Reminder: Homework 2

- Homework 2 out
  - Due February 11 (next Monday)
  - LC-3b microcode
  - ISA concepts, ISA vs. microarchitecture, microcoded machines

- Remember: Homework 1 solutions were out
Reminder: Lab Assignment 2

- Lab Assignment 1.5
  - Verilog practice
  - Not to be turned in

- Lab Assignment 2
  - Due Feb 15
  - Single-cycle MIPS implementation in Verilog
  - All labs are individual assignments
  - No collaboration; please respect the honor code
  - Do not forget the extra credit portion!
Homework 1 Grades

HW 1 Score Distribution

Average = 92%
Standard Deviation = 8%
Lab 1 Score Distribution

Average = 81%
Standard Deviation = 17%
Readings for Next Few Lectures

- P&H Chapter 4.9-4.11

  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts
Today’s Agenda

- Deep dive into pipelining
  - Dependence handling
Review: Pipelining: Basic Idea

- **Idea:**
  - Divide the instruction processing cycle into distinct “stages” of processing
  - Ensure there are enough hardware resources to process one instruction in each stage
  - **Process a different instruction in each stage**
    - Instructions consecutive in program order are processed in consecutive stages
- **Benefit:** Increases instruction processing throughput (1/CPI)
- **Downside:** ??
Review: Execution of Four Independent ADDs

- Multi-cycle: 4 cycles per instruction

<table>
<thead>
<tr>
<th>F</th>
<th>D</th>
<th>E</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
</tr>
</tbody>
</table>

  Time

- Pipelined: 4 cycles per 4 instructions (steady state)

<table>
<thead>
<tr>
<th>F</th>
<th>D</th>
<th>E</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
</tr>
</tbody>
</table>

  Is life always this beautiful?

  Time
Review: Pipelined Operation Example

Is life always this beautiful?

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Review: Instruction Pipeline: Not An Ideal Pipeline

- **Identical operations ... NOT!**
  - \( \Rightarrow \) different instructions do not need all stages
    - Forcing different instructions to go through the same multi-function pipe
      \( \Rightarrow \) external fragmentation (some pipe stages idle for some instructions)

- **Uniform suboperations ... NOT!**
  - \( \Rightarrow \) difficult to balance the different pipeline stages
    - Not all pipeline stages do the same amount of work
      \( \Rightarrow \) internal fragmentation (some pipe stages are too-fast but take the same clock cycle time)

- **Independent operations ... NOT!**
  - \( \Rightarrow \) instructions are not independent of each other
    - Need to detect and resolve inter-instruction dependencies to ensure the pipeline operates correctly
      \( \Rightarrow \) Pipeline is not always moving (it stalls)
Review: Fundamental 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
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 is 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
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)
Pipelined Operation Example

What if the SUB were dependent on LW?

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
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 wait until value is available in register file
  - Detect and forward/bypass data to dependent instruction
  - Detect and eliminate the dependence at the software level
    - No need for the hardware to detect dependence
  - Predict the needed value(s), execute “speculatively”, and verify
  - Do something else (fine-grained multithreading)
    - No need to detect
Interlocking

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

- Software based interlocking vs.
- Hardware based interlocking

- MIPS acronym?
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.
Not Stalling on Anti and Output Dependences

- What changes would you make to the scoreboard to enable this?
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 (flash-forward: think superscalar execution)
Once You Detect the Dependence in Hardware

- What do you do afterwards?

- Observation: Dependence between two instructions is detected before the communicated data value becomes available

- Option 1: Stall the dependent instruction right away
- Option 2: Stall the dependent instruction only when necessary → data forwarding/bypassing
- Option 3: ...
Data Forwarding/Bypassing

- **Problem:** A consumer (dependent) instruction has to wait in decode stage until the producer instruction writes its value in the register file.

- **Goal:** We do not want to stall the pipeline unnecessarily.

- **Observation:** The data value needed by the consumer instruction can be supplied directly from a later stage in the pipeline (instead of only from the register file).

- **Idea:** Add additional dependence check logic and data forwarding paths (buses) to supply the producer’s value to the consumer right after the value is available.

