Topics for today’s lecture

* Instruction Set Architectures (ISAs)

* Single-Cycle CPU Design

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

MIPS32™ Architecture For Programmers
Volume II: The MIPS32™ Instruction Set

Document Number: MD00086
Revision 2.00
June 9, 2003
New successful instruction sets are rare

Implementors suffer with original sins of ISAs, to support the installed base of software.
**Instruction Sets: A Thin Interface**

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

![Diagram of instruction set architecture]

**Fieldsize:** 6 bits  5 bits  5 bits  5 bits  5 bits  5 bits  6 bits  
**Bitfield:** opcode rs rt rd shamt funct

**Binary:** 000000  01001  01010  01000  00000  100000  
**In Hexadecimal:** 012A4020

---

**Additional Details:**

- **Digital Design**
- **Circuit Design**
- **Transistors**
- **Memory**
- **Processor**
- **I/O system**
- **Operating System (Mac OS X)**
- **Compiler**
- **Assembler**
- **Application (iTunes)**

---

**Notes:**

- The diagram illustrates the hierarchy of hardware and software components involved in executing instructions.
- The instruction set architecture is shown with specific fields and bit lengths.
- The syntax of the instruction and its semantics are clearly defined.
- The binary and hexadecimal representations are provided for clarity.

---

**References:**

- For further details on instruction sets and architecture, refer to relevant computer science textbooks or online resources.
- The specific values and formats are based on common computational systems and architectures.
Hardware implements semantics ...

**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.
ADD syntax & semantics, as seen in the MIPS ISA document.
Memory Instructions: \texttt{LW} $1,32($2)

- **Fetch the load inst from memory**
  
  \begin{tabular}{cccc}
  \texttt{opcode} & \texttt{rs} & \texttt{rt} & \texttt{offset} \\
  \end{tabular} 

  \textbf{“I-Format”}

- **Decode fields to get**: \texttt{LW} $1, 32($2)

  \textbf{“Retrieve” register value:} $2$

- **Compute memory address**: $32 + $2$

- **Load memory address contents into**: $1$

- **Prepare to fetch instr that follows the \texttt{LW} in the program. Depending on load semantics, new \$1 is visible to that instr, or not until the following instr (“delayed loads”).**
Load Word

\[ \text{LW} \]

Syntax & semantics, as seen in the MIPS ISA document.

**Format:** \( \text{LW } rt, \text{offset}(\text{base}) \)  

**Purpose:**  
To load a word from memory as a signed value

**Description:**  
\( rt \leftarrow \text{memory[base+offset]} \)  
The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR \( rt \). The 16-bit signed \( \text{offset} \) is added to the contents of GPR \( \text{base} \) to form the effective address.

**Restrictions:**  
The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

**Operation:**  
\[
\begin{align*}
\text{vAddr} &\leftarrow \text{sign_extend} (\text{offset}) + \text{GPR[base]} \\
\text{if } &\text{vAddr}_{11:0} \neq 0 \text{ then} \\
&\text{SignalException(AddressError)} \\
\text{endif} \\
(p\text{Addr}, \text{CCA}) &\leftarrow \text{AddressTranslation} (\text{vAddr}, \text{DATA, LOAD}) \\
\text{memword} &\leftarrow \text{LoadMemory} (\text{CCA, WORD, pAddr}, \text{vAddr, DATA}) \\
\text{GPR[rt]} &\leftarrow \text{memword}
\end{align*}
\]

**Exceptions:**  
TLB Refill, TLB Invalid, Bus Error, Address Error, Watch
Branch Instructions: **BEQ $1, $2, 25**

- **Fetch branch inst from memory**
  - Decode fields to get: `BEQ $1, $2, 25`
  - "Retrieve" register values: $1, $2
  - Compute if we take branch: $1 == $2 ?

- **"I-Format"**

ALWAYS prepare to fetch instr that follows the BEQ in the program ("delayed branch"). IF we take branch, the instr we fetch AFTER that instruction is PC + 4 + 100.

**PC == "Program Counter"**
BEQ syntax & semantics, as seen in the MIPS ISA document.
define: The Architect’s Contract

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

- As each instruction completes, the machine state (regs, mem) 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.
Single Cycle CPU Design
Single cycle data paths: Assumptions

Processor uses synchronous logic design (a “clock”).

All state elements act like positive edge-triggered flip flops.

<table>
<thead>
<tr>
<th>f</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MHz</td>
<td>1 μs</td>
</tr>
<tr>
<td>10 MHz</td>
<td>100 ns</td>
</tr>
<tr>
<td>100 MHz</td>
<td>10 ns</td>
</tr>
<tr>
<td>1 GHz</td>
<td>1 ns</td>
</tr>
</tbody>
</table>
Review: Edge-Triggered D Flip Flops

Value of D is sampled on positive clock edge. Q outputs sampled value for rest of cycle.
Review: Edge-Triggering in Verilog

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

module ff(D, Q, CLK);
input D, CLK;
output Q;
always @ (CLK)
  Q <= D;
endmodule

Module code has two bugs.

