Pipelining Datapath and Control
§6.2-6.3

Partially adapted from slides by Mary Jane Irwin, Penn State
And Kurtis Kredo, UCD
Data Forwarding

• Take the result from the earliest point that it exists in any of the pipeline state registers and forward it to the functional units (e.g., the ALU) that need it that cycle.

• For ALU functional unit: the inputs can come from any pipeline register rather than just from ID/EX by
  – adding multiplexors to the inputs of the ALU
  – connecting the Rd write data in EX/MEM or MEM/WB to either (or both) of the EX’s stage Rs and Rt ALU mux inputs
  – adding the proper control hardware to control the new muxes

• Other functional units may need similar forwarding logic.

• With forwarding, the CPU can achieve a CPI of 1 even in the presence of data dependencies.
Data Forwarding Conditions

- Only forward when state changes
- Forwarding unnecessary in other cases
- Forward if either source register needs it
Data Forwarding Conditions

• Only forward when state changes
  – Use RegWrite control signal
  – Don’t forward if destination is $0$
  – Forward if previous destination current source

• Forwarding unnecessary in other cases

• Forward if either source register needs it
EX/MEM Forwarding

- Register value needed by next instruction
  - Calculated by ALU this clock cycle
  - Needed as input to ALU on next clock cycle

```
add $4, $5, $6
add $8, $4, $7
or  add $8, $7, $4
```
EX/MEM Forwarding

- **ID/EX**
  - R[Rs]
  - R[Rt]
  - Immediate

- **EX/MEM**
  - ALU
  - ALU Cntrl
  - MemWrite
  - RegDst

- **MEM/WB**
  - MemWrite
  - MemtoReg
  - RegWrite
EX/MEM Forwarding

Diagram showing the flow of data through the ID/EX, EX/MEM, and MEM/WB stages of a pipeline, with connections for registers R[Rs], R[Rt], Immediate, Rd, Rt, Rs, and signals for ALU, ALU Cntrl, MemWrite, RegWrite, R[Rs], R[Rt], MemtoReg, and RegDst.
MEM/WB Forwarding

- Register value needed two instructions later
  - Calculated by ALU this clock cycle
  - Needed as input to ALU in two clock cycles

```
add $4, $5, $6
add $8, $4, $7
or
add $8, $7, $4
```

Unrelated Instruction
MEM/WB Forwarding

MEM/WB Forwarding diagram with labels for various paths and units such as ID/EX, EX/MEM, MEM/WB, Forward Unit, ALU, RegDst, MemWrite, RegWrite, MemtoReg, R[Rs], R[Rt], Immediate, Rd, Rt, Rs.
MEM/WB Forwarding

- **R[Rs]**
- **R[Rt]**
- **Immediate**
- **Rd**
- **Rt**
- **Rs**

**Forward Unit**

**ALU Cntrl**

**ALU**

**MemWrite**

**RegWrite**

**MemtoReg**
Forwarding Complication

• Forward unit must forward most recent value
  – It may appear necessary to do MEM/WB and EX/MEM forwarding simultaneously
  – Only do EX/MEM forwarding this cycle
  – Do EX/MEM forwarding again next cycle

\[
\text{add } \$4, \$5, \$6 \\
\text{add } \$4, \$4, \$13 \\
\text{add } \$8, \$4, \$7
\]
Complete ALU Input Forwarding
Register Definition

• How can we specify a particular signal?
  – Each state register has a copy
  – May vary across stages

• Reference the register that contains the value
  – RegWrite value in EX/MEM state register
    \texttt{EX/MEM.RegWrite}
  – RegWrite value in MEM/WB state register
    \texttt{MEM/WB.RegWrite}
Forwarding Conditions

• We want to forward when
  – Previous instruction updates state
  – Previous destination used as current source
  – Previous destination not $0

• Data Hazard code
  \[
  \text{add } $4, $5, $6 \\
  \text{sub } $8, $4, $9
  \]

• How do we do this in hardware?
Forwarding Unit

ID/EX
- R[Rs]
- R[Rt]
- Immediate
- Rd
- Rt
- Rs

EX/MEM
- ALU
- ALU Cntrl
- MemWrite

MEM/WB
- RegWrite
- MemtoReg

Forward Unit
- RegDst
- Forward Unit
Forwarding Unit Details

```
EX/MEM.RegWrite
EX/MEM.RegisterRd[4]
ID/EX.RegisterRs[4]
:
:
EX/MEM.RegisterRd[0]
ID/EX.RegisterRs[0]
EX/MEM.RegisterRd[4]
  0
:
:
EX/MEM.RegisterRd[0]
  0
```

