CS 152
Computer Architecture and Engineering
Lecture 2 – Single Cycle Wrap-up

2014-1-23
John Lazzaro
(not a prof - “John” is always OK)

TA: Eric Love

www-inst.eecs.berkeley.edu/~cs152/
Nvidia
Tegra K1
Tech Talk

5:30 PM
this
Thursday
in the Woz.

Tegra K1 remixes the Kepler GPU architecture for lowpower SOCs.
Topics for today’s lecture

- Single-Cycle CPU Design
- Short Break.

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

Walk up to John and Eric during the break to discuss individual administrative issues.
Single Cycle CPU Design
Single Cycle CPU design

All instructions execute in a **single cycle** of the clock.
(positive edge to positive edge)

All state elements act like positive edge-triggered flip flops.
No delayed branches.

The PC of the next instruction executed after a taken branch is the branch target of the taken branch.

No delayed loads.

The next instruction executed after the load sees the value that was retrieved by the load in the appropriate register.

We will re-introduce delayed branch and delayed load semantics in the pipelining lecture.
Architected state

The state **visible** to the programmer.

The state that appears in **machine language** instructions.

32 32-bit Registers

- R0 [hardwired to constant 0]
- R1
- ...
- R30
- R31

Program Counter (PC)

32 bits

Main Memory

$2^{32}$ bytes
organized as 32-bit words

<table>
<thead>
<tr>
<th>addr</th>
<th>next instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000</td>
<td>...</td>
</tr>
<tr>
<td>000000004</td>
<td></td>
</tr>
<tr>
<td>000000008</td>
<td></td>
</tr>
<tr>
<td>FFFFFFFF8</td>
<td></td>
</tr>
<tr>
<td>FFFFFFFFC</td>
<td></td>
</tr>
<tr>
<td>FFFFFFFFF</td>
<td></td>
</tr>
</tbody>
</table>
Architected state

All state elements in our single-cycle CPU design hold architected state.

32 32-bit Registers

<table>
<thead>
<tr>
<th>R0 [hardwired to constant 0]</th>
<th>R1</th>
<th>...</th>
<th>R30</th>
<th>R31</th>
</tr>
</thead>
</table>

Program Counter (PC)

Main Memory

$2^{32}$ bytes
organized as 32-bit words

<table>
<thead>
<tr>
<th>addr</th>
<th>next instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000</td>
<td></td>
</tr>
<tr>
<td>0000000004</td>
<td></td>
</tr>
<tr>
<td>0000000008</td>
<td></td>
</tr>
<tr>
<td>FFFFFFFF8</td>
<td></td>
</tr>
<tr>
<td>FFFFFFFFC</td>
<td></td>
</tr>
<tr>
<td>FFFFFFFF</td>
<td></td>
</tr>
</tbody>
</table>

next instr

...
Recall: MIPS R-format instructions

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

---

**Instruction Fetch**

**Instruction Decode**

**Operand Fetch**

**Execute**

**Result Store**

**Next Instruction**

**Fetch next inst from memory:** 012A4020

<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

"Retrieve" register values: $9 $10

Add $9 to $10

Place this sum in $8

Prepare to fetch instruction that follows the ADD in the program.
Goal #1: An R-format single-cycle 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.
Separate Read-Only Instruction Memory

Reads are **combinational**: Put a stable address on input, a short time later data appears on output.

Not concerned about how programs are loaded into this memory.

Related to separate instruction and data caches in "real" designs.
Task #1: Straight-line Instruction Fetch

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

Why increment every cycle? Straight-line code.
Why +4 and not +1? 32-bit instructions.

PC == Program Counter, points to next instruction.
New Component: Register (for PC)

Built out of an array of flip-flops

In later examples, we will add an “enable” input: clock edge updates state only if enable is high.

How to design? Mux Q back to D.
New Component: A 32-bit adder (ALU)

**Combinational:**
Put A and B values on inputs, a short time later A + B appears on output.

**ALU:** Combinational part that is able to execute many functions of A and B (add, sub, and, or, ...). The “op” value selects the function.

Sometimes, extra outputs for use by control logic ...
Design: Straight-line Instruction Fetch

State machine design in the service of an ISA

CLK

Addr

Data

+4 in hexadecimal

IMem[PC]

IMem[PC + 4]

IMem[PC + 8]
Goal #1: An R-format single-cycle CPU

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

Decode fields to get: ADD $8 $9 $10

“Retrieve” register values: $9 $10

Add $9 to $10

Place this sum in $8

Prepare to fetch instruction that follows the ADD in the program.

Done! To continue, we need registers...
Why is R0 special?

R0 - The constant 0

Why is R0 special?

“two read ports”

How do we add a second write port?

Duplicate write buses, add muxes.
Register File Schematic Symbol

Why do we need WE?  
Advanced planning, for instructions that don’t write the register file.

![Register File Schematic Symbol](image)

If we had a MIPS register file w/o WE, how could we work around it?

**Do writes to the hardwired-to-zero register R0.**
Goal #1: An R-format single-cycle CPU

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

What do we do with these?

