Last Time: Goal #1, an R-format CPU

Syntax: ADD $8 $9 $10  Semantics: $8 = $9 + $10

| opcode | rs | rt | rd | shamt | funct |

Sample program:
ADD $8 $9 $10
SUB $4 $8 $3
AND $9 $8 $4
...

How registers get their initial values are not of concern to us right now.

No branches or jumps: machine only runs straight line code.

No loads or stores: machine has no use for data memory, only instruction memory.
Last Time: An R-format CPU design

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

Decode fields to get: ADD $8 $9 $10
Last Time: Goal #2, I-format instructions

Syntax: ORI $8 $9 64  Semantics: $8 = $9 | 64

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
</table>

In this example, $9 is rs and $8 is rt.

16-bit immediate extended to 32 bits.

Zero-extend: 0x8000  $\Rightarrow$  0x00008000

Sign-extend: 0x8000  $\Rightarrow$  0xFFFFF8000

Some MIPS instructions zero-extend immediate field, other instructions sign-extend.
Last Time: The merged data path ...

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

RegFile

- RS1
- RS2
- WS
- WD
- RD1
- RD2

RegDest

- 32
- 5
- 5
- 5

ALUctr

- 32

Ext

- ExtOp
- ALUsrc

ALU

- 32

op

- rs
- rt

immediate

- 32
Today’s Lecture: Single-Cycle Wrap-up

* Design stand-alone machines for other major classes of instructions: branches, load/store.

* Implementing control structures for the single-cycle datapath.

* Very Long Instruction Words (VLIW): Doing more work in a single cycle.

And also, Design Notebook for Lab 2 ...
Memory Instructions
**Loads, Stores, and Data Memory ...**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
</table>

**Syntax:** `LW $1, 32($2)`  
**Action:** $1 = M[$2 + 32]

**Syntax:** `SW $3, 12($4)`  
**Action:** M[$4 + 12] = $3

**Zero-extend or sign-extend immediate field?**

**Reads are combinational:** Put a stable address on `Addr`, a short time later `Dout` is ready.

**Writes are clocked:** If `WE` is high, memory `Addr` captures `Din` on positive edge of clock.

**Note:** Not a realistic main memory (DRAM) model ...
Adding data memory to the data path

Load delay slot CPU, or not?

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax: \( \text{LW} \ \$1, \ 32(\$2) \)
Action: \( \$1 = M[\$2 + 32] \)

Syntax: \( \text{SW} \ \$3, \ 12(\$4) \)
Action: \( M[\$4 + 12] = \$3 \)
Branch Instructions
Conditional Branches in MIPS ...

Syntax: BEQ $1, $2, 12

Action: If ($1 != $2), PC = PC + 4

Action: If ($1 == $2), PC = PC + 4 + 48

Immediate field codes # words, not # bytes. Why is this encoding a good idea?

Zero-extend or sign-extend immediate field? Why is this extension method a good idea?
Adding branch testing to the data path

Syntax: BEQ $1, $2, 12
Action: If ($1 != $2), PC = PC + 4
Action: If ($1 == $2), PC = PC + 4 + 48
Recall: Straight-line Instruction Fetch

Fetching straight-line MIPS instructions requires a machine that generates this timing diagram:

PC == Program Counter, points to next instruction.
Recall: Straight-line Instruction Fetch

Syntax: \texttt{BEQ \$1, \$2, 12}

Action: If ($\$1 \neq \$2$), $\text{PC} = \text{PC} + 4$

Action: If ($\$1 == \$2$), $\text{PC} = \text{PC} + 4 + 48$

How do we add this behavior?
Design: Instruction Fetch with Branch

Syntax: BEQ $1, $2, 12
Action: If ($1 != $2), PC = PC + 4
Action: If ($1 == $2), PC = PC + 4 + 48
Single-Cycle Control
What is single cycle control?

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

Instr Mem

Addr Data

Equal

rs, rt, rd, imm

RegFile

rs1 rd1

rs2 rs1

ws rd2

wd WE

RegDest

RegWr

ExtOp

ALUsrc

ALUctr

32

32

32

32

32

32

32

32

RegDest

RegWr

ExtOp

ALUsrc

ALUctr

32

32

32

32

32

32

32

32

RegDest

RegWr

ExtOp

ALUsrc

ALUctr

32

32

32

32

32

32

32

32

Data Memory

Addr Dout

32

32

32

32

32

32

32

32

MemToReg

MemWr

MemToReg

MemWr

MemToReg

MemWr

MemToReg

MemWr

MemToReg

MemWr
Two goals when specifying control logic

**Bug-free:** One “0” that should be a “1” in the control logic function breaks contract with the programmer.

Should be easy for humans to read and understand: sensible signal names, symbolic constants ...

