Reminder: Homeworks

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

- Lab Assignment 2
  - Due Friday, Feb 17, at the end of the lab
  - Individual assignment
    - No collaboration; please respect the honor code

- Lab Assignment 3
  - Already out
  - Extra credit
    - Early check off: 5%
    - Fastest three designs: 5% + prizes
    - More on this later
Reminder: Extra Credit for Lab Assignment 2

- Complete your normal (single-cycle) implementation first, and get it checked off in lab.
- Then, implement the MIPS core using a microcoded approach similar to what we are discussing in class.

- We are not specifying any particular details of the microcode format or the microarchitecture; you should be creative.
- For the extra credit, the microcoded implementation should execute the same programs that your ordinary implementation does, and you should demo it by the normal lab deadline.
Readings for Today

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

- Optional
  - Hamacher et al. book, Chapter 6, “Pipelining”
Review: Pipelining: Basic Idea

- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: “Assembly line processing” of instructions

- 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: Start thinking about this...
Example: Execution of Four Independent ADDs

- Multi-cycle: 4 cycles per instruction

- Pipelined: 4 cycles per 4 instructions (steady state)
Review: The Laundry Analogy

- “place one dirty load of clothes in the washer”
- “when the washer is finished, place the wet load in the dryer”
- “when the dryer is finished, take out the dry load and fold”
- “when folding is finished, ask your roommate (??) to put the clothes away”

- steps to do a load are sequentially dependent
- no dependence between different loads
- different steps do not share resources

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Review: Pipelining Multiple Loads of Laundry

- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4
- latency per load is the same
Review: Pipelining Multiple Loads of Laundry: In Practice

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Pipelining Multiple Loads of Laundry: In Practice

Throughput restored (2 loads per hour) using 2 dryers

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

- Goal: Increase throughput with little increase in cost (hardware cost, in case of instruction processing)

- Repetition of identical operations
  - The same operation is repeated on a large number of different inputs

- Repetition of independent operations
  - No dependencies between repeated operations

- Uniformly partitionable suboperations
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)

- Good examples: automobile assembly line, doing laundry
  - What about instruction processing pipeline?
Ideal Pipelining

- Combinational logic (F,D,E,M,W) T psec
  - BW = ~(1/T)

- T/2 ps (F,D,E) T/2 ps (M,W)
  - BW = ~(2/T)

- T/3 ps (F,D) T/3 ps (E,M) T/3 ps (M,W)
  - BW = ~(3/T)
More Realistic Pipeline: Throughput

- Nonpipelined version with delay $T$
  \[ BW = \frac{1}{T+S} \] where $S =$ latch delay

- $k$-stage pipelined version
  \[ BW_{k\text{-stage}} = \frac{1}{T/k + S} \]
  \[ BW_{\text{max}} = \frac{1}{1 \text{ gate delay} + S} \]
More Realistic Pipeline: Cost

- Nonpipelined version with combinational cost $G$
  \[ \text{Cost} = G + L \quad \text{where} \quad L = \text{latch cost} \]

- $k$-stage pipelined version
  \[ \text{Cost}_{k\text{-stage}} = G + Lk \]
Pipelining Instruction Processing
Remember: The Instruction Processing Cycle