Fetch next inst from memory: 012A4020

Decode fields to get: ADD $8 $9 $10

"Retrieve" register values: $9 $10

Add $9 to $10

Place this sum in $8

Prepare to fetch instruction that follows the ADD in the program.
Computing engine of the R-format CPU

Decide fields to get: ADD $8 $9 $10

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

```
opcode | rs  | rt  | rd  | shamt | funct |
+-------+-----+-----+-----+-------+-------|
```

Logic

What do we do with WE?

Hardwire to always write.
Putting it all together ...

Is it safe to use same clock for PC and RegFile? Yes!

To rs1, rs2, ws, op decode logic ...
Recall: Our ideal-world D Flip-Flop

Value of D is sampled on positive clock edge. Q outputs sampled value for rest of cycle.

Also assume: clocks arrive at all flip flops simultaneously.
Reminder: How data flows after posedge
Next posedge: Update state and repeat

In this ideal world, as long as the clock is slow enough, the machine gets the right answer.

In Metrics lecture, we look at the assumptions behind ideality.
Next Step ...

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

Learn how to efficiently “merge” single-function machines to make one general-purpose machine.
**Goal #2: add I-format ALU instructions**

**Syntax:** ORI $8$ $9$ 64  
**Semantics:** $8 = 9 \mid 64$

<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:** $0x8000 \Rightarrow 0xFFFF8000$

Some MIPS instructions zero-extend immediate field, other instructions sign-extend.
Computing engine of the I-format CPU

Decode fields to get: ORI $8 $9 64

In a Verilog implementation, what should we do with rs2?
Tie to the value that minimizes energy consumption.
Merging data paths ...

Add muxes

How many? 2
(ignoring ALU control)

Where?
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
- rd1
- rd2
- WE

RegDest

- 5
- 5
- 5
- 32

Ext

- 32

ExtOp

- ALUsrc

ALUctr

- op

ALU

- 32

RegDest

- 32

Immediate

If you watched it being designed, it's understandable...
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: \( \text{LW} \; $1, \; 32($2) \)  
Action: \( $1 = M[\; $2 + 32] \)

Syntax: \( \text{SW} \; $3, \; 12($4) \)  
Action: \( M[\; $4 + 12] = $3 \)

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

Reads are **combinational**: Put a stable address on \( \text{Addr} \), a short time later \( \text{Dout} \) is ready.

 Writes are **clocked**: If \( \text{WE} \) is high, memory \( \text{Addr} \) captures \( \text{Din} \) on positive edge of clock.

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

Recall spec: no load delay slot.

<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
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?

Increases branch range to 128 KB.

Zero-extend or sign-extend immediate field?  Sign-extend.

Why is this extension method a good idea?

Supports forward and backward branches.
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 $(\$1 \neq \$2)$, $PC = PC + 4$

Action: If $(\$1 == \$2)$, $PC = PC + 4 + 48$
**Design: Instruction Fetch with Branch**

**Syntax:** \( \text{BEQ } $1, $2, 12 \)

**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
rs, rt, rd, imm

RegFile
rs1
rs2
ws
wd
rd1
rd2
WE

RegDest
RegWr
ExtOp
ALUsrc

ALUctr
Equal

MemToReg
PCSrc
MemWr

Data Memory
Addr
Dout
WE

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

Instr Mem
Addr Data
rs, rt, rd, imm

RegFile
rs1
rs2
ws
wd
rd1
rd2
WE

RegDest
RegWr
ExtOp
ALUsrc

ALUctr
Equal

MemToReg
PCSrc
MemWr

Data Memory
Addr
Dout
WE

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

Instr Mem
Addr Data
rs, rt, rd, imm

RegFile
rs1
rs2
ws
wd
rd1
rd2
WE

RegDest
RegWr
ExtOp
ALUsrc

ALUctr
Equal

MemToReg
PCSrc
MemWr

Data Memory
Addr
Dout
WE

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

Instr Mem
Addr Data
rs, rt, rd, imm

RegFile
rs1
rs2
ws
wd
rd1
rd2
WE

RegDest
RegWr
ExtOp
ALUsrc

ALUctr
Equal

MemToReg
PCSrc
MemWr

Data Memory
Addr
Dout
WE

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

Efficient: Logic function specification should map to hardware with good performance properties: fast, small, low power, etc.

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

Thursday, January 23, 14
TIME MACHINE BACK TO FPGA-ORIENTED 2006 CS 152 ...
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.
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,slt,sltu</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 ...
A slide from Eric’s section on 1/22 ...

What do Architects do?

- I.e., what do they *spend* their time doing?
- Most time not spent building chips!
- Characterize workloads, simulate processors, analyze performance tradeoffs

Actually, I agree ...

However ... if you aren’t able to design at the level we just worked through, you will unwitting propose unbuildable ideas.

... and lose the confidence of your fellow team members.
Upcoming 2014 Lab 1 ...

If you understood this lecture, you now have the conceptual foundation to modify this design.