Where?
Review: Edge-Triggered D Flip Flops

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

module ff(D, Q, CLK);
input D, CLK;
output Q;
reg Q;
always @ (posedge CLK)
    Q <= D;
endmodule
define: **Single-cycle datapath**

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

**Advantage:** a great way to learn CPUs.

**Drawbacks:** unrealistic hardware assumptions, slow clock period
Recall: MIPS R-format instructions

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

- **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.**
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:

```
<table>
<thead>
<tr>
<th>CLK</th>
<th>Addr</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>PC + 4</td>
<td>PC + 8</td>
</tr>
</tbody>
</table>
```

Why do we increment every clock cycle? Why +4 and not +1?

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.

Logic design?
New Component: A 32-bit adder (ALU)

**Combinational:** Put a 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

0x4

+4 in hexadecimal
**Goal #1: An R-format single-cycle CPU**

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

**Done! To continue, we need registers...**

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.
MIPS Register file: From the top down

Why is R0 special?

**R0 - The constant 0**

“two read ports”

How do we add a second write port?

**Why is R0 special?**

**two read ports**

**How do we add a second write port?**
Why do we need WE?

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

Decode fields to get: ADD $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>

What do we do with WE?
Putting it all together ...

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

To rs1, rs2, ws, op decode logic ...

Logic

0x4
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

0x4

PC

Instr Mem

Addr Data

Logic

RegFile

rs1
rs2
ws
wd

rd1
rd2

WE

ALU

op

D Q

Mem

Instr

Data

Addr

PC

Q

+
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 Timing 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 | 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

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

In a Verilog implementation, what should we do with rs2?
Merging data paths ...

Add muxes

How many? (ignore ALU control)

Where?

I-format

R-format
The merged data path...

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: LW $1, 32($2)  Syntax: SW $3, 12($4)

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>Syntax: <strong>LW</strong> $1, 32($2)</td>
<td>Syntax: <strong>SW</strong> $3, 12($4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Action:</strong> $1 = M[$2 + 32]</td>
<td><strong>Action:</strong> M[$4 + 12] = $3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:** \( \text{BEQ } $1, $2, 12 \)

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

**Action:** If \( ($1 == $2) \), \( \text{PC} = \text{PC} + 4 + 48 \)
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:** \( \text{BEQ } \$1, \$2, 12 \)

**How do we add this behavior?**

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

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

Syntax:

\[
\text{BEQ } \$1, \$2, 12
\]

Action:

- If \( \$1 \neq \$2 \), \( \text{PC} = \text{PC} + 4 \)
- If \( \$1 = \$2 \), \( \text{PC} = \text{PC} + 4 + 48 \)
**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

RegFile
rs1 rs2 ws wd rd1 rd2

Equal
RegDest
RegWr
ExtOp
ALUsrc
ALUctr
MemWr
MemToReg

RegDest
rs, rt, rd, imm

Ext
Din
Addr
Data Memory

PCSrc

Instr Mem

PCSrc

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 ...
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.

In practice: Use behavioral Verilog

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.
F06 152 Labs: A small subset of MIPS ...

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>

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

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

**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:

```
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

64
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 \(\rightarrow\) rd1
rs2 \(\rightarrow\) rd2
ws1 \(\rightarrow\) rd3
wd1 \(\rightarrow\) rd4

ALU

op \(\rightarrow\) A
L
U

32

op

5

32

5

32

5

32

5

32

RegFile

rs1
rs2
ws1
wd1

32

5

32

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>
</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?

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>fau0e</td>
<td>add.f64</td>
<td>lsb.r4,r8,r0</td>
</tr>
<tr>
<td></td>
<td>cl0</td>
<td>fau1e</td>
<td>add.f64</td>
<td>lsb.r6,r40,r32</td>
</tr>
<tr>
<td></td>
<td>cl1</td>
<td>ialu0l</td>
<td>dld.64</td>
<td>fb1.r4,r2,17#208</td>
</tr>
<tr>
<td></td>
<td>cl1</td>
<td>ialu0e</td>
<td>dld.64</td>
<td>fb1.r34,r1,17#216</td>
</tr>
<tr>
<td></td>
<td>cl1</td>
<td>ialu1e</td>
<td>cgt.s32</td>
<td>li1bb,r3,r32,zero</td>
</tr>
<tr>
<td></td>
<td>cl1</td>
<td>fau0e</td>
<td>add.f64</td>
<td>lsb.r4,r8,r6</td>
</tr>
<tr>
<td></td>
<td>cl1</td>
<td>fau1e</td>
<td>st.64</td>
<td>lsb.r6,r40,r38</td>
</tr>
<tr>
<td></td>
<td>cl1</td>
<td>ialu0l</td>
<td>add.u32</td>
<td>lib.r32,r36,6#32</td>
</tr>
<tr>
<td></td>
<td>cl1</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>

---

Table 2. Hardware performance of the Trace 300 family.
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

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

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

**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 ...

\[ N \text{ ALUs require } 2 \times N \text{ read ports and } N \text{ write ports.} \]

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.
Branch policy: All *instr* 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>opcode</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</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

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 Monday:

First Design Review

This Friday: Look-ahead for Design Review, 125 Cory, 10 AM