**Efficient:** Logic function specification should map to hardware with good performance properties: fast, small, low power, etc.
In practice: Use behavioral Verilog

Advice: Carefully written Verilog will yield identical semantics in ModelSim and Synplicity. If you write your code in this way, many “works in Modelsim but not on Xilinx” issues disappear.

Always check log files, and inspect output tools produce!

Look for tell-tale Synplicity “warnings and errors” messages!

“latch generated”, “combinational loop detected”, etc

Automate with scripts if possible.
Labs: A small subset of MIPS...

<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>

What if some other instruction appears in the instruction stream?

Note that unlike commercial implementations, your processor does not implement exception handling. So, if an instruction other than the ones listed above appears in the instruction stream, what your processor does is undefined by this spec (a practical option is to treat undefined instructions as no-ops).

For labs: undefined.  Real world: exceptions.
Why not in labs? Doubles complexity!

Components in blue handle exceptions ...
Will cover this (pipelined CPU) example later in the term ...
Lab 2: Learn from the past ...

Implement the following instructions in your processor:

<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>

Compare semantics are tricky!

Most project teams fail the Lab 2 checkoff the first time because they did not implement MIPS ISA for these correctly!
Where we are now, and what is next

We know how to map ISA syntax and semantics into single-cycle hardware

<table>
<thead>
<tr>
<th>Th 9/7</th>
<th>Testing and Teamwork</th>
</tr>
</thead>
<tbody>
<tr>
<td>F 9/8</td>
<td></td>
</tr>
<tr>
<td>Sa 9/9</td>
<td></td>
</tr>
<tr>
<td>Su 9/10</td>
<td></td>
</tr>
<tr>
<td>M 9/11</td>
<td></td>
</tr>
<tr>
<td>T 9/12</td>
<td>Timing</td>
</tr>
</tbody>
</table>

How to make sure your Lab 2 design implements the ISA correctly.

Software for teamwork, group dynamics, etc ...

Top-down view of how signals move through your processor in time.
Tonight: Lab 1 final report due, 11:59 PM, via the submit program.

To submit the Lab 1 final report:

1. Log into the Windows Terminal Server iserver2.eecs.berkeley.edu, using RDC (see Resources page) or via other means.

2. Once logged into iserver2, copy the file m:\bin\submit-fall2006.exe to your desktop (which is on the C: drive).

3. Execute the copied file on your desktop and follow the instructions.

At the moment, submit-fall2006 is only known to work on iserver2 -- once we know it works on the machines in 125 and 119 Cory we will make an announcement.
Administrivia: Upcoming deadlines...

- **Tonight:** Lab 1 final report due, 11:59 PM, via the submit program.
- **Thursday:** Lab 2 preliminary design document due to TAs via email, 11:59 PM.
- **Friday:** “Design Document Review” in section, 125 Cory.
- **Monday:** Lab 2 final design document due to TAs via email, 11:59 PM.
Office Hours, Mid-terms ...

Udam: MW 6-7 PM, 125 Cory
Jue: TTh 3-4 PM, 125 Cory
John: TTh 10-11AM, 315 Soda

Mid-term 1: Tuesday October 3rd, 6:00 to 9:00 PM, TBA.

Mid-term 2: Tuesday December 5th, 6:00 to 9:00 PM, TBA.

Last call for schedule conflicts ...
Josh Fisher: idea grew out of his Ph.D (1979) in compilers


VLIW

Very Long Instruction Words

Led to a startup (MultiFlow) whose computers worked, but which went out of business ... the ideas remain influential.
Basic Idea: Super-sized Instructions

Example: All instructions are 64-bit. Each instruction consists of two 32-bit MIPS instructions, that execute in parallel.

Syntax: \texttt{ADD $8 \ $9 \ $10} \ \textbf{Semantics:} \ $8 = $9 + $10

Syntax: \texttt{ADD $7 \ $8 \ $9} \ \textbf{Semantics:} \ $7 = $8 + $9

A 64-bit VLIW instruction
VLIW Assembly Syntax ...

Denotes start of an instruction word.

Instr:

ADD $8 $9 $10
ADD $7 $8 $9

Listed operators all execute in parallel.

Instr:

SUB $2 $3 $0
OR  $1 $5 $4

Execute in parallel.

[ ... ]

Label:

AND $5 $2 $3
OR  $1 $5 $4

Branch label name instead of default “instr”.
32-bit & 64-bit semantics different? Yes!

