CS 250
VLSI System Design
Lecture 10 – Design Verification

2010-10-11
John Wawrzynek and Krste Asanovic
with John Lazzaro

TA: Yunsup Lee

www-inst.eecs.berkeley.edu/~cs250/
IBM Power 4
174 Million Transistors
A complex design ...
First silicon booted AIX & Linux, on a 16-die system.
96% of all bugs were caught before first tape-out.

How ???

UC Regents Fall 2010 © UCB
Three main components ...

(1) Specify chip behavior at the RTL level, and comprehensively simulate it.

(2) Use formal verification to show equivalence between Verilog RTL and circuit schematic RTL.

(3) Technology layer: do the electrons implement the RTL, at speed and power?

Today, we focus on (1).
Lecture Focus: Functional Design Test

The processor design correctly executes programs written in the Instruction Set Architecture.

“Correct” == meets the “Architect’s Contract”

testing goal

Not manufacturing tests ...

Architect’s “Contract with the Programmer”

To the program, it appears that instructions execute in the correct order defined by the ISA.

As each instruction completes, the architected machine state appears to the program to obey the ISA.

What the machine actually does is up to the hardware designers, as long as the contract is kept.
Three models (at least) to cross-check.

- The “contract” specification
  “The answer” (correct, we hope). Simulates the ISA model in C. Fast.
  Better: two models coded independently.

- The Verilog RTL model
  Logical semantics of the Verilog model
  we will use to create gates. Runs on a software simulator or FPGA hardware.

- Chip-level schematic RTL
  Catch synthesis bugs. Formally verify netlist against Verilog RTL. Also used for timing and power.

Where do bugs come from?
Where bugs come from (a partial list) ...

- **The contract is wrong.**
  You understand the contract, create a design that correctly implements it, write correct Verilog for the design ...

- **The contract is misread.**
  Your design is a correct implementation of what you think the contract means ... but you misunderstand the contract.

- **Conceptual error in design.**
  You understand the contract, but devise an incorrect implementation of it ...

- **Verilog coding errors.**
  You express your correct design idea in Verilog .. with incorrect Verilog semantics.

Verilog: name misspellings, latch implication, combinational loops.

CS 250 L10: Design Verification
Four Types of Testing
how it works

Assemble the complete processor.

Execute test program suite on the processor.

Check results.

Checks contract model against Verilog RTL. Test suite runs the gamut from “1-line programs” to “boot the OS”.

Top-down testing
- complete processor testing

Bottom-up testing

Big Bang: Complete Processor Testing
Methodical Approach: Unit Testing

how it works

Remove a block from the design.

Test it in isolation against specification.

Requires writing a bug-free “contract model” for the unit.
Climbing the Hierarchy: Multi-unit Testing

**how it works**

Remove connected blocks from design.

Test in isolation against specification.

Choice of partition determines if the test moves the project forward.
Processor Testing with Self-Checking Units

how it works

Add self-checking to units

Perform complete processor testing

Self-checks are unit tests built into CPU, that generate the “right answer” on the fly. Slower to simulate.
Testing: Verification vs. Diagnostics

- **Verification:**
  A yes/no answer to the question “Does the processor have one more bug?”

- **Diagnostics:**
  Clues to help find and fix the bug.

Diagnosis of bugs found during “complete processor” testing is hard ...
“CPU program” diagnosis is tricky ...

Observation: On a buggy CPU model, the correctness of every executed instruction is suspect.

Consequence: One needs to verify the correctness of instructions that surround the suspected buggy instruction.

Depends on (1) number of “instructions in flight” in the machine, and (2) lifetime of non-architected state (may be “indefinite”).
State observability and controllability

- **Observability:**
  Does my model expose the state I need to diagnose the bug?

- **Controllability:**
  Does my model support changing the state value I need to change to diagnose the bug?

Support != “yes, just rewrite the model code”!
Writing a Test Plan
The testing timeline ...

Plan in advance what tests to do when ...

- Top-down testing
- Processor testing complete
- Processor testing with self-checks
- Multi-unit testing
- Unit testing
- Bottom-up testing

Epoch 1: Processor assembly complete
Epoch 2: Correctly executes single instructions
Epoch 3: Correctly executes short programs
Epoch 4: Time
An example test plan ...
Unit Testing
Combinational Unit Testing: 3-bit Adder

Number of input bits? 7

Total number of possible input values?

\[2^7 = 128\]

Just test them all ...

