Pipelining

- **Improve performance by increasing instruction throughput**

  Ideal speedup is number of stages in the pipeline. Do we achieve this?

  ![Diagram showing pipelining](See Figure 6.3)

  - What makes it easy
    - all instructions are the same length
    - just a few instruction formats
    - memory operands appear only in loads and stores
  
  - What makes it hard?
    - structural hazards: suppose we had only one memory
    - control hazards: need to worry about branch instructions
    - data hazards: an instruction depends on a previous instruction

  - We’ll build a simple pipeline and look at these issues

  - We’ll talk about modern processors and what really makes it hard:
    - exception handling
    - trying to improve performance with out-of-order execution, etc.
Basic Idea
(See Figure 6.10)

• What do we need to add to actually split the datapath into stages?

Pipelined Datapath
(See Figure 6.12)

Can you find a problem even if there are no dependencies?
What instructions can we execute to manifest the problem?
Corrected Datapath

To preserve write register number for lw instruction

Example: Pipe Stages for lw instruction
Chapter 6

Example: Pipe Stages for \texttt{lw} instruction (See Figure 6.13)

Example: Pipe Stages for \texttt{lw} instruction (See Figure 6.14)
Example: Pipe Stages for `lw` instruction

(See Figure 6.15)
Datapath for all pipe stages for **lw** instruction

(See Figure 6.19)

Example: Pipe Stages for **sw** instruction

(See Figure 6.13)
Example: Pipe Stages for \texttt{sw} instruction \hfill (See Figure 6.13)

![Diagram of pipeline stages for \texttt{sw} instruction]

Example: Pipe Stages for \texttt{sw} instruction \hfill (See Figure 6.16)

![Diagram of pipeline stages for \texttt{sw} instruction]
Example: Pipe Stages for \texttt{sw} instruction

Instruction memory

Address

Register

Write data

Add

Add result

Shift left 2

	exttt{IF/ID}

	exttt{EX/MEM}

	exttt{MEM/WB}

Write back

Address

Read register 1

Read register 2

Read data 1

Read data 2

Write memory

ALU

Zero

16 Sign extend

Read registers

Read data

Write data

Data memory

Address

	exttt{PC}

	exttt{WXYZ}

	exttt{MM}

(See Figure 6.17)
Datapath for all pipe stages for **sw** instruction

Graphically Representing Pipelines

- Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths
Graphically Representing Pipelines

<table>
<thead>
<tr>
<th>Program execution order (in instructions)</th>
<th>Time (in clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $10, $20($1)</td>
<td>CC 1</td>
</tr>
<tr>
<td>sub $11, $2, $3</td>
<td>CC 2</td>
</tr>
<tr>
<td></td>
<td>CC 3</td>
</tr>
<tr>
<td></td>
<td>CC 4</td>
</tr>
<tr>
<td></td>
<td>CC 5</td>
</tr>
<tr>
<td></td>
<td>CC 6</td>
</tr>
</tbody>
</table>

- Traditional multiple-clock-cycle pipeline diagram of two instructions
- Alternatively, single-clock-cycle pipeline diagrams, such as those in slides 6 through 10 and slides 12 through 16, can be used.
  - Shows the state of the entire datapath during a single clock cycle

Single-cycle pipeline diagram for two instructions

(See Figure 6.22)
Single-cycle pipeline diagram for two instructions

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

(See Figure 6.22)

Single-cycle pipeline diagram for two instructions

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

(See Figure 6.23)
Single-cycle pipeline diagram for two instructions

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

(See Figure 6.23)
Chapter 6

Single-cycle pipeline diagram for two instructions

sub $11, $2, $3

Write back

Pipeline Control

(See Figure 6.25)
Pipeline control

- We have 5 stages. What needs to be controlled in each stage?
  - Instruction Fetch and PC Increment
  - Instruction Decode / Register Fetch
  - Execution
  - Memory Stage
  - Write Back

- How would control be handled in an automobile plant?
  - a fancy control center telling everyone what to do?
  - should we use a finite state machine?