1. Instruction fetch (IF)
2. Instruction decode and register operand fetch (ID/RF)
3. Execute/Evaluate memory address (EX/AG)
4. Memory operand fetch (MEM)
5. Store/writeback result (WB)
Remember the Single-Cycle Uarch

```
Remember the Single-Cycle Uarch

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

**BW =~ (1/T)**
Is this the correct partitioning?
Why not 4 or 6 stages? Why not different boundaries?

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

5-stage speedup is 4, not 5 as predicated by the ideal model
Enabling Pipelined Processing: Pipeline Registers

No resource is used by more than 1 stage!

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

All instruction classes must follow the same path and timing through the pipeline stages. Any performance impact?

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

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

\[
\begin{array}{cccccc}
 t_0 & t_1 & t_2 & t_3 & t_4 & t_5 \\
 \text{Inst}_0 & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
 \text{Inst}_1 & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
 \text{Inst}_2 & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
 \text{Inst}_3 & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
 \text{Inst}_4 & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\end{array}
\]
### Illustrating Pipeline Operation: Resource View

<table>
<thead>
<tr>
<th></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>t₈</th>
<th>t₉</th>
<th>t₁₀</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IF</strong></td>
<td>I₀</td>
<td>I₁</td>
<td>I₂</td>
<td>I₃</td>
<td>I₄</td>
<td>I₅</td>
<td>I₆</td>
<td>I₇</td>
<td>I₈</td>
<td>I₉</td>
<td>I₁₀</td>
</tr>
<tr>
<td><strong>ID</strong></td>
<td>I₀</td>
<td>I₁</td>
<td>I₂</td>
<td>I₃</td>
<td>I₄</td>
<td>I₅</td>
<td>I₆</td>
<td>I₇</td>
<td>I₈</td>
<td>I₉</td>
<td></td>
</tr>
<tr>
<td><strong>EX</strong></td>
<td>I₀</td>
<td>I₁</td>
<td>I₂</td>
<td>I₃</td>
<td>I₄</td>
<td>I₅</td>
<td>I₆</td>
<td>I₇</td>
<td>I₈</td>
<td>I₉</td>
<td></td>
</tr>
<tr>
<td><strong>MEM</strong></td>
<td>I₀</td>
<td>I₁</td>
<td>I₂</td>
<td>I₃</td>
<td>I₄</td>
<td>I₅</td>
<td>I₆</td>
<td>I₇</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>WB</strong></td>
<td>I₀</td>
<td>I₁</td>
<td>I₂</td>
<td>I₃</td>
<td>I₄</td>
<td>I₅</td>
<td>I₆</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Control Points in a Pipeline

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

Identical set of control points as the single-cycle datapath!!
Control Signals in a Pipeline

- For a given instruction
  - same control signals as single-cycle, but
  - control signals required at different cycles, depending on stage
    ⇒ decode once using the same logic as single-cycle and buffer control signals until consumed

⇒ or carry relevant “instruction word/field” down the pipeline and decode locally within each stage (still same logic)

Which one is better?
Pipelined Control Signals

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

- Goal: Increase throughput with little increase in cost (hardware cost, in case of instruction processing)

- Repetition of identical operations
  - The same operation is repeated on a large number of different inputs

- Repetition of independent operations
  - No dependencies between repeated operations

- Uniformly partitionable suboperations
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)

- Good examples: automobile assembly line, doing laundry
  - What about instruction processing pipeline?
Instruction Pipeline: Not An Ideal Pipeline

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

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

- Independent operations ... NOT!
  => instructions are not independent of each other
    - Need to detect and resolve inter-instruction dependencies to ensure the pipeline operates correctly
      => Pipeline is not always moving (it stalls)
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
Causes of Pipeline Stalls

- Resource contention

- Dependences (between instructions)
  - Data
  - Control

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

- Also called “dependency” or *much less desirably* “hazard”

- 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
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?
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
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

- Four fundamental ways of handling flow dependences
  - Detect and stall
  - Detect and forward/bypass data to dependent instruction
  - Eliminate the dependence at the software level
    - No need to detect
  - Do something else (fine-grained multithreading)
    - No need to detect
  - Predict the needed values and execute “speculatively”
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.
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)
We did not cover the following slides in lecture. These are for your preparation for the next lecture.
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)
Delayed Branching (I)

- Change the semantics of a branch instruction
  - Branch after N instructions
  - Branch after N cycles

- Idea: Delay the execution of a branch. N instructions (delay slots) that come after the branch are always executed regardless of branch direction.

- Problem: How do you find instructions to fill the delay slots?
  - Branch must be independent of delay slot instructions

- Unconditional branch: Easier to find instructions to fill the delay slot
- Conditional branch: Condition computation should not depend on instructions in delay slots → difficult to fill the delay slot
Delayed Branching (II)

**Normal code:**