---

Forward
Forwarding Unit Details

EX/MEM.RegWrite

EX/MEM.RegisterRd[4]
ID/EX.RegisterRs[4]

ID/EX.RegisterRs[0]

EX/MEM.RegisterRd[0]

EX/MEM.RegisterRd[4]

0

EX/MEM.RegisterRd[0]

0

EX/MEM.RegisterRd ≠ 0

EX/MEM.RegisterRd = ID/EX.RegisterRs
Other Forwarding Possible

• Forwarding to Data Memory

\[
\text{add } \$4, \$5, \$6 \\
\text{sw } \$4, 40(\$7)
\]

• Data memory to data memory copy

\[
\text{lw } \$4, 16(\$7) \\
\text{sw } \$4, 40(\$7)
\]
Forwarding to Memory

• What happens here?
  add $5, $6, $7
  sw $5, 8($10)

• Forwarding must occur, but not through ALU
Forwarding to Memory
Forwarding to Memory

Diagram showing the flow of data through the ID/EX, EX/MEM, and MEM/WB stages of a pipeline. The diagram includes

- R[Rs] and R[Rt] signals
- Immediate input
- Rd, Rt, Rs inputs
- MemWrite path
- RegWrite output
- MemtoReg flow
- Forward Unit interaction with ALU
- ALU Cntrl signal flow
Load Use Hazards Require Stalls

• No forwarding can help

\[
\text{lw } \$4, 16(\$5) \\
\text{nop} \\
\text{add } \$8, \$4, \$7
\]

• Requires a Hazard Detection Unit
  – Detects hazards
  – Inserts pipeline bubble
Stalling The Pipeline

- Stalls occur by inserting pipeline bubble
  - Hold some state registers (stage repeats)
  - Allow other stages to continue processing

```
lw $4, 16($5)
add $8, $4, $7
```

- Adding becomes `nop`

```
add $8, $4, $7
```
Stalling The Pipeline

- Load Use Hazard code

  \[ \text{lw} \; \$4, \; 16(\$5) \]
  \[ \text{add} \; \$8, \; \$4, \; \$7 \]
Stalling The Pipeline

• Load Use Hazard code

\[
\text{lw } \$4, \ 16(\$5) \\
\text{add } \$8, \ \$4, \ \$7
\]
Stalling The Pipeline

- Load Use Hazard code

\[
\text{lw} \ $4, \ 16($5) \\
\text{add} \ $8, \ $4, \ $7
\]

Bubble Inserted

Stage Repeated
Stalling The Pipeline

• Load Use Hazard code

\[
\text{lw} \; \$4, \; 16(\$5) \\
\text{add} \; \$8, \; \$4, \; \$7
\]
Stalling The Pipeline

• Load Use Hazard code

\[
\text{lw } $4, 16($5) \\
\text{add } $8, $4, $7
\]

No Register Written
How To Stall The Pipeline

• Two ways to stall the pipeline
  – Set control signals to safe values
  – Change instruction

• Set control signals
  –
  –

• Change the instruction
  –
  –
How To Stall The Pipeline

• Two ways to stall the pipeline
  – Set control signals to safe values
  – Change instruction

• Set control signals
  – All control signals become 0
  – Only MemWrite and RegWrite need to be 0

• Change the instruction
  – Make destination register $0
  – MIPS \texttt{nop} = 0x00000000 (\texttt{sll } \$0, \$0, 0)
How To Stall The Pipeline

- Read Address
- IM
- Write Data
- IF/ID
- Control
- Read Addr 1
- Read Addr 2
- Read Data 1
- Read Data 2
- Write Addr
- Write Data
- ID/EX
How To Stall The Pipeline
Hazard Detection Details

• Stall pipeline when all of the following occur
  –
  –
  –
  –
  or
Hazard Detection Details

• Stall pipeline when all of the following occur
  - ID/EX.MemRead
  - ID/EX.RegisterRt ≠ 0
  - ID/EX.RegisterRt = IF/ID.RegisterRs
    or
  ID/EX.RegisterRt = IF/ID.RegisterRt
Control Hazard Review

• Caused when next instruction is unknown
  – Conditional branch result unknown
  – Unconditional branch before destination calculated
Control Hazards

- No simple way to handle control hazards
  - Stalls often required
  - Incorrect guesses lead to wasted cycles
- Forwarding handles most data hazards
  - Decreases CPI to nearly 1 for data hazards
  - Only specific situations require a stall
- Thankfully, control hazards occur less frequently
Stalling Control Hazards