Pipeline Control

(See Figure 6.29)

- Pass control signals along just like the data

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution/Address Calculation stage control lines</th>
<th>Memory access stage control lines</th>
<th>stage control lines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reg Dst</td>
<td>ALU Op1</td>
<td>ALU Op0</td>
</tr>
<tr>
<td>R-format</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b=eq</td>
<td>X</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Instruction register

Control

IF/ID

ID/EX

EX/MEM

MEM/WB

(See Figure 6.29)
Datapath with Control

(See Figure 6.30)

Labeled Pipeline Execution with Control

(See Figure 6.31)
Labeled Pipeline Execution with Control

(See Figure 6.31)

**IF:** sub $11, $2, $3  
**ID:** lw $10, 20($1)  
**EX:** before<1>  
**MEM:** before<2>  
**WB:** before<3>

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

(See Figure 6.32)

**IF:** and $12, $4, $5  
**ID:** sub $11, $2, $3  
**EX:** lw $10, 20($1)  
**MEM:** before<1>  
**WB:** before<2>

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

Chapter 6
Labeled Pipeline Execution with Control  (See Figure 6.32)

| IF:  | ID: and $12, $4, $5 | EX: sub $11, $2, $3 | MEM: lw $10, 20($1) | WB: before<1>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $10, 20($1)</td>
<td>sub $11, $2, $3 and $12, $4, $5</td>
<td>or $13, $6, $7</td>
<td>add $14, $8, $9</td>
<td></td>
</tr>
</tbody>
</table>

(See Figure 6.33)

| IF: | ID: or $13, $6, $7 | EX: $12, $4, $5 | MEM: sub $11, $2, $3 | WB: lw $10 ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $10, 20($1)</td>
<td>sub $11, $2, $3 and $12, $4, $5 or $13, $6, $7</td>
<td>add $14, $8, $9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Chapter 6

Labeled Pipeline Execution with Control  (See Figure 6.32)

| IF: or $13, $6, $7 | ID: and $12, $4, $5 | EX: sub $11, $2, $3 | MEM: lw $10, 20($1) | WB: before<1>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $10, 20($1)</td>
<td>sub $11, $2, $3 and $12, $4, $5</td>
<td>or $13, $6, $7</td>
<td>add $14, $8, $9</td>
<td></td>
</tr>
</tbody>
</table>

(See Figure 6.33)

| IF: | ID: or $13, $6, $7 | EX: $12, $4, $5 | MEM: sub $11, $2, $3 | WB: lw $10 ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $10, 20($1)</td>
<td>sub $11, $2, $3 and $12, $4, $5 or $13, $6, $7</td>
<td>add $14, $8, $9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Labeled Pipeline Execution with Control

IF: after<1>
ID: add $14, $8, $9
EX: or $13, $6, $7
MEM: and $12, $4, $6
WB: sub $11...

Clock 6

IF: after<2>
ID: after<1>
EX: add $14, $8, $9
MEM: or $13, $6, $7
WB: and $12...

Clock 7

(See Figure 6.33)
Labeled Pipeline Execution with Control

(See Figure 6.34)

Chapter 6

Labeled Pipeline Execution with Control

(See Figure 6.35)

Chapter 6
Pipelined Dependencies

• Problem with starting next instruction before first is finished
  – dependencies that “go backward in time” are data hazards

<table>
<thead>
<tr>
<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>Value of register $2$:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/–</td>
<td>–20</td>
</tr>
</tbody>
</table>

Program execution order (in instructions):

- sub $2, $1, $3
- and $12, $2, $5
- or $13, $6, $2
- add $14, $2, $2
- sw $15, 100($2)

Software Solution

• Have compiler guarantee no hazards
• Where do we insert the “nops”? 

sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)

• Problem: this really slows us down!
Use temporary results, don’t wait for them to be written
- register file forwarding to handle read/write to same register
- ALU forwarding

### Time (in clock cycles)

<table>
<thead>
<tr>
<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>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10–20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

### Program execution order (in instructions)

- `sub $2, $1, $3`
- `and $12, $1, $5`
- `or $13, $6, $2`
- `add $14, $2, $2`
- `sw $15, 100($2)`

Value of register $2: X X X - 20 X X X X X
Value of EX/MEM: X X X - 20 X X X X X
Value of MEM/WB: X X X X - 20 X X X X X