- **Benefit:** Consumer can move in the pipeline until the point the value can be supplied → less stalling.
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: More Depth & Implementation
Remember: 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)
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 wait until value is available in register file
  - Detect and forward/bypass data to dependent instruction
  - Detect and eliminate the dependence at the software level
    - No need for the hardware to detect dependence
  - Predict the needed value(s), execute “speculatively”, and verify
  - Do something else (fine-grained multithreading)
    - No need to detect
RAW Dependence Handling

- 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></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></td>
<td>read RF</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>
Safe and Unsafe Movement of Pipeline

\[ \text{dist}(i,j) \leq \text{dist}(X,Y) \Rightarrow \text{Unsafe to keep j moving} \]
\[ \text{dist}(i,j) > \text{dist}(X,Y) \Rightarrow \text{Safe} \]
RAW Dependence Analysis Example

- 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

$\text{Inst}_i$:

$t_0$:
- **IF**

$t_1$:
- **IF**

$t_2$:
- **ID**
- **ALU**

$t_3$:
- **ID**
- **MEM**

$t_4$:
- **ID**
- **WB**

$t_5$:
- **IF**
- **ID**
- **ALU**
- **MEM**

$\text{Inst}_j$:

$t_0$:
- **IF**

$t_1$:
- **IF**

$t_2$:
- **ID**
- **ALU**

$t_3$:
- **ID**
- **MEM**

$t_4$:
- **ID**
- **WB**

$t_5$:
- **ID**
- **ALU**
- **MEM**

$\text{Inst}_k$:

$t_0$:
- **IF**

$t_1$:
- **IF**

$t_2$:
- **IF**
- **ID**

$t_3$:
- **IF**
- **ID**
- **ID**

$t_4$:
- **IF**
- **ID**
- **ID**

$t_5$:
- **IF**
- **IF**
- **IF**

$\text{Inst}_l$:

$t_0$:
- **IF**

$t_1$:
- **IF**

$t_2$:
- **IF**

$t_3$:
- **IF**

$t_4$:
- **IF**

$t_5$:
- **IF**

$i: r_x \leftarrow _$
bubble

$j: _ \leftarrow r_x$
dist(i,j)=4

$\text{Inst}_h$:

$t_0$:
- **IF**

$t_1$:
- **ID**

$t_2$:
- **ALU**

$t_3$:
- **MEM**

$t_4$:
- **WB**

$t_5$:
- **ID**
- **ALU**
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
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 \( rs \neq r0 \)

- **Stall when**
  - \( (rs(IR_{id})==dest_{EX}) && use\_rs(IR_{id}) && RegWrite_{EX} \) or
  - \( (rs(IR_{id})==dest_{MEM}) && use\_rs(IR_{id}) && RegWrite_{MEM} \) or
  - \( (rs(IR_{id})==dest_{WB}) && use\_rs(IR_{id}) && RegWrite_{WB} \) or
  - \( (rt(IR_{id})==dest_{EX}) && use\_rt(IR_{id}) && RegWrite_{EX} \) or
  - \( (rt(IR_{id})==dest_{MEM}) && use\_rt(IR_{id}) && RegWrite_{MEM} \) or
  - \( (rt(IR_{id})==dest_{WB}) && use\_rt(IR_{id}) && RegWrite_{WB} \)

- **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, the Average CPI is 
  \[ \text{Average CPI} = \frac{N + S}{N} \]

- 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) { ...... }

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

```
<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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
  - $\Rightarrow$ retrieve operand from datapath instead of the RF
  - $\Rightarrow$ 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

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

if \( (rs_{EX} \neq 0) \&\& (rs_{EX} == \text{dest}_{MEM}) \&\& \text{RegWrite}_{MEM} \) then
forward operand from MEM stage  // dist=1
else if \( (rs_{EX} \neq 0) \&\& (rs_{EX} == \text{dest}_{WB}) \&\& \text{RegWrite}_{WB} \) then
forward operand from WB stage  // dist=2
else
use \( A_{EX} \) (operand from register file)  // dist \geq 3

Ordering matters!! Must check youngest match first

Why doesn’t \( \text{use}_{rs}( ) \) appear in the forwarding logic?

What does the above not take into account?
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) { ...... }
  