Assume: $7 = 7$, $8 = 8$, $9 = 9$, $10 = 10$ (decimal)

32-bit MIPS:

```
ADD $8 $9 $10;  Result: $8 = 19
ADD $7 $8 $9;  Result: $7 = 28
```

VLIW:

```
Instr: ADD $8 $9 $10 ; result $8 = 19
ADD $7 $8 $9 ; result $7 = 17 (not 28)
```
Design: A 64-bit VLIW R-format CPU

Syntax: ADD $8 $9 $10  Semantics: $8 = $9 + $10

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

Syntax: ADD $7 $8 $9  Semantics: $7 = $8 + $9

No branches or jumps: machine only runs **straight line** code.

No loads or stores: machine has no use for **data memory**, only **instruction memory**.
VLIW: Straight-line Instruction Fetch

Simple changes to support 64-bit instructions ...

+8 in hexadecimal -- 64 bit instructions

CLK

Addr

Data

0x8
Computing engine of VLIW R-format CPU

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

- **RegFile**
  - rs1
  - rs2
  - rd1
  - rd2
  - ws1
  - ws2
  - wd1
  - wd2
  - WE1
  - WE2

- ALU
  - op
  - 32

- ALU
  - op
  - 32

- ALU
  - op
  - 32
What have we gained with 64-bit VLIW?

If:

★ Clock speed remains the same.
★ All 32-bit operators do useful work.

Performance doubles!

Syntax: ADD $8 $9 $10  Semantics: $8 = $9 + $10

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

Syntax: ADD $7 $8 $9  Semantics: $7 = $8 + $9

N x 32-bit VLIW yields factor of N speedup!

Multiflow: $N = 7, 14, or 28 (3 CPUs in product family)
What does $N = 14$ assembly look like?

<table>
<thead>
<tr>
<th>instr</th>
<th>cl0</th>
<th>cl0</th>
<th>cl0</th>
<th>cl0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ialu0e</td>
<td>ialu1e</td>
<td>fau0e</td>
<td>fau1e</td>
</tr>
<tr>
<td></td>
<td>st.64</td>
<td>cgt.s32</td>
<td>add.f64</td>
<td>add.f64</td>
</tr>
<tr>
<td></td>
<td>sb1.r0,r2,17#144</td>
<td>li1bb.r4,r34,6#31</td>
<td>lsb.r4,r8,r0</td>
<td>lsb.r6,r40,r32</td>
</tr>
<tr>
<td></td>
<td>br</td>
<td>br</td>
<td>br</td>
<td>br</td>
</tr>
<tr>
<td></td>
<td>true and r3</td>
<td>false or r4</td>
<td>true and r3</td>
<td>false or r4</td>
</tr>
</tbody>
</table>

Two instructions from a scientific benchmark (Linpack) for a MultiFlow CPU with 14 operations per instruction.
What have we gained with 64-bit VLIW?

If: A very big “if”!

* Clock speed remains the same
* All 32-bit operators do useful work.

Performance doubles!

Syntax: ADD $8 $9 $10  Semantics: $8 = $9 + $10

Syntax: ADD $7 $8 $9  Semantics: $7 = $8 + $9

N x 32-bit VLIW yields factor of N speedup!

Multiflow: N = 7, 14, or 28 (3 CPUs in product family)
As N scales, HW and SW needs conflict

Software need: All operators do useful work.

Hardware need: Clock does not slow down.
Example problem: Register file ports ...

N ALUs require $2N$ read ports and $N$ write ports. Why is this a problem?
Recall: Register File Design

More read ports increases fanout, slows down reads.

More write ports adds data muxes, demux OR tree.
Split register files: A solution?

Software need: All operators do useful work.

Too often, the data an ALU needs to do "useful work" will not be in its own regfile.
Architect’s job: Find a good compromise

Example solution: **Split** register files, with a dedicated bus and special instructions for moves between regfiles.

May hurt software more than it helps hardware :-(

Instruction Set Architecture: Where the conflict plays out.

---

CS 152 L1: The MIPS ISA
Branch policy: All instruction operators execute

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BNE $8 $9 Label ADD $7 $8 $9

ADD executes if branch is taken or not taken.

Problem: Large N machines find it hard to fill all operators with useful work.

Solution: New “predication” operator.