What if this $2 was $13?
Instruction Sequence with Forwarding

IF: sub $2, $1, $3
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2

EX: before<1>
MEM: before<3>
WB: before<4>

Clock 1

Instruction Sequence with Forwarding

IF: and $4, $2, $5
sub $2, $1, $3
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2

EX: before<1>
MEM: before<3>
WB: before<4>

Clock 2
Instruction Sequence with Forwarding

(See Figure 6.41)

IF: or $4, $4, $2
ID: and $4, $2, $5
EX: sub $2, $1, $3
MEM: before<1>
WB: before<2>

Clock 3

IF: add $9, $4, $2
ID: or $4, $4, $2
EX: and $4, $2, $5
MEM: sub $2, ... WB: before<1>

Clock 4

sub $2, $1, $3
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2

Chapter 6

1998 Morgan Kaufmann Publishers
Modifications by Dr. J
2002 Cal State Univ, Chico
Instruction Sequence with Forwarding
(See Figure 6.42)

IF: after<1>

ID: add $9, $4, $2

EX: or $4, $4, $2

MEM: and $4 ... $2

WB: sub $2 ...

Clock 5

sub $2, $1, $3
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2

Instruction Sequence with Forwarding
(See Figure 6.42)

IF: after<2>

ID: after<1>

EX: add $9, $4, $2

MEM: or $4 ...

WB: and $4 ...

Clock 6

sub $2, $1, $3
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2
Can't always forward

- Load word can still cause a hazard:
  - an instruction tries to read a register following a load instruction that writes to the same register.
  
  ![Diagram showing load word causing a hazard]

- Thus, we need a hazard detection unit to “stall” the load instruction

Stalling

- We can stall the pipeline by keeping an instruction in the same stage

![Diagram showing stalling in the pipeline]
Hazard Detection Unit

- Stall by letting an instruction that won’t write anything go forward

Instruction Sequence with Hazard Detection

IF: lw $2, 20($1)
ID: before<1>
EX: before<2>
MEM: before<3>
WB: before<4>

1w $2, 20($1) and $4, $2, $5
or $4, $4, $2
add $5, $4, $2
Instruction Sequence with Hazard Detection

(See Figure 6.48)

IF: or $4, $2, $2
ID: and $4, $2, $5
EX: bubble
MEM: lw $2 ...
WB: before<1>

Clock 4

lw $2, 20($1)
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2

55

Instruction Sequence with Hazard Detection

(See Figure 6.48)

IF: add $9, $4, $2
ID: or $4, $4, $2
EX: and $4, $2, $5
MEM: bubble
WB: lw $2 ...

Clock 5

lw $2, 20($1)
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2

56
Instruction Sequence with Hazard Detection

(See Figure 6.49)

IF: after<1>  ID: add $9, $4, $2  EX: or $4, $4, $2  MEM: and $4 ...  WB: bubble

Clock 6

IF: after<1>  ID: add $9, $4, $2  EX: or $4, $4, $2  MEM: and $4 ...  WB: bubble

Clock 7
Branch Hazards

When we decide to branch, other instructions are in the pipeline!

Program execution order (in instructions):
- 40 beq $1, $3, 7
- 44 add $12, $2, $5
- 46 or $13, $6, $2
- 52 add $14, $2, $2
- 72 lw $4, 50($7)

Approach #1: Assume/Predict “branch not taken”
- Hence, load instructions sequentially following a branch.
- If branch is taken (incorrect prediction), instructions instructions in the IF, ID, and EX stages must be discarded.
- Need to add hardware for flushing if the branch is taken.

Approach #2: Reduce the delay of branches
- Move the branch decision hardware earlier in the pipeline – from the MEM stage to the ID stage.
- Branch address calculation has to occur in the ID stage!