- Stalling always possible, but affects CPI
CPU May Assume Branch Not Taken

- Always assume branch is not taken
  - No delay if branch not taken

```assembly
beq $4, $5, label
```

```assembly
add $7, $8, $9
```

```assembly
label:
```
Branch Delay

- A taken branch must flush instructions

```plaintext
beq $4, $5, label
add $7, $8, $9
or $10, $11, $12
...
label: sub $7, $8, $9
```
Branch Delay

- A taken branch must flush instructions
  
  ```
  beq $4, $5, label
  add $7, $8, $9
  or $10, $11, $12
  ...
  label: sub $7, $8, $9
  ```

```
IM
  sub

DM
  nop

Reg
  beq
```
Adding nop To Decode Stage
Adding nop To Decode Stage

The diagram illustrates the process flow for adding a nop (no operation) to the decode stage. The process involves reading the address and data from the instruction memory (IM) and then checking for hazards. The control logic determines whether to clear the decode stage or proceed to the next stage. The register file is accessed to read and write addresses and data. The diagram highlights the flow of information through the pipeline stages: IF/ID, ID/EX, and control logic for hazard detection.
Reducing Branch Delay

• Reduce delay by computing branch in Decode
  – Comparison hardware required
  – Stage longer: read Register File, then compare
  – Branch destination adder moved to Decode
  – Only removes one stall

• Forwarding logic required

```
add $4, $5, $6
or $9, $10, $3
beq $4, $7, label
```
Branch Delay in Deep Pipelines

• Deeper pipelines incur larger delay
Dynamic Branch Prediction

- For deep pipelines or superscalar CPUs static prediction too costly
- Decreasing transistor size means more room for other functionality
- Have CPU guess branch result based on history of previous branches
  - Branch Prediction Buffer: Table of instruction addresses and branch results
  - Branch Target Buffer: Table of instruction addresses and branch destinations
  - When instruction executed again, look at buffers to predict result and destination
Dynamic Branch Prediction

<table>
<thead>
<tr>
<th>Branch Prediction Buffer</th>
<th>Branch Target Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0x4C</strong></td>
<td>T, N, T, N, N, N</td>
</tr>
<tr>
<td><strong>0x84</strong></td>
<td>T, T, T, N, T, N</td>
</tr>
<tr>
<td><strong>0xF8</strong></td>
<td>T, N, N, N, N, T</td>
</tr>
</tbody>
</table>

Only lower portion of address used

Branch history depth depends on implementation

Only lower portion of address used

Branch destination
Dynamic Branch Prediction

• When prediction correct no delay occurs
  – Pipeline remains full
  – CPI remains near 1

• When prediction incorrect
  – Flush instructions currently in pipeline
  – Fetch correct instruction
  – CPI increases above 1
1-bit Branch Prediction

• CPU uses last branch result as prediction
  for: add $4, $5, $9

  ...
  addi $8, $8, -1
  bne $8, $0, for

• Loop executes 10 times
  – First time prediction
  – 2-9 predictions
  – Last time prediction

• Short history causes incorrect predictions
  – Branch almost always taken
  – Predicted incorrect at least twice
1-bit Branch Prediction

• CPU uses last branch result as prediction
  for: add $4, $5, $9

  ...

  addi $8, $8, -1
  bne $8, $0, for

• Loop executes 10 times
  – First time prediction incorrect (predicted not taken)
  – 2-9 predictions correct (taken)
  – Last time prediction incorrect (predicted taken)

• Short history causes incorrect predictions
  – Branch almost always taken
  – Predicted incorrect at least twice
2-bit Branch Prediction

- Require two consecutive wrong answers to change prediction
2-bit Branch Prediction

• Previous example
  
  for: add $4, $5, $9
  ...
  addi $8, $8, -1
  bne $8, $0, for

• 2-bit Prediction better
  – First prediction
  – 2-9 predictions
  – Last prediction

• Improves to one incorrect per loop
2-bit Branch Prediction

- Previous example
  for: add $4, $5, $9
  ...
  addi $8, $8, -1
  bne $8, $0, for

- 2-bit Prediction better
  - First prediction correct
  - 2-9 predictions correct
  - Last prediction incorrect

- Improves to one incorrect per loop
Other Branch Prediction Schemes

• Many other ways to predict branch results
  – Correlating predictor
    • Maintain local history (per branch instruction)
    • Maintain global history (for all branches)
    • Use combination of histories to make prediction
  – Tournament predictor
    • Maintain multiple predictors per branch instruction
    • Use selector to choose most accurate predictor