Apply “test vectors” 0,1,2 ... 127 to inputs.

100% input space “coverage”

“Exhaustive testing”
Combinational Unit Testing: 32-bit Adder

Number of input bits? 65

Total number of possible input values?

\[ 2^{65} = 3.689e+19 \]

Just test them all?

Exhaustive testing does not “scale”.

“Combinatorial explosion!”
Test Approach 1: Random Vectors

**how it works**

Apply random $A$, $B$, $C_{in}$ to adder.

Check $Sum$, $C_{out}$.

**When to stop testing?  Bug curve.**

**How?  Use $random$ to set inputs to the testbench.**

**Bug Rate**

- Bugs found per minute of testing
Test Approach 2: Directed Vectors

How it works:
Hand-craft test vectors to cover "corner cases"

A == B == Cin == 0

"Black-box": Corner cases based on functional properties.

"Clear-box": Corner cases based on unit internal structure.
State Machine Testing

CPU design examples
DRAM controller state machines
Cache control state machines
Branch prediction state machines
Testing State Machines: Break Feedback

Isolate “Next State” logic.
Test as a combinational unit.

Easier with certain Verilog coding styles ...
Testing State Machines: Arc Coverage

Force machine into each state. Test behavior of each arc.

Intractable for state machines with high edge density...
Regression Testing

Or, how to find the last bug ...
Writing “complete CPU” test programs

Top-down testing

Single instructions with directed-random field values.

Epoch 1
processor testing with self-checks
processor assembly complete

Epoch 2
processor testing with self-checks
correctly executes single instructions

Epoch 3
processor testing with self-checks
correctly executes short programs

Epoch 4
complete processor testing

Bottom-up testing

White-box “Instructions-in-flight” sized programs that stress design.

Tests that stress long-lived non-architected state.

Regression testing: re-run subsets of the test library, and then the entire library, after a fix.
Pipelining Basics
Starting Point: Performance Equation

Rationale:
Every additional instruction you execute takes time.

Rationale:
By shortening the period for each cycle, we shorten execution time.

Seconds
Program = Instructions
Program

Cycles
Instruction

Seconds
Cycle

CPI: The average number of clock Cycles Per Instruction For the Program

Different programs have different CPIs, for a variety of reasons.
Consider machine with a data cache ...

A program’s load instructions “stride” through every memory address.

The cache never “hits”, so every load goes to DRAM (100x slower than loads that go to cache).

Thus, the average number of cycles for load instructions is higher for this program.

Thus, the average number of cycles for all instructions is higher for this program.

Thus, program takes longer to run!
Starting Point: Single-cycle processor

Challenge: Speed up clock while keeping CPI == 1

\[
\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \quad \frac{\text{Cycles}}{\text{Instruction}} \quad \frac{\text{Seconds}}{\text{Cycle}}
\]

CPI == 1
This is good.

Slow.
This is bad.

32
A
L
U
32
32
op
Ext

Ext

Data Memory
Addr
Din
WE
MemToReg

RegFile
rs1
rs2
ws
rd1
rd2
WE

Dout

Data Memory
WE

Din
Addr
Mem
Instr

Instr
Mem

PC

D
Q

0x4

PC

D
Q

Seconds

Instructions

Cycles

Seconds

Program

Program

Instruction

Cycle

33
Observation: Logic idle most of cycle

For most of cycle, ALU is either “waiting” for its inputs, or “holding” its output

Ideal: a CPU architecture where each part is always “working”
Inspiration: Automobile assembly line

Assembly line moves on a steady clock. Each station does the same task on each car.

The clock

Merge station

Bolting station

Car body shell

Car chassis
Inspiration: Automobile assembly line

Simpler station tasks → more cars per hour. Simple tasks take less time, clock is faster.
Inspiration: Automobile assembly line

Line speed limited by slowest task. Most efficient if all tasks take same time to do.
Inspiration: Automobile assembly line

Simpler tasks, complex car → long line!