Note: These two approaches are not mutually exclusive …
Chapter 6

Flushing Instructions

(See Figure 6.51)

Instruction Sequence with Pipelined Branch

<table>
<thead>
<tr>
<th>IF: sub $10, $4, $8</th>
<th>ID: before&lt;1&gt;</th>
<th>EX: before&lt;2&gt;</th>
<th>MEM: before&lt;3&gt;</th>
<th>WB: before&lt;4&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IF: sub $10, $4, $8
ID: before<1>
EX: before<2>
MEM: before<3>
WB: before<4>

Clock 1

| 36 | sub $10, $4, $8 |
| 40 | beq $1, $3, 7 |
| 44 | and $12, $2, $5 |
| 72 | lw $4, 50($7) |
Instruction Sequence with Pipelined Branch

IF: `beq $1, $3, 7`
ID: `sub $10, $4, $8`
EX: before<1>
MEM: before<2>
WB: before<3>

Clock 2

36 sub $10, $4, $8
40 `beq $1, $3, 7`
44 and $12, $2, $5
...
72 lw $4, 50($7)

Instruction Sequence with Pipelined Branch

IF: `and $12, $2, $5`
ID: `beq $1, $3, 7`
EX: `sub $10, $4, $8`
MEM: before<1>
WB: before<2>

Clock 3

36 sub $10, $4, $8
40 `beq $1, $3, 7`
44 and $12, $2, $5
...
72 lw $4, 50($7)

(See Figure 6.52)
Dynamic Branch Prediction

- **Branch prediction** optimizes the two techniques that deal with branch hazards …

- **Branch prediction buffer** or **branch history table**
  - Use a small memory indexed by the lower portion of the address of the branch instruction.
  - This memory contains a bit to indicate whether the branch was recently taken (1) or not (0).
  - Example:
    - Consider a loop branch that branches nine times in a row, then is not taken once.
    - The prediction accuracy of this branch that is taken 90% of the time is only 80% - we will likely predict incorrectly twice, rather than once, when a branch is not taken.
  - Question: At what stage of the pipeline should the branch prediction hardware reside?
Dynamic Branch Prediction

- **Branch prediction buffer**, continued …
  - 2-bit prediction schemes provide better accuracy
    - 2 bits used to encode state information for the finite state machine
    - Question: What is the prediction accuracy for a loop branch that branches nine times in a row, then is not taken once?

Scheduling the Branch Delay Slot

- A **branch delay slot** is the position immediately following a branch, typically containing an instruction that must be flushed if the branch is taken.
Exceptions

- An *exception* or *interrupt* is essentially an unscheduled procedure call.
- The address of the instruction causing the exception is saved in a register (e.g. in MIPS, this is the *exception program counter* or the *EPC*), and the computer jumps to a predefined address to invoke the appropriate routine for that exception.
- Causes of exceptions include:
  - Arithmetic overflow or underflow
  - I/O device requests
  - Invoking an operating system service from a user program
  - Using an undefined instruction
  - Hardware malfunction
- In MIPS, in the case of an arithmetic overflow, we need to
  - flush instructions in the IF, ID, and EX stages
  - transfer control to the *exception routine* at location 0x40000040

(See Figure 6.55)
Exceptions

Overflow exception

sw $16, 50($7)
si: $15, $6, $7
add $1, $2, $1
or $13, . . . , and $12, . . .

Clock 5

Exceptions

sw $25, 1000($0)
bubble (nop)
bubble
bubble
or $13, . . .

Clock 6

(See Figure 6.56)
Superscalar Pipelining

- **Superpipelining** – pipelined processors that have longer pipelines
- **Superscalar** – replicate internal components of a computer so it can launch/issue several multiple instructions in every pipelining stage

Improving Performance

- Try and avoid stalls! For example, *reorder* these instructions:
  
  \[
  \begin{align*}
  \text{lw} & \quad $t0, \ 0($t1) \\
  \text{lw} & \quad $t2, \ 4($t1) \\
  \text{sw} & \quad $t2, \ 0($t1) \\
  \text{sw} & \quad $t0, \ 4($t1)
  \end{align*}
  \]

- Can reordering be implemented on software?
- Can reordering be implemented on hardware?
Dynamic Pipeline Scheduling

- The hardware performs the “scheduling”
  - hardware tries to find instructions to execute
  - out of order execution is possible
  - speculative execution and dynamic branch prediction

Dynamic Pipeline Scheduling

- All modern processors are very complicated
  - DEC Alpha 21264: 9 stage pipeline, 6 instruction issue
  - PowerPC and Pentium: branch history table
  - Compiler technology important

- This class has given you the background you need to learn more
Example: Single-cycle vs. multicycle vs. pipelined control

- **Problem:** Compare performance for single-cycle, multicycle, and pipelined control using the instruction mix for gcc from Figure 4.54 (frequency of the MIPS instructions for two programs, gcc and spice) on page 311.
  
