Tema 4: Disseny del processador escalar segmentat

Eduard Ayguadè i Josep Llosa

These slides have been prepared using material available at the companion web site for "Computer Organization & Design. The Hardware/Software Interface. Copyright 1998 Morgan Kaufmann Publishers." Some slides have also been adapted from "David Albonesi's Computer Organization course ECSE200 at the University of Rochester. Copyright 2002 David H. Albonesi and the University of Rochester."

Chapter outline

- Pipelining: the technique
- Pipelined processor motivation: increasing instruction throughput
- MIPS 5-stage pipeline
- Hazards
- Handling exceptions
Pipelining: the technique

- Function $F$ ...

- ... divided into subfunctions ...

- ... and isolated with staging registers:

Stage registers clocked:

- One new result every $t_c$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$</td>
<td>$in_1$</td>
<td>$in_2$</td>
<td>$in_3$</td>
<td>$in_4$</td>
<td>$in_5$</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$f_2$</td>
<td>$in_1$</td>
<td>$in_2$</td>
<td>$in_3$</td>
<td>$in_4$</td>
<td>$in_5$</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$f_3$</td>
<td>$in_1$</td>
<td>$in_2$</td>
<td>$in_3$</td>
<td>$in_4$</td>
<td>$in_5$</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$f_4$</td>
<td>$in_1$</td>
<td>$in_2$</td>
<td>$in_3$</td>
<td>$in_4$</td>
<td>$in_5$</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$f_5$</td>
<td>$in_1$</td>
<td>$in_2$</td>
<td>$in_3$</td>
<td>$in_4$</td>
<td>$in_5$</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

$t_c = \max(t_i) + \delta$

$\varepsilon \cdot t_c > T$

($\varepsilon$ = num. of stages)
Pipelining: the technique

- **Speed-up:**
  - Single result: \( S_1 = \frac{T}{(\varepsilon \cdot t_c)} < 1 \)
  - \( n \) results: \( S_n = \frac{n \cdot T}{((\varepsilon \cdot t_c) + (n-1) \cdot t_c)} \)

  - Limit when \( n \to \infty \), ideal case (equal \( t_i \), \( \delta = 0 \)): \( S_\infty = \varepsilon \)

- **Exercise:** make a pipelined design for a 4-bit adder

---

Pipelining motivation

- **Need both low CPI and high frequency for best performance**
  - Want a multicycle for high frequency, but need better CPI

- **Idea behind pipelining is to have a multicycle implementation that operates like a factory assembly line**
  - Each “worker” in the pipeline performs a particular task, hands off to the next “worker”, while getting new work
  - Tasks should take about the same time – if one “worker” is much faster than the rest, then that “worker” will stand idle
  - Once the assembly line is full, a new “product” (instruction) comes out of the back-end of the line each time period
Pipeline motivation

In a computer assembly line (pipeline), each task is called a stage and the time period is one clock cycle.

Program execution order (in instructions):

1. lw $10, 20($1)
2. sub $11, $2, $3

Time (in clock cycles): CC 1, CC 2, CC 3, CC 4, CC 5, CC 6
Evaluation of pipelined implementation

- \( t_c = 10 \text{ ns} \)
- CPI=5 cycles (only for first instruction)
- CPI=1 cycle (for the rest of instructions)
- Speedup = \( \frac{N \times 1 \times 40}{(1 \times 5 \times 10) + ((N-1) \times 1 \times 10)} \)
  - Speedup reaches 4 when \( N \rightarrow \infty \)

MIPS 5-stage pipeline

- Like single cycle datapath but with registers separating each stage
MIPS 5-stage pipeline

- 5 stages for each instruction
  - IF: instruction fetch
  - ID: instruction decode and register file read
  - EX: instruction execution or effective address calculation
  - MEM: memory access for load and store
  - WB: write back results to register file

- Delays of all 5 stages are relatively the same
- Staging registers are used to hold data and control as instructions pass between stages
- All instructions pass through all 5 stages
- As an instruction leaves a stage in a particular clock period, the next instruction enters it