These lines go 24 x 7, and rarely shut down.
Key analogy: The instruction is the car

Pipeline Stage #1: Instruction Fetch

Stage #2

Stage #3

Stage #4

Stage #5

Controls hardware in stage 2

Controls hardware in stage 3

Controls hardware in stage 4

Controls hardware in stage 5

"Data-stationary control"
Example: Decode & Register Fetch stage

Pipeline Stage #1: Instr Fetch
- SUB R10, R9, R8

Stage #2: Decode & Reg Fetch
- OR R7, R6, R5

Stage #3
- ADD R4, R3, R2

A sample program
- ADD R4, R3, R2
- OR R7, R6, R5
- SUB R10, R9, R8

R’s chosen so that instructions are independent - like cars on the line.
Performance Equation and Pipelining

\[
\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}
\]

**Instr Fetch**

**Decode & Reg Fetch**

**Stage #3**

CPI == 1

Once pipe is fill, one instruction completes per cycle

Clock period is shorter
Less work to do in each cycle

To get shortest clock period, balance the work to do in each pipeline stage.

To get shortest clock period, balance the work to do in each pipeline stage.
Hazards: An instruction is not a car ...

... wrong value of R4 fetched from RegFile, contract with programmer broken! **Oops!**

New sample program

```
ADD R4, R3, R2
OR R5, R4, R2
```

An example of a “hazard” -- we must (1) detect and (2) resolve all hazards to make a CPU that matches ISA
Performance Equation and Hazards

Seconds Program = Instructions Program / Cycles Instruction / Seconds Cycle

Instr Fetch → Decode & Reg Fetch → Stage #3

Some ways to cope with hazards makes CPI > 1 “stalling pipeline”

Added logic to detect and resolve hazards increases clock period
A (simplified) 5-stage pipelined CPU

1. \textbf{"IF" Stage} 
   - Instr Fetch

2. \textbf{"ID/RF" Stage} 
   - Decode & Reg Fetch

3. \textbf{"EX" Stage} 
   - Execution

4. \textbf{"MEM" Stage} 
   - Memory

5. \textbf{WB} 
   - Write Back

- IR
- PC
- D
- Q
- +
- 0x4
- Instr Mem
- Addr Data
- Mux, Logic
- RegFile
- rs1
- rs2
- rd1
- ws
- wd
- WE
- M
- 32
- A
- B
- Ext
- Dout
- Addr
- Data Memory
- Addr
- Dout
- WE
- MemToReg
- R
- 32
- A
- 0
- 32

- IR
- WE, MemToReg

Visualizing Pipelines
Pipeline Representation #1: Timeline

Sample Program

I1: ADD R4,R3,R2
I2: AND R6,R5,R4
I3: SUB R1,R9,R8
I4: XOR R3,R2,R1
I5: OR R7,R6,R5

Time: t1 t2 t3 t4 t5 t6 t7 t8
Inst
I1: IF ID EX MEM WB
I2: IF ID EX MEM WB
I3: IF ID EX MEM WB
I4: IF ID EX MEM WB
I5: IF ID EX MEM WB
I6: IF ID EX MEM WB

Pipeline is “full”

Good for visualizing pipeline fills.
**Representation #2: Resource Usage**

Good for visualizing pipeline stalls.

**Sample Program**

<table>
<thead>
<tr>
<th>I1</th>
<th>ADD R4,R3,R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>AND R6,R5,R4</td>
</tr>
<tr>
<td>I3</td>
<td>SUB R1,R9,R8</td>
</tr>
<tr>
<td>I4</td>
<td>XOR R3,R2,R1</td>
</tr>
<tr>
<td>I5</td>
<td>OR R7,R6,R5</td>
</tr>
</tbody>
</table>

**Time:**

<table>
<thead>
<tr>
<th>Stage</th>
<th>I1</th>
<th>I2</th>
<th>I3</th>
<th>I4</th>
<th>I5</th>
<th>I6</th>
<th>I7</th>
<th>I8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</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>
</tr>
<tr>
<td>ID</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td></td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>I7</td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td>I1</td>
<td>I2</td>
<td></td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
</tr>
<tr>
<td>MEM</td>
<td>I1</td>
<td></td>
<td></td>
<td></td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
</tr>
<tr>
<td>WB</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>
</tr>
</tbody>
</table>

**Pipeline is “full”**
Data and Control Hazards
Data Hazards: 3 Types (RAW, WAR, WAW)

Several pipeline stages read or write the same data location in an incompatible way.