  - Assume the operation times for the major functional units are 2
    ns for memory access, 2 ns for ALU operations, and 1 ns for
    register file read or write.
  
  - For pipelined execution, assume that half of the load instructions
    are immediately followed by an instruction that uses the result,
    that the branch delay on misprediction is 1 clock cycle, and that
    one-quarter of the branches are mispredicted.
  
  - Assume that jumps always pay 1 full clock cycle of delay, so their
    average time is 2 clock cycles.

Example: Single-cycle vs. multicycle vs. pipelined control

- **Answer:** From Figure 4.54 (frequency of the MIPS instructions for two programs, gcc and spice) on page 311, assume that the instruction mix for gcc is

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Frequency</th>
<th>Instruction</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>loads</td>
<td>23%</td>
<td>branches</td>
<td>19%</td>
</tr>
<tr>
<td>stores</td>
<td>13%</td>
<td>jumps</td>
<td>2%</td>
</tr>
<tr>
<td>R-format</td>
<td>43%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Recall that

\[
\text{CPU exec time} = \frac{\text{Instruction count}}{\text{CPI}} \times \text{Clock cycle time}
\]
Example: Single-cycle vs. multicycle vs. pipelined control

- Answer (single-cycle control):

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>6 ns</td>
</tr>
<tr>
<td>load word</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>8 ns</td>
</tr>
<tr>
<td>store word</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td>7 ns</td>
</tr>
<tr>
<td>branch</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>5 ns</td>
</tr>
<tr>
<td>jump</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 ns</td>
</tr>
</tbody>
</table>

- Single clock for all instructions
  - cycle time = max (6,8,7,5,2) = 8.00 ns
- Variable clock
  - cycle time = 6\%43\% + 8\%23\% + 7\%13\% + 5\%19\% + 2\%2\% = 6.32 ns

Example: Single-cycle vs. multicycle vs. pipelined control

- Answer (multicycle control):

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>No. of states</th>
</tr>
</thead>
<tbody>
<tr>
<td>loads</td>
<td>5</td>
</tr>
<tr>
<td>R-format</td>
<td>4</td>
</tr>
<tr>
<td>stores</td>
<td>4</td>
</tr>
<tr>
<td>branches</td>
<td>3</td>
</tr>
<tr>
<td>jumps</td>
<td>3</td>
</tr>
</tbody>
</table>

- Then, assuming each state requires 1 clock cycle,
  - cycle time = 5\%23\% + 4\%43\% + 4\%13\% + 3\%19\% + 3\%2\% = 4.02 ns
Example: Single-cycle vs. multicycle vs. pipelined control

• Answer (pipelined control):
  – Loads take 1 clock cycle w/o dependency and 2 w/ dependency
    • Average clock cycles per load = 1.5
  – Correctly predicted branches take 1 clock cycle and 2 otherwise
    • Average clock cycles per branch = 1\%75\% + 2\%25\% = 1.25
  – Stores and R-format instructions
    • CPI = 1
  – Jump instructions
    • CPI = 2
  – Therefore,
    • CPI = 1\%23\% + 1.25\%19\% + 1\%13\% + 1\%43\% + 2\%2\% = 1.1825
    • Cycle time = 1.1825\% divide by 2 = 2.365 ns

Example: Single-cycle vs. multicycle vs. pipelined control

• Answer:

<table>
<thead>
<tr>
<th>Control type</th>
<th>Cycle time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>single-cycle, single clock</td>
<td>8.00</td>
</tr>
<tr>
<td>single-cycle, variable clock</td>
<td>6.32</td>
</tr>
<tr>
<td>multicycle</td>
<td>4.02</td>
</tr>
<tr>
<td>pipelined</td>
<td>2.37</td>
</tr>
</tbody>
</table>

– So, for pipelined vs multicycle control,
  • pipelined is 4.02\% divide by 2.37 = 1.69 times faster than multicycle control

– So, for pipelined vs single-cycle control,
  • pipelined is 6.32\% divide by 2.37 = 2.67 times faster than variable clock
  • pipelined is 8.00\% divide by 2.37 = 3.38 times faster than single clock