]</td>
</tr>
<tr>
<td></td>
<td>L.D.AGEX/1;</td>
<td>NO(0), LOAD(1) ;destination IR[11:9]</td>
</tr>
<tr>
<td></td>
<td>V.AGEX.LD.CC/1;</td>
<td>NO(0), LOAD(1) ;destination IR[11:9]</td>
</tr>
<tr>
<td></td>
<td>V.MEM.LD.CC/1;</td>
<td>NO(0), LOAD(1) ;destination IR[11:9]</td>
</tr>
<tr>
<td></td>
<td>V.SR.LD.CC/1;</td>
<td>NO(0), LOAD(1) ;destination IR[11:9]</td>
</tr>
<tr>
<td></td>
<td>V.AGEX.LD.REG/1;</td>
<td>NO(0), LOAD(1) ;destination IR[11:9]</td>
</tr>
<tr>
<td></td>
<td>V.MEM.LD.REG/1;</td>
<td>NO(0), LOAD(1) ;destination IR[11:9]</td>
</tr>
<tr>
<td></td>
<td>V.SR.LD.REG/1;</td>
<td>NO(0), LOAD(1) ;destination IR[11:9]</td>
</tr>
<tr>
<td>AGEX</td>
<td>ADDR1MUX/1;</td>
<td>NPC ;select value from AGEX.NPC</td>
</tr>
<tr>
<td></td>
<td>ADDR2MUX/2;</td>
<td>BaseR ;select value from AGEX.SR1(BaseR)</td>
</tr>
<tr>
<td></td>
<td>ADDRMUX/2;</td>
<td>ZERO ;select value from AGEX.SR1 zero</td>
</tr>
<tr>
<td></td>
<td></td>
<td>offset6 ;offset6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC.offset9 ;PC.offset9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC.offset11 ;PC.offset11</td>
</tr>
<tr>
<td></td>
<td>LSHFI/1;</td>
<td>NO(0), lbit Left shift(1)</td>
</tr>
<tr>
<td></td>
<td>ADDRESSEDMUX/1;</td>
<td>7.0 ;select LSHF(ZEXT[IR[7:0]],1)</td>
</tr>
<tr>
<td></td>
<td>SR2MUX/1;</td>
<td>ADDER ;select output of address adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SR2 ;select from AGEX.SR2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.0 ;select from AGEX.SR2</td>
</tr>
<tr>
<td></td>
<td>ALUK/2;</td>
<td>ADD(00), AND(01) ;select output of the shifter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XOR(10), PASSB(11)</td>
</tr>
<tr>
<td></td>
<td>ALU.RESULTMUX/1;</td>
<td>SHIFTER ;select output of the shifter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALU ;select output of the ALU</td>
</tr>
<tr>
<td></td>
<td>MEM/1;</td>
<td>NO(0), LOAD(1) ;select value from SR.ADDRESS</td>
</tr>
<tr>
<td>MEM</td>
<td>DCACHE.EN/1;</td>
<td>NO(0), YES(1) ;asserted if the instruction accesses memory</td>
</tr>
<tr>
<td></td>
<td>DCACHE.RW/1;</td>
<td>RD(0), WR(1) ;asserted if the instruction accesses memory</td>
</tr>
<tr>
<td></td>
<td>DATA.SIZE/1;</td>
<td>BYTE(0), WORD(1)</td>
</tr>
<tr>
<td></td>
<td>BR.OP/1;</td>
<td>NO(0), BR(1) ;asserted if the instruction accesses memory</td>
</tr>
<tr>
<td></td>
<td>UNCON.OP/1;</td>
<td>NO(0), Uncond.BR(1) ;asserted if the instruction accesses memory</td>
</tr>
<tr>
<td></td>
<td>TRAP.OP/1;</td>
<td>NO(0), Trap(1) ;asserted if the instruction accesses memory</td>
</tr>
<tr>
<td>SR</td>
<td>DR.VALUEMUX/2;</td>
<td>ADDRESS ;select value from SR.ADDRESS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DATA ;select value from SR.DATA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NPC ;select value from SR.NPC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALU ;select value from SR.ALU.RESULT</td>
</tr>
<tr>
<td></td>
<td>LD.REG/1;</td>
<td>NO(0), LOAD(1) ;select value from SR.ALU.RESULT</td>
</tr>
<tr>
<td></td>
<td>LD.CC/1;</td>
<td>NO(0), LOAD(1) ;select value from SR.ALU.RESULT</td>
</tr>
</tbody>
</table>

Table 1: Data Path Control Signals
†: The control signal is generated by logic in that stage
‡‡: The control signal is generated by logic in another stage
## Control Store in a Pipelined Machine