Pipeline operation for lw

- Stage 1: Instruction fetch
Pipeline operation for lw

Stage 2: Instruction decode and register file read

Stage 3: Effective address calculation
Pipeline operation for lw

Stage 4: Memory access

Stage 5: Write back

Instruction info in IF/ID is gone – won’t work
Modified pipeline with write back fix

- Write register bits from the instruction must be carried through the pipeline with the instruction

Pipeline operation for lw and sub sequence
Pipeline operation for lw and sub sequence
Pipeline operation for lw and sub sequence

Clock 4

Pipeline operation for lw and sub sequence

Clock 5
Pipeline operation for lw and sub sequence

Basic control is similar to the “single cycle” design
Pipeline control

- Control is generated in ID and travels with the instruction and data through the pipeline.
- When an instruction enters a stage, it’s control signals set the operation of that stage.
For the following code fragment ...

lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9

... show the datapath and control usage as the instruction sequence travels down the pipeline
Multiple instruction example

Clock 4

Multiple instruction example

Clock 5
Multiple instruction example

Clock 6

Multiple instruction example

Clock 7
Pipeline hazards

- A hazard is a conflict, regarding data, control, or hardware resources
- **Data hazards**: a value is not available in registers when you need it
- **Control hazards**: you don’t know if an instruction needs to be executed
- **Structural hazards**: a resource is not available, such as:
  - A single memory for instructions and data
  - A multi-cycle, non-pipelined functional unit (such as a divider)

Data dependences

- A read-after-write (RAW) dependence occurs when the register written by an instruction is a source register of a subsequent instruction
  
  lw $10, 20($1)
  sub $11, $10, $3
  and $12, $4, $11
  or $13, $11, $4
  add $14, $13, $9

- Also have write-after-read (WAR) and write-after-write (WAW) data dependences (later)
Pipelining and RAW dependences

- RAW dependences that are close by may cause data hazards in the pipeline
- Consider the following code sequence:

```
sub $2, $1, $3
and $12, $2, $6
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

Data hazards with first three instructions

<table>
<thead>
<tr>
<th>Program execution order (in instructions)</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub $2, $1, $3</td>
<td>10</td>
<td>20</td>
<td>10</td>
<td>10</td>
<td>10-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
</tr>
<tr>
<td>and $12, $2, $6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or $13, $6, $2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $14, $2, $2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw $15, 100($2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipelining and RAW dependences

- Naive solution: stall the pipeline as many cycles as needed so that the RAW is honored

```plaintext
sub $2, $1, $3  F D E M W
and $12, $2, $5  F D D E E M W
or $13, $6, $2  F F F D E M W
add $14, $2, $2  F D E M W
sw $15, 100($2)  F D E M W
```

- Stalling the pipeline means:
  - Do not load the IF/ID register and the PC
  - Force the control equivalent to a NOP ("no operation") instruction into EX stage

Example

- Which is the CPI achieved in this sequence of instructions?

```plaintext
add $16, $18, $15
ld $24, 0($16)
ld $25, 4($16)
slt $8, $25, $24
beq $8, $0, exit2
```
Forwarding results between pipe stages

<table>
<thead>
<tr>
<th></th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of EX/MEM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Program execution order (in instructions):
- sub $b3$, $b1$, $b3$
- add $b12$, $b2$, $b6$
- or $b13$, $b6$, $b2$
- add $b14$, $b2$, $b2$
- sw $b10$, 100

- **at this point, result of sub is available**

Forwarding datapaths

- Bypass paths feed data from MEM and WB back to MUXes at the EX ALU inputs
- Do we still have to write the register file in WB?
Detecting forwarding from MEM to EX

- To the upper ALU input (ALU\textsubscript{upper})
  - EX/MEM.RegWrite = 1
  - EX/MEM.RegisterRd not equal 0
  - EX/MEM.RegisterRd = ID/EX.RegisterRs

- To the lower ALU input (ALU\textsubscript{lower})
  - EX/MEM.RegWrite = 1
  - EX/MEM.RegisterRd not equal 0
  - EX/MEM.RegisterRd = ID/EX.RegisterRt

Detecting forwarding from WB to EX

- To the upper ALU input
  - MEM/WB.RegWrite = 1
  - MEM/WB.RegisterRd not equal 0
  - MEM/WB.RegisterRd = ID/EX.RegisterRs
  - The value is not being forwarded from MEM (why?)

- To the lower ALU input
  - MEM/WB.RegWrite = 1
  - MEM/WB.RegisterRd not equal 0
  - MEM/WB.RegisterRd = ID/EX.RegisterRt
  - The value is not being forwarded from MEM
Forwarding control

Control is handled by the forwarding unit

RAW hazards involving loads

Loads produce results in MEM – can't forward to an immediately following R-type instruction
RAW hazards involving loads

Solution: stall the stages behind the load for one cycle, after which the result can be forwarded.

Detecting load-use hazards

Instruction in EX is a load
- ID/EX.MemRead = 1

Instruction in ID has a source register that matches the load destination register
- ID/EX.RegisterRt = IF/ID.RegisterRs OR
  ID/EX.RegisterRt = IF/ID.RegisterRt
Stalling the stages behind the load

- Force nop ("no operation") instruction into EX stage on next clock cycle
  - Force ID/EX.MemWrite input to zero
  - Force ID/EX.RegWrite input to zero

- Hold instructions in ID and IF stages for one clock cycle
  - Hold the contents of PC
  - Hold the contents of IF/ID

Control for load-use hazards

- Control is handled by the hazard detection unit
Which is the CPI achieved in this sequence of instructions, when forwarding is implemented?