Read After Write (RAW) hazards.
Instruction I2 expects to read a data value written by an earlier instruction, but I2 executes “too early” and reads the wrong copy of the data.

Note “data value”, not “register”. Data hazards are possible for any architected state (such as main memory). In practice, main memory hazard avoidance is the job of the memory system.
Recall: RAW example

Sample program

ADD R4, R3, R2
OR R5, R4, R2

... wrong value of R4 fetched from RegFile, contract with programmer broken! Oops!

This is what we mean when we say Read After Write (RAW) Hazard
Control Hazards: A taken branch/jump

Sample Program (ISA w/o branch delay slot)

I1: BEQ R4, R3, 25
I2: AND R6, R5, R4
I3: SUB R1, R9, R8

Note: with branch delay slot, I2 MUST complete, I3 MUST NOT complete.

Time: t1 t2 t3 t4 t5 t6 t7 t8
Inst
I1: IF ID EX MEM WB
I2: IF ID
I3: IF
I4: 
I5: 
I6: 

EX stage computes if branch is taken

If branch is taken, these instructions MUST NOT complete!
Hazard Resolution Tools
The Hazard Resolution Toolkit

- **Stall** earlier instructions in pipeline.
- **Forward** results computed in later pipeline stages to earlier stages.
- **Add** new hardware or **rearrange** hardware design to eliminate hazard.
- **Change ISA** to eliminate hazard.
- **Kill** earlier instructions in pipeline.
- Make hardware handle **concurrent requests** to eliminate hazard.
Resolving a RAW hazard by stalling

**Stage #1**
- Instr Fetch

**Stage #2**
- Decode & Reg Fetch

**Stage #3**
- ADD R4, R3, R2

### Sample program
ADD R4, R3, R2
OR R5, R4, R2

### New datapath hardware
1. Mux into IR 2/3 to feed in NOP.
2. Write enable on PC and IR 1/2

**Keep executing OR instruction until R4 is ready. Until then, send NOPS to IR 2/3.**

**Let ADD proceed to WB stage, so that R4 is written to RegFile.**

**Freeze PC and IR until stall is over.**
The Hazard Resolution Toolkit

- Stall earlier instructions in pipeline.
- Forward results computed in later pipeline stages to earlier stages.
- Add new hardware or rearrange hardware design to eliminate hazard.
- Change ISA to eliminate hazard.
- Kill earlier instructions in pipeline.
- Make hardware handle concurrent requests to eliminate hazard.
Resolving a RAW hazard by **forwarding**

Sample program

ADD R4, R3, R2
OR R5, R4, R2

**Just forward it back!**

Unlike stalling, does not change CPI. May hurt cycle time.
The Hazard Resolution Toolkit

- **Stall** earlier instructions in pipeline.
- **Forward** results computed in later pipeline stages to earlier stages.
- **Add** new hardware or **rearrange** hardware design to eliminate hazard.
- **Change ISA** to eliminate hazard.
- **Kill** earlier instructions in pipeline.
- **Make hardware handle concurrent requests** to eliminate hazard.
Control Hazards: Fix with more hardware

Sample Program
(ISA w/o branch delay slot)

I1: BEQ R4,R3,25
I2: AND R6,R5,R4
I3: SUB R1,R9,R8

If we add hardware, can we move it here?

If branch is taken, these instructions MUST NOT complete!

EX stage computes if branch is taken

Time: t1 t2 t3 t4 t5 t6 t7 t8
Inst
I1: IF ID EX MEM WB
I2: IF ID
I3: IF
I4: I5: I6:
Resolving control hazard with hardware

Stage #1
Instr Fetch

Stage #2
Decode & Reg Fetch

Stage #3

To branch control logic

IR
RegFile
rd1
rd2
WE
wd
rs1
rs2
ws
rd1
rs2
rd2
WD
WE
Ext

Instr Mem
PC
0x4
Addr
Data

Instr Fetch
Stage #1

 Decode & Reg Fetch
Stage #2

To branch control logic

Stage #3

Control Hazards: After more hardware

If we change ISA, can we always let I2 complete ("branch delay slot") and eliminate the control hazard.