```
for2tst:  
addi $s1, $s0, -1

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 stages later 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)</td>
</tr>
<tr>
<td></td>
<td>LD.DE/1;†</td>
<td>NO(0), LOAD(1)</td>
</tr>
<tr>
<td>DECODE</td>
<td>DRMUX/1:</td>
<td>11.9; destination IR[11:9]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R7; destination R7</td>
</tr>
<tr>
<td></td>
<td>SR1.NEED/1:</td>
<td>NO(0), YES(1); asserted if instruction needs SR1</td>
</tr>
<tr>
<td></td>
<td>SR2.NEED/1:</td>
<td>NO(0), YES(1); asserted if instruction needs SR2</td>
</tr>
<tr>
<td></td>
<td>DE.BR.OP/1:</td>
<td>NO(0), BR(1); BR Opcode</td>
</tr>
<tr>
<td></td>
<td>SR2.IDMUX/1:</td>
<td>2.0; source IR[2:0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11.9; source IR[11:9]</td>
</tr>
<tr>
<td></td>
<td>LD.AGEX/1:†</td>
<td>NO(0), LOADX(1)</td>
</tr>
<tr>
<td></td>
<td>V.AGEX,LD.CC/1;††</td>
<td>NO(0), LOAD(1)</td>
</tr>
<tr>
<td></td>
<td>V.MEM,LD.CC/1;††</td>
<td>NO(0), LOAD(1)</td>
</tr>
<tr>
<td></td>
<td>V.SR,LD.CC/1;††</td>
<td>NO(0), LOAD(1)</td>
</tr>
<tr>
<td></td>
<td>V.AGEX,LD.REG/1;††</td>
<td>NO(0), LOAD(1)</td>
</tr>
<tr>
<td></td>
<td>V.MEM,LD.REG/1;††</td>
<td>NO(0), LOAD(1)</td>
</tr>
<tr>
<td></td>
<td>V.SR,LD.REG/1;††</td>
<td>NO(0), LOAD(1)</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></td>
<td>ZERO; select the value zero</td>
</tr>
<tr>
<td></td>
<td></td>
<td>offset6; select SEXT[IR[5:0]]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PCoffset9; select SEXT[IR[8:0]]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PCoffset11; select SEXT[IR[10:0]]</td>
</tr>
<tr>
<td></td>
<td>LSHIF/1:</td>
<td>NO(0), lbit Left shift(1)</td>
</tr>
<tr>
<td></td>
<td>ADDRESSMUX/1:</td>
<td>7.0; select LSHF(ZEXT[IR[7:0]],1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADDER; select output of address adder</td>
</tr>
<tr>
<td></td>
<td>SR2MUX/1:</td>
<td>SR2; select from AGEX.SR2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.0; IR[4:0]</td>
</tr>
<tr>
<td></td>
<td>ALUK/2:</td>
<td>ADD(00), AND(01)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XOR(10), PASSB(11)</td>
</tr>
<tr>
<td>ALU.ResultMUX/1:</td>
<td>SHIFTER; select output of the shifter</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALU; select tput out the ALU</td>
</tr>
<tr>
<td>MEM</td>
<td>LD.MEM/1;†</td>
<td>NO(0), LOAD(1)</td>
</tr>
<tr>
<td></td>
<td>DCACHE.ENABLE/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)</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); BR</td>
</tr>
<tr>
<td></td>
<td>UNCON.OP/1:</td>
<td>NO(0), Uncond.BR(1); JMP,RET, JSR, JSRR</td>
</tr>
<tr>
<td></td>
<td>TRAP.OP/1:</td>
<td>NO(0), Trap(1); TRAP</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)</td>
</tr>
<tr>
<td></td>
<td>LD.CC/1:</td>
<td>NO(0), LOAD(1)</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>LSHF1</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 complete
- Stall Signals: Ensure the pipeline operates correctly in the presence of such an operation
- Why could an operation in a stage not complete?

<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