```
add $16, $18, $15
ld $24, 0($16)
ld $25, 4($16)
slt $8, $25, $24
beq $8, $0, exit2
```

The new CPI is lower than the original one, but what about $t_c$?

---

Control hazards

- Taken branches and jumps change the PC to the target address from which the next instruction is to be fetched.
- In our pipeline, the PC is changed when the taken beq instruction is in the MEM stage.
When beq changes the PC?

```
instr_1+3

instr_1+2

instr_1+1

beq $2, $3, 7
```

Control hazards

Control hazard with the following sequential instructions: do they need to be executed?

<table>
<thead>
<tr>
<th>Program execution order (in instructions)</th>
<th>Time (in clock cycles)</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 beq $1, $3, 7</td>
<td>IM</td>
<td>IM</td>
<td>DL</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
</tr>
<tr>
<td>44 and $12, $2, $5</td>
<td>IM</td>
<td>IM</td>
<td>DL</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
</tr>
<tr>
<td>48 or $13, $6, $22</td>
<td>IM</td>
<td>IM</td>
<td>DL</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
</tr>
<tr>
<td>52 add $14, $2, $2</td>
<td>IM</td>
<td>IM</td>
<td>DL</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
</tr>
<tr>
<td>72 lw $4, 50($7)</td>
<td>IM</td>
<td>IM</td>
<td>DL</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
</tr>
</tbody>
</table>
Control hazards

- **Several options:**
  - The compiler inserts a number of nop ("no operation") after beq
  - Stall the pipeline as many cycles as needed (i.e. until the hazard disappears)
  - Continue execution and discard instructions if branch is taken: is that possible?
    - Nullified brach: convert instructions in the not-taken path into nop ("no operation")

- In our pipelined design, the *branch delay* is three

Delayed branching

- Try to execute useful instructions in the delay slots

- The ISA states that the instruction following the branch is always executed irregardless of the branch outcome
  - Hardware must adhere to this rule!