Sample Program (ISA w/o branch delay slot)
I1: BEQ R4,R3,25
I2: AND R6,R5,R4
I3: SUB R1,R9,R8

If branch is taken, this instruction MUST NOT complete!
The Hazard Resolution Toolkit

- **Stall** earlier instructions in pipeline.
- **Forward** results computed in later pipeline stages to earlier stages.
- **Add** new hardware or **rearrange** hardware design to eliminate hazard.
- **Change ISA** to eliminate hazard.
- **Kill** earlier instructions in pipeline.
- Make hardware handle **concurrent requests** to eliminate hazard.
Resolve control hazard by killing instr

Sample program (no delay slot)

J 200
OR R5, R4, R2

Detect J instruction, mux a NOP into IR 1/2

This hurts CPI.

One can do better.

Compute new PC using hardware not shown ...
Hazard Diagnosis

Assume MIPS ISA in examples to follow ...
Data Hazards: Read After Write

Read After Write (RAW) hazards. Instruction I2 expects to read a data value written by an earlier instruction, but I2 executes “too early” and reads the wrong copy of the data.

Classic solution: use forwarding heavily, fall back on stalling when forwarding won’t work or slows down the critical path too much.
Full bypass network ...

---

**Mux, Logic**

**RegFile**
- rs1
- rs2
- ws
- rd1
- rd2
- wd
- WE

**Ext**

**IR**

**RD1**

**RD2**

**WE**

**WD**

**RS1**

**RS2**

**WS**

**MemToReg**

**Data Memory**
- Addr
- Din
- WE
- Dout
- MemToReg

**Dout**

**A**

**ALU**

**B**

**Ext**

**M**

**M**

**ID (Decode)**

**EX**

**MEM**

**WB**

**From WB**

---

65
Common bug: Multiple forwards ...

Which do we forward from?

ADD R4, R3, R2
OR R2, R3, R1
AND R2, R2, R1

Mux, Logic

RegFile
rs1
rs2
ws
rd1
rd2
wd
WE

EX

MEM

WB

DATA MEMORY

WE, MemToReg

R

MemToReg

RegToReg

Ext

From
WB

From
IR

IR

IR

IR

RegToReg

Mux, Logic

Add

Data Memory

Addr

Din

WE

MemToReg

R

WeReg

ADD R4, R3, R2
OR R2, R3, R1
AND R2, R2, R1

Which do we forward from?
Common bug: Multiple forwards II ...

Which do we forward from?

ADD R4, R0, R2
OR R0, R3, R1
AND R0, R2, R1

ID (Decode)
EX
MEM
WB

Mux, Logic
RegFile
rd1
rd2
rs1
rs2
ws
wd
WE

IR
Ext

IR
RegFile
Din
Addr
MemToReg

R

Data Memory
Y
A
M

B

Dout
Data Memory
WE, MemToReg

From
WB

Which do we forward from?
## LW and Hazards

<table>
<thead>
<tr>
<th>Type</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>addu, subu, addiu</td>
</tr>
<tr>
<td>logical</td>
<td>and, andi, or, ori, xor, xori, lui</td>
</tr>
<tr>
<td>shift</td>
<td>sll, sra, srl</td>
</tr>
<tr>
<td>compare</td>
<td>slt, slti, sltu, sltui</td>
</tr>
<tr>
<td>control</td>
<td>beq, bne, bgez, bltz, j, jr, jal</td>
</tr>
<tr>
<td>data transfer</td>
<td>lw, sw</td>
</tr>
<tr>
<td>Other:</td>
<td>break</td>
</tr>
</tbody>
</table>

No load “delay slot”
Questions about LW and forwarding

Do we need to stall?

ADDIU R1 R1 24
OR R3,R3,R2 LW R1 128(R29)

ID (Decode) EX MEM WB

RegFile
Mux, Logic

Ext

IR
RegFile
rd1
rd2
ws
rd2
ws
rd2
WE

MemToReg

Data Memory

R
Questions about LW and forwarding

**ADDIU** R1 R1 24  
**LW** R1 128(R29) **OR** R1, R3, R1

Do we need to stall?