```
A
B
C
BC X
D
E
F
G
```

**Timeline:**

```
F E
A
B
A
C
B
BC
C
BC
--
BC
G --
```

**Delayed branch code:**

```
A
C
BC X
B
D
E
F
```

**Timeline:**

```
F E
A
C
A
BC
C
B
BC
G
B
```

6 cycles

5 cycles
Fancy Delayed Branching (III)

- Delayed branch with squashing
  - In SPARC
  - If the branch falls through (not taken), the delay slot instruction is not executed
  - Why could this help?

Normal code: Delayed branch code: Delayed branch w/ squashing:

X: | A | B | C | BC X | D | E
---|---|---|----|-----|---|---
X: | A | B | C | BC X | NOP | D | E
X: | A | B | C | BC X | A | D | E
Delayed Branching (IV)

**Advantages:**

+ Keeps the pipeline full with useful instructions assuming
  1. Number of delay slots == number of instructions to keep the pipeline full before the branch resolves
  2. All delay slots can be filled with useful instructions

**Disadvantages:**

-- Not easy to fill the delay slots (even with a 2-stage pipeline)
  1. Number of delay slots increases with pipeline depth, issue width, instruction window size.
  2. Number of delay slots should be variable with variable latency operations. Why?

-- Ties ISA semantics to hardware implementation
  -- SPARC, MIPS, HP-PA: 1 delay slot
  -- What if pipeline implementation changes with the next design?
Fine-Grained Multithreading

- Idea: Hardware has multiple thread contexts. Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, there is no need to fetch another instruction from the same thread
  - Branch resolution latency overlapped with execution of other threads’ instructions

+ No logic needed for handling control and data dependences within a thread
-- Single thread performance suffers
-- Does not overlap latency if not enough threads to cover the whole pipeline
-- Extra logic for keeping thread contexts
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>;select pc+2</td>
</tr>
<tr>
<td></td>
<td>TRAP.PC</td>
<td>;select MEM.TARGET.PC (branch target)</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>R7</td>
<td>;destination R7</td>
</tr>
<tr>
<td></td>
<td>SR1.NEEDED/1:</td>
<td>NO(0), YES(1) ;asserted if instruction needs SR1</td>
</tr>
<tr>
<td></td>
<td>SR2.NEEDED/1:</td>
<td>NO(0), YES(1) ;asserted if instruction needs SR2</td>
</tr>
<tr>
<td></td>
<td>DE.BR.OP1/1:</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>LD.AGEX/1;†</td>
<td>NO(0), LOAD(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>ADDR2MUX/2:</td>
<td>ZERO ;select the value zero</td>
</tr>
<tr>
<td></td>
<td>ADDR2MUX/2:</td>
<td>offset6 ;select SEXT[IR[5:0]]</td>
</tr>
<tr>
<td></td>
<td>ADDR2MUX/2:</td>
<td>PCoffset9 ;select SEXT[IR[8:0]]</td>
</tr>
<tr>
<td></td>
<td>ADDR2MUX/2:</td>
<td>PCoffset11 ;select SEXT[IR[10:0]]</td>
</tr>
<tr>
<td></td>
<td>LSHFI/1:</td>
<td>NO(0), 1bit Left shift(1)</td>
</tr>
<tr>
<td></td>
<td>ADDRESSMUX/1/1:</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>SR2MUX/1:</td>
<td>SR2 ;select from AGEX.SR2</td>
</tr>
<tr>
<td></td>
<td>SR2MUX/1:</td>
<td>4.0 ;[IR[4:0]]</td>
</tr>
<tr>
<td>ALU</td>
<td>ALUK/2:</td>
<td>AD(X0), AND(X01)</td>
</tr>
<tr>
<td></td>
<td>ALUK/2:</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>ALU.RESULTMUX/1:</td>
<td>ALU ;select put out the ALU</td>
</tr>
<tr>
<td></td>
<td>LD.MEM/1;†</td>
<td>NO(0), LOAD(1)</td>
</tr>
<tr>
<td>MEM</td>
<td>DCACHE.L/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.OP1/1:</td>
<td>NO(0), BR(1) ;BR</td>
</tr>
<tr>
<td></td>
<td>UNCON.OP1/1:</td>
<td>NO(0), Uncond.BR(1)</td>
</tr>
<tr>
<td></td>
<td>TRAP.OP1/1:</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>DATA</td>
<td>;select value from SR.DA</td>
</tr>
<tr>
<td></td>
<td>NPC</td>
<td>;select value from SR.NPC</td>
</tr>
<tr>
<td></td>
<td>ALU</td>
<td>;select value from SR.ALU</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.NEEDDED</td>
<td>DECODE</td>
</tr>
<tr>
<td>1</td>
<td>SR2.NEEDDED</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>ADDREXXMUX</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>AL.U.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.(VALUEUMUX)1</td>
<td>SR</td>
</tr>
<tr>
<td>20</td>
<td>DR.(VALUEUMUX)0</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>Mode</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