  ```
  beq $4, $8, 6
  inst in delay1
  inst in delay2
  inst in delay3
  ```

- The compiler finds an appropriate instruction to place after the branch (in the branch delay slot)
Delayed branching

The compiler finds instructions for the delay slot

- From before
  
  \[
  \text{add } s1, s2, s3 \\
  \text{if } s2 = 0 \text{ then} \\
  \text{Delay slot}
  \]

  Becomes

  \[
  \text{if } s2 = 0 \text{ then} \\
  \text{add } s1, s2, s3
  \]

- From target
  
  \[
  \text{sub } s4, s5, s6 \\
  \]

  ... 

  \[
  \text{add } s1, s2, s3 \\
  \text{if } s1 = 0 \text{ then} \\
  \text{Delay slot}
  \]

  Becomes

  \[
  \text{if } s1 = 0 \text{ then} \\
  \text{add } s1, s2, s3
  \text{sub } s4, s5, s6
  \]

- From full through
  
  \[
  \text{add } s1, s2, s3 \\
  \text{if } s1 = 0 \text{ then} \\
  \text{Delay slot}
  \]

  Becomes

  \[
  \text{add } s1, s2, s3 \\
  \text{if } s1 = 0 \text{ then} \\
  \text{sub } s4, s5, s6
  \]

Limitations of delayed branching

- Frequently, the compiler can't fill the delay slots with useful instructions while maintaining correctness (has to insert nops instead)

- Modern pipelines may have >10 delay slots
  - Multi-cycle instruction fetch and decode
  - Multiple instructions in each pipeline stage
  - Example:
    - Pipeline: IF1-IF2-ID1-ID2
    - Branch calculation performed in ID2
    - Four instructions in each stage
    - 12 delay slots

- Solution: branch prediction (later)
Reducing the delay slots

- Reducing the delay slots reduces:
  - the number of nop that need to be inserted, or
  - the number of stall cycles, or
  - the number of instructions that have to be discarded on a taken branch

- We can reduce the number of delay slots to one for beq by moving both the equality test and the branch target address calculation into ID
Forwarding and stalling changes

- Results in MEM and WB must be forwarded to ID for use as possible beq source operand values
- beq may have to stall in ID to wait for source operand values to be produced
- Examples

```
addi $2, $2, -1
beq $2, $0, 20
```

Stall beq one cycle; forward $2 from MEM to upper equality input in ID

```
lw $8, 20($1)
beq $4, $8, 6
```

Stall beq two cycles; forward $8 from WB to lower equality input in ID

Precise exceptions

- Exceptions require a change of control to a special handler routine
- The PC of the user program is saved in EPC and restored after the handler completes so that the user program can resume at that instruction
- For the user program to work correctly after resuming,
  - All instructions before the excepting one must have written their result
  - All subsequent instructions must not have written their result
- Exceptions handled this way are called precise
Pipelining and precise exceptions

- There may be instructions from before and after the excepting one when the exception occurs
- Exceptions may be detected out of program order

Exceptions may be detected out of program order

Each instruction in the pipeline has an exception field that travels with it
When an exception is detected, the type of exception is encoded in the exception field
The RegWrite and MemWrite control signals for the instruction are set to 0
At the end of MEM, the exception field is checked to see if an exception occurred
If so, the instructions in IF, ID, and EX are made into nops, and the address of the exception handler is loaded into the PC
Complete Data Path (with nullification)

Multicycle EX

- Instructions do not usually take the same number of EX cycles to complete:
  - e.g. floating point operations vs. Integer operations

- Two different pipelines inside the data path:
  - Integer data path: R-type, ld, st and beq
  - Floating-point data path
Multicycle EX

- Blocking one data path does not affect the other
- Blocking one FP operation blocks the whole FP data path

Examples:
- \texttt{mul.f $f1, $f2, $f3}
- \texttt{add.f $f4, $f1, $f2}
- \texttt{add $3, $4, $5}
- \texttt{ld $f5, 10($3)}
- \texttt{add.f $f4, $f5, $f5}
- \texttt{mul.f $f1, $f2, $f3}
- \texttt{add.f $f4, $f1, $f2}
- \texttt{st $f4, 10($1)}
- \texttt{mul.f $f5, $f2, $f3}
- \texttt{add.f $f6, $f5, $f5}

Should we block here?
... so now, blocking one FP operation does not block the whole FP data path.
**Structural hazards in the pipeline:**
- Two operations in the FP datapath can finish at the same time:
  - `mul.f $f1, $f2, $f3`
  - `add.f $f4, $f2, $f2`
  - `add.f $f5, $f3, $f3`

**New data hazards in the pipeline:**
- WB not done in lexicographical order: (what about exceptions?)
  - `div.f $f1, $f2, $f3`
  - `mul.f $f4, $f1, $f1`
  - `add.f $f1, $f5, $f6`
- Why this is not correct? WAR and WAW dependences

**Conclusion:**
- Control unit to handle all these situations seems to be complicated

- Block here if no resource available
- Block here if RAW dependence
- Block here if WAW dependence
- Block here to avoid incorrect bypasses due to WAR dependences