**ID** (Decode)

**EX**

**MEM**

**WB**

**Mux, Logic**

**RegFile**

**MemToReg**

**Data Memory**
Branches and Hazards

<table>
<thead>
<tr>
<th>Type</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>addu, subu, addiu</td>
</tr>
<tr>
<td>logical</td>
<td>and, andi, or, ori, xor, xori, lui</td>
</tr>
<tr>
<td>shift</td>
<td>sll, sra, srl</td>
</tr>
<tr>
<td>compare</td>
<td>slt, slti, sltu, sltui</td>
</tr>
<tr>
<td>control</td>
<td>beq, bne, bgez, bltz, j, jr, jal</td>
</tr>
<tr>
<td>data transfer</td>
<td>lw, sw</td>
</tr>
<tr>
<td>Other:</td>
<td>break</td>
</tr>
</tbody>
</table>
Recall: Control hazard and hardware

Stage #1:
- Instr Fetch

Stage #2:
- Decode & Reg Fetch

Stage #3

To branch control logic

IR

RegFile

rd1

rd2

WE

wd

rs1

rs2

ws

0x4

Instr Fetch

Stage #1

Stage #2

Stage #3

==

IR

IR

IR

A

M

B

Ext

PC

Mem

Addr

Data

Instr

Mem

PC

D

Q

M

Ext

RegFile

rs1

rs2

wd

WE

drd1

drd2

0x4

+
Recall: After more hardware, change ISA

If we change ISA, can we always let I2 complete ("branch delay slot") and eliminate the control hazard.

Sample Program (ISA w/o branch delay slot)

I1: BEQ R4,R3,25
I2: AND R6,R5,R4
I3: SUB R1,R9,R8

If branch is taken, this instruction MUST NOT complete!
Question about branch and forwards:

BEQ R1 R3 label

Will this work as shown?

OR R3, R3, R1

ID (Decode)

EX

MEM

WB

To branch control logic

Mux, Logic

RegFile

rs1

rs2

ws

wd

WE

rd1

rd2

==

Dout

Data Memory

WE

Din

Addr

MemToReg

R

op

IR

32

32

32

32

32

op

B

A

Y

M

M

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

IR

I
Lessons learned

- Pipelining is hard
- Study every instruction
- Write test code in advance
- Think about interactions ...
Control Implementation
Recall: What is single cycle control?

Combinational Logic (Only Gates, No Flip Flops) Just specify logic functions!

Instruction Memory

RegFile

Data Memory
In pipelines, all IR registers are used.

Equal

RegDest
RegWr
ExtOp
MemToReg

Combinational Logic (Only Gates, No Flip Flops) (add extra state outside!)

A “conceptual” design -- for shortest critical path, IR registers may hold decoded info, not the complete 32-bit instruction.
Advanced Pipelining
5 Stage Pipeline: A point of departure

At best, the 5-stage pipeline executes one instruction per clock, with a clock period determined by the slowest stage.

Processor has no “multi-cycle” instructions (ex: multiply with an accumulate register)
Superpipelining: Add more stages

Goal: Reduce critical path by adding more pipeline stages.

Example: 8-stage ARM XScale: extra IF, ID, data cache stages.

Difficulties: Added penalties for load delays and branch misses.

Ultimate Limiter: As logic delay goes to 0, FF clk-to-Q and setup.

Also, power!
Superscalar: Multiple issues per cycle

Goal: Improve CPI by issuing several instructions per cycle.

Example: CPU with floating point ALUs: issue 1 FP + 1 integer instruction per cycle.

Difficulties: Load and branch delays affect more instructions.

Ultimate Limiter: Programs may be a poor match to issue rules.
Throughput and multiple threads

Goal: Use multiple CPUs (real and virtual) to improve (1) throughput of machines that run many programs (2) execution time of multi-threaded programs.

Example: Sun Niagara (8 SPARCs on one chip).

Difficulties: Gaining full advantage requires rewriting applications, OS, libraries.