<table>
<thead>
<tr>
<th>Number</th>
<th>Signal Name</th>
<th>Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SR1.NEEDED</td>
<td>DECODE</td>
</tr>
<tr>
<td>1</td>
<td>SR2.NEEDED</td>
<td>DECODE</td>
</tr>
<tr>
<td>2</td>
<td>DRMUX</td>
<td>DECODE</td>
</tr>
<tr>
<td>3</td>
<td>ADDR1MUX</td>
<td>AGEX</td>
</tr>
<tr>
<td>4</td>
<td>ADDR2MUX1</td>
<td>AGEX</td>
</tr>
<tr>
<td>5</td>
<td>ADDR2MUX0</td>
<td>AGEX</td>
</tr>
<tr>
<td>6</td>
<td>LSHFL</td>
<td>AGEX</td>
</tr>
<tr>
<td>7</td>
<td>ADDRESSMUX</td>
<td>AGEX</td>
</tr>
<tr>
<td>8</td>
<td>SR2MUX</td>
<td>AGEX</td>
</tr>
<tr>
<td>9</td>
<td>ALUK1</td>
<td>AGEX</td>
</tr>
<tr>
<td>10</td>
<td>ALUK0</td>
<td>AGEX</td>
</tr>
<tr>
<td>11</td>
<td>ALU.RESULTMUX</td>
<td>AGEX</td>
</tr>
<tr>
<td>12</td>
<td>BR.OP</td>
<td>DECODE, MEM</td>
</tr>
<tr>
<td>13</td>
<td>UNCON.OP</td>
<td>MEM</td>
</tr>
<tr>
<td>14</td>
<td>TRAP.OP</td>
<td>MEM</td>
</tr>
<tr>
<td>15</td>
<td>BR.STALL</td>
<td>DECODE, AGEX, MEM</td>
</tr>
<tr>
<td>16</td>
<td>DCACHE.EN</td>
<td>MEM</td>
</tr>
<tr>
<td>17</td>
<td>DCACHE.RW</td>
<td>MEM</td>
</tr>
<tr>
<td>18</td>
<td>DATA.SIZE</td>
<td>MEM</td>
</tr>
<tr>
<td>19</td>
<td>DR.VALUEMUX1</td>
<td>SR</td>
</tr>
<tr>
<td>20</td>
<td>DR.VALUEMUX0</td>
<td>SR</td>
</tr>
<tr>
<td>21</td>
<td>LD.REG</td>
<td>AGEX, MEM, SR</td>
</tr>
<tr>
<td>22</td>
<td>LD.CC</td>
<td>AGEX, MEM, SR</td>
</tr>
</tbody>
</table>

Table 2: Control Store ROM Signals
Stall Signals

- Pipeline stall: Pipeline does not move because an operation in a stage cannot finish.
- Stall Signals: Ensure the pipeline operates correctly in the presence.
- Why could an operation in a stage not finish?

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Generated in</th>
<th>Generated in</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICACHE.R/1:</td>
<td>FETCH</td>
<td>NO, READY</td>
</tr>
<tr>
<td>DEP.STALL/1:</td>
<td>DEC</td>
<td>NO, STALL</td>
</tr>
<tr>
<td>V.DE.BR.STALL/1:</td>
<td>DEC</td>
<td>NO, STALL</td>
</tr>
<tr>
<td>V.AGEX.BR.STALL/1:</td>
<td>AGEX</td>
<td>NO, STALL</td>
</tr>
<tr>
<td>MEM.STALL/1:</td>
<td>MEM</td>
<td>NO, STALL</td>
</tr>
<tr>
<td>V.MEM.BR.STALL/1:</td>
<td>MEM</td>
<td>NO, STALL</td>
</tr>
</tbody>
</table>

Table 3: STALL Signals
Control Dependence Handling
Control Dependence

- Question: What should the fetch PC be in the next cycle?
- Answer: The address of the next instruction
  - All instructions are control dependent on previous ones. Why?

- If the fetched instruction is a non-control-flow instruction:
  - Next Fetch PC is the address of the next-sequential instruction
  - Easy to determine if we know the size of the fetched instruction

- If the instruction that is fetched is a control-flow instruction:
  - How do we determine the next Fetch PC?

- In fact, how do we know whether or not the fetched instruction is a control-flow instruction?
## Branch Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Direction at fetch time</th>
<th>Number of possible next fetch addresses</th>
<th>When is next fetch address resolved?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional</td>
<td>Unknown</td>
<td>2</td>
<td>Execution (register dependent)</td>
</tr>
<tr>
<td>Unconditional</td>
<td>Always taken</td>
<td>1</td>
<td>Decode (PC + offset)</td>
</tr>
<tr>
<td>Call</td>
<td>Always taken</td>
<td>1</td>
<td>Decode (PC + offset)</td>
</tr>
<tr>
<td>Return</td>
<td>Always taken</td>
<td>Many</td>
<td>Execution (register dependent)</td>
</tr>
<tr>
<td>Indirect</td>
<td>Always taken</td>
<td>Many</td>
<td>Execution (register dependent)</td>
</tr>
</tbody>
</table>

Different branch types can be handled differently.
How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions. Potential solutions:

  - If the instruction is a control-flow instruction:
    - Stall the pipeline until we know the next fetch address
    - Guess the next fetch address. How?
  - Employ delayed branching (branch delay slot)
  - Do something else (fine-grained multithreading)
  - Eliminate control-flow instructions (predicated execution)
  - Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)