Syntax: SELECT $7 $8 $9 $10

Semantics: If $8 == 0, $7 = $10, else $7 = $9

Permits simple branches to be converted to inline code.
Branch nesting in a single instruction ...

```
BEQ $8 $9 LabelOne

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BEQ $11 $12 LabelTwo

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Conundrum: How to define the semantics of multiple branches in one instruction?

Solution: Nested branch semantics

If $8 == $9, branch to LabelOne
Else $11 == $12, branch to LabelTwo

MultiFlow: N-way Branch priority set in an opcode field.
Will return to VLIW later in semester ...
So many “teamwork” topics, we need to do one early …

Design Notebook
Why should you keep a design notebook?

° Keep track of the design decisions **and the reasons behind them**
  • Otherwise, it will be hard to debug and/or refine the design
  • Write it down so that can remember in long project: 2 weeks ->2 yrs
  • Others can review notebook to see what happened

° Record insights you have on certain aspect of the design as they come up

° Record of the different design & debug experiments
  • Memory can fail when very tired

° Industry practice: learn from others mistakes
Why do we keep it on-line?

° You need to force yourself to take notes
  • Open a window and leave an editor running:
    1) Acts as reminder to take notes
    2) Makes it easy to take notes
  • 1) + 2) => will actually do it

° Take advantage of the window system’s “cut and paste” features

° It is much easier to read typing than writing

° Also, paper log books have problems
  • Limited capacity => end up with many books
  • May not have right book with you.
  • Can use computer to search files.
How to do it? See “Resources” web page

° Keep it simple
  • DON’T make it too elaborate (fonts, layout, ...)

° Separate the entries by dates
  • type “date” command in another window and cut&paste

° Start day with problems going to work on today

° Record output of simulation into log with cut&paste; add date
  • May help sort out which version of simulation did what

° Record key email with cut&paste

° Record of what works & doesn’t helps team decide what went wrong after you left

° Index: write a one-line summary of what you did at end of each day
* Index

Wed Sep  6 00:47:28 PDT 1995 - Created the 32-bit comparator component
Thu Sep  7 14:02:21 PDT 1995 - Tested the comparator
Mon Sep 11 12:01:45 PDT 1995 - Investigated bug found by Bart in comp32 and fixed it

---

Wed Sep  6 00:47:28 PDT 1995

Goal: Layout the schematic for a 32-bit comparator

I've layed out the schematics and made a symbol for the comparator.
I named it comp32. The files are
~/wv/proj1/sch/comp32.sch
~/wv/proj1/sch/comp32.sym

---

Wed Sep  6 02:29:22 PDT 1995

- Add 1 line index at front of log file at end of each session: date+summary
- Start with date, time of day + goal
- Make comments during day, summary of work
- End with date, time of day (and add 1 line summary at front of file)
Goal: Test the comparator component

I've written a command file to test comp32. I've placed it in ~/wv/proj1/diagnostics/comp32.cmd.

I ran the command file in viewsim and it looks like the comparator is working fine. I saved the output into a log file called ~/wv/proj1/diagnostics/comp32.log

Notified the rest of the group that the comparator is done.
Goal: Investigate bug discovered in comp32 and hopefully fix it

Bart found a bug in my comparator component. He left the following e-mail.

Hey Bruce,
I think there's a bug in your comparator. The comparator seems to think that ffffffff and fffffff7 are equal.

Can you take a look at this?
Bart
I verified the bug. here's a viewsim of the bug as it appeared..
(equal should be 0 instead of 1)

------------------
SIM>stepsize 10ns
SIM>v a_in A[31:0]
SIM>v b_in B[31:0]
SIM>w a__in b_in equal
SIM>a a__in fffffffff\h
SIM>a b__in ffffffff7\h
SIM>sim
Simulation stopped at 10.0ns.
------------------

Ah. I've discovered the bug. I mislabeled the 4th net in
the comp32 schematic.

I corrected the mistake and re-checked all the other
labels, just in case.

I re-ran the old diagnostic test file and tested it against
the bug Bart found. It seems to be working fine. hopefully
there aren't any more bugs:)
On second inspection of the whole layout, I think I can remove one level of gates in the design and make it go faster. But who cares! the comparator is not in the critical path right now. the delay through the ALU is dominating the critical path. so unless the ALU gets a lot faster, we can live with a less than optimal comparator.

I e-mailed the group that the bug has been fixed

Mon Sep 11 14:03:41 PDT 1995

• Perhaps later critical path changes; what was idea to make comparator faster? Check log book!
Where we are now, and what is next

We know how to map ISA syntax and semantics into single-cycle hardware

<table>
<thead>
<tr>
<th>Th 9/8</th>
<th>Testing and Teamwork</th>
</tr>
</thead>
<tbody>
<tr>
<td>F 9/9</td>
<td></td>
</tr>
<tr>
<td>Sa 9/10</td>
<td></td>
</tr>
<tr>
<td>Su 9/11</td>
<td></td>
</tr>
<tr>
<td>M 9/12</td>
<td></td>
</tr>
<tr>
<td>T 9/13</td>
<td>Timing</td>
</tr>
</tbody>
</table>

How to make sure your Lab 2 design implements the ISA correctly.

Software for teamwork, group dynamics, etc ...

Top-down view of how signals move through your processor in time.

Good luck with Lab 1!