Ultimate limiter: Amdahl’s law, memory system performance.
Superpipelining
5 Stage

Note: Some stages now overlap, some instructions take extra stages.

IF
-> IR
ID+RF
-> IR
EX
-> IR
MEM
-> IR
WB

IF now takes 2 stages (pipelined I-cache)

ID and RF each get a stage.

ALU split over 3 stages

MEM takes 2 stages (pipelined D-cache)

8 Stage

UC Regents Fall 2010 © UCB
Superpipelining techniques ...

- Split **ALU** and **decode** logic over several pipeline stages.

- **Pipeline memory**: Use more banks of smaller arrays, add pipeline stages between decoders, muxes.

- Remove “rarely-used” **forwarding networks** that are on critical path. **Creates stalls, affects CPI.**

- Pipeline the wires of frequently used **forwarding networks**.

Also: Clocking tricks (example: negedge register file)
Add pipeline stages, reduce clock period

Q. Could adding pipeline stages hurt the CPI for an application?
A. Yes, due to these problems:

<table>
<thead>
<tr>
<th>CPI Problem</th>
<th>Possible Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taken branches cause longer stalls</td>
<td>Branch prediction, loop unrolling</td>
</tr>
<tr>
<td>Cache misses take more clock cycles</td>
<td>Larger caches, add prefetch opcodes to ISA</td>
</tr>
</tbody>
</table>
Recall: Control hazards ...

We avoid stalling by (1) adding a branch delay slot, and (2) adding comparator to ID stage.
If we add more early stages, we must stall.

Sample Program
(ISA w/o branch delay slot)

I1: BEQ R4, R3, 25
I2: AND R6, R5, R4
I3: SUB R1, R9, R8

Time: t1 t2 t3 t4 t5 t6 t7 t8
Inst
  I1: IF ID EX MEM WB
  I2: IF ID
  I3:
  I4:
  I5:
  I6:

EX stage computes if branch is taken
If branch is taken, these instructions MUST NOT complete!
Solution: Branch prediction...

We update the PC based on the outputs of the branch predictor. If it is perfect, pipe stays full!

Dynamic Predictors: a cache of branch history

Time: t1  t2  t3  t4  t5  t6  t7  t8
Inst
I1:  IF  ID  EX  MEM  WB
I2:  IF  ID
I3:  IF
I4:  
I5:  
I6:  

If we predicted incorrectly, these instructions MUST NOT complete!
Superscalar

Basic Idea: Improve CPI by issuing several instructions per cycle.
Sustaining Dual Instr Issues (no forwarding)

ADD R8, R0, R0
ADD R11, R0, R0
ADD R27, R26, R25
ADD R30, R29, R28
ADD R21, R20, R19
ADD R24, R23, R22
ADD R15, R14, R13
ADD R18, R17, R16
ADD R9, R8, R7
ADD R12, R11, R10

It’s rarely this good...
We add 12 forwarding buses (not shown). (6 to each ID from stages of both pipes).

Worst-Case Instruction Issue

ADD R8, R0, R0
ADD R9, R8, R0
ADD R10, R9, R0
ADD R11, R10, R0

Dependencies force “serialization”
Multi-Threading
Recall: Bypass network prevents stalls

Instead of bypass: Interleave threads on the pipeline to prevent stalls...
**Introducted in 1964 by Seymour Cray**

*Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe*

T1: LW r1, 0(r2)  
T2: ADD r7, r1, r4  
T3: XORI r5, r4, #12  
T4: SW 0(r7), r5  
T1: LW r5, 12(r1)

```
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>FD</td>
<td>DX</td>
<td>MW</td>
</tr>
<tr>
<td>F</td>
<td>FD</td>
<td>DX</td>
<td>MW</td>
</tr>
<tr>
<td>F</td>
<td>FD</td>
<td>DX</td>
<td>MW</td>
</tr>
<tr>
<td>F</td>
<td>FD</td>
<td>DX</td>
<td>MW</td>
</tr>
</tbody>
</table>
```

Last instruction in a thread always completes writeback before next instruction in same thread reads regfile

```
+1
2
Thread
```

 forty CPUs, each run at 1/4 clock

Many variants ...
Upcoming: Project Proposals

Wed Oct 13 All Initial project proposal presentations.