Written in Chisel

RISC-V Single-Cycle CPU

... if you’re willing to spend a few days to teach yourself Chisel.
Break
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.**

<table>
<thead>
<tr>
<th>Syntax: ADD $8 $9 $10</th>
<th>Semantics: $8 = $9 + $10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>opcode</strong></td>
<td><strong>rs</strong></td>
</tr>
<tr>
<td><strong>opcode</strong></td>
<td><strong>rs</strong></td>
</tr>
</tbody>
</table>

Syntax: ADD $7 $8 $9  
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:

\[
\begin{align*}
&\text{ADD} \quad \$8 \; \$9 \; \$10; \quad \text{Result: } \$8 = 19 \\
&\text{ADD} \quad \$7 \; \$8 \; \$9; \quad \text{Result: } \$7 = 28
\end{align*}
\]

VLIW:

\[
\begin{align*}
&\text{Instr: } \quad \text{ADD} \quad \$8 \; \$9 \; \$10 \quad ; \quad \text{result } \$8 = 19 \\
&\quad \text{ADD} \quad \$7 \; \$8 \; \$9 \quad ; \quad \text{result } \$7 = 17 \; \text{(not 28)}
\end{align*}
\]
**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
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
- rd3
- rd4
- WE1
- WE2

ALU

- op
- 32
- 32
- 32
- 32
- 32
- 32
- 32
- 32

RegFile inputs:
- 5
- 5
- 5
- 32
- 5
- 32
- 5
- 32
- 5
- 32
- 5
- 32
- 5
- 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>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Syntax:</th>
<th>ADD</th>
<th>$7</th>
<th>$8</th>
<th>$9</th>
<th>Semantics:</th>
<th>$7 = $8 + $9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax:</td>
<td>ADD</td>
<td>$8</td>
<td>$9</td>
<td>$10</td>
<td>Semantics:</td>
<td>$8 = $9 + $10</td>
</tr>
</tbody>
</table>
```

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?

Two instructions from a scientific benchmark (Linpack) for a MultiFlow CPU with 14 operations per instruction.

<table>
<thead>
<tr>
<th>instr</th>
<th>cl0</th>
<th>ialu0e</th>
<th>st.64</th>
<th>sb1.r0,r2,17#144</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cl0</td>
<td>ialu1e</td>
<td>cgt.s32</td>
<td>li1bb.r4,r34,6#31</td>
</tr>
<tr>
<td></td>
<td>cl0</td>
<td>falu0e</td>
<td>add.f64</td>
<td>lsb.r4,r8,r0</td>
</tr>
<tr>
<td></td>
<td>cl0</td>
<td>falu1e</td>
<td>add.f64</td>
<td>lsb.r6,r40,r32</td>
</tr>
<tr>
<td></td>
<td>cl0</td>
<td>ialu0l</td>
<td>dld.64</td>
<td>fb1.r4,r2,17#208</td>
</tr>
<tr>
<td></td>
<td>cl0</td>
<td>ialu0e</td>
<td>dld.64</td>
<td>fb1.r34,r1,17#216</td>
</tr>
<tr>
<td></td>
<td>cl0</td>
<td>ialu1e</td>
<td>cgt.s32</td>
<td>li1bb.r3,r32,zero</td>
</tr>
<tr>
<td></td>
<td>cl0</td>
<td>falu0e</td>
<td>add.f64</td>
<td>lsb.r4,r8,r6</td>
</tr>
<tr>
<td></td>
<td>cl0</td>
<td>falu1e</td>
<td>add.f64</td>
<td>lsb.r6,r40,r38</td>
</tr>
<tr>
<td></td>
<td>cl0</td>
<td>ialu0l</td>
<td>st.64</td>
<td>sb1.r2,r1,17#152</td>
</tr>
<tr>
<td></td>
<td>cl0</td>
<td>ialu1l</td>
<td>add.u32</td>
<td>lib.r32,r36,6#32</td>
</tr>
<tr>
<td></td>
<td>cl0</td>
<td>br</td>
<td>true and r3</td>
<td>L23?3</td>
</tr>
<tr>
<td></td>
<td>cl0</td>
<td>br</td>
<td>false or r4</td>
<td>L24?3</td>
</tr>
</tbody>
</table>
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

<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)
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.

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.
Branch policy: All instructions execute

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

 opcode|   rs  |   rt |   rd |  shamt |  funct

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

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>
</table>

BEQ $11 $12 LabelTwo

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 ...
Next Tuesday

How to measure the “goodness” of an architecture (and an implementation) ...

<table>
<thead>
<tr>
<th></th>
<th>Metrics</th>
<th>Chapter 1.5-11.</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>1/28</td>
<td></td>
</tr>
</tbody>
</table>

... and if we have time, we’ll discuss microcode (on class website, click on link for reading PDF).

<table>
<thead>
<tr>
<th></th>
<th>Single-Cycle Wrap-up + VLIW + Microcode</th>
<th>Chapter 3.7, COD Appendix D.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Th</td>
<td>1/23</td>
<td></td>
</tr>
</tbody>
</table>

Have a good weekend!