This lecture deals with Instruction Set Design.
The instruction set for a processor is the set of all the instructions supported by that processor.
Basic Processor Design

Start with RISC\textsuperscript{1} assumptions since we want a simple processor.

- Fixed instruction length
  - keeps control simple (very useful for pipelining\textsuperscript{2})
- Load/store architecture & a large number of general purpose registers
  - avoids slow memory access
- Very few addressing modes
  - complex addressing modes are seldom used
- No complex instructions
  - keep everything simple

\textsuperscript{1}\textit{RISC} = Reduced Instruction Set Computer
\textsuperscript{2}\textit{RISC} machines were initially designed to increase performance through pipelining - your processor will not use pipelining
Basic Processor Design – RISC

• All instructions are the same length
  – typically 1 word (i.e. 16 bits)

• Load/Store Architecture
  – All arithmetic and logic instructions deal only with registers and immediate values\(^3\)
    e.g. \texttt{ADD R3, R2, R5} \quad R5 \leftarrow R3 + R2
    \quad \texttt{OR R3, 13, R5} \quad R5 \leftarrow R3 \mid 13
  – Separate instructions are needed for access to locations in memory.
    e.g. \texttt{LD [R4+13], R7} \quad R7 \leftarrow \textit{mem}(R4 + 13)
    \quad \texttt{ST R7, [R4+13]} \quad \textit{mem}(R4 + 13) \leftarrow R7

\textit{mem(nnn)} is shorthand for the data location in memory with address \textit{nnn}.

– Instruction set is maximally orthogonal

\(^3\)an immediate (or literal) value is a data value encoded in the instruction word.
Basic Processor Design – RISC

Q1<sub>RISC</sub>
How many Register Addresses in an Arithmetic/Logic Instruction?

- Usually 2 or 3 for RISC
  2: ADD Rx, Ry \( Rx \leftarrow Rx + Ry \)
  3: ADD Rx, Ry, Rz \( Rz \leftarrow Rx + Ry \)

Q2<sub>RISC</sub>
How many General Purpose Registers?

- Usually \( 2^n - 1 \) for RISC
  This gives \( 2^n \) addressable registers including the dummy register, R0\(^4\).
  We then need \( n \) bits per register address in the instruction.
  With a 16 bit instruction length, \( n = 2 \) (i.e. 3 registers + R0) or \( n = 3 \) (i.e. 7 registers + R0) are sensible values.

\(^4\)R0 is always zero
Basic Processor Design – RISC

**Q3\textsubscript{RISC}**
How many Bits do we use for Short Immediates?

- Used in instructions like

\[
\begin{align*}
\text{ADD} & \quad R3, 5, R2 & R2 & \leftarrow R3 + 5 \\
\text{ST} & \quad R7, [R4+13] & \text{mem}(R4 + 13) & \leftarrow R7
\end{align*}
\]

Sensible values for a 16 bit instruction length are in the range \(4 \leq s \leq 9\)

giving 2’s complement values in the range \(-2^{s-1} \leq imm \leq 2^{s-1} - 1\)

**Q3A\textsubscript{RISC}**
Do we support All Arithmetic/Logic instructions and All Load/Store instructions in both Register-Register and Register-Immediate forms?

- Some RISC processors support only Register-Immediate form for Load/Store instructions.

- Some RISC processors support Register-Register form for all Arithmetic/Logic instructions and Register-Immediate form for a subset of these instructions.
Basic Processor Design – RISC

Q4<sub>RISC</sub>

What Instruction Fields do we provide? How are they arranged?

- RISC instruction coding is highly orthogonal - any instruction may use any registers.
- Requirement for maximum length short immediate makes RISC coding tight.

Assume Q1<sub>RISC</sub>: 3, Q2<sub>RISC</sub>: 2<sup>n</sup> - 1, Q3<sub>RISC</sub>: s, Q3A<sub>RISC</sub>: YES

A suitable coding for Arithmetic/Logic and Load/Store instructions is:

\[
x = 15 - 2n - s
\]

Example: If \( n = 2 \) and \( s = 7 \) then \( x = 4 \) giving up to \( 2^x (=16) \) instructions.
Most RISC processors support an instruction to set the upper bits of a register. The MIPS processor calls it LUI (load upper immediate) while the SPARC processor calls it SETHI.

For SPARC, the sequence of instructions required to set upper and lower parts of a register is:

\[
\text{SETHI} \ 200, \ Rx \\
\text{ADD} \ Rx, 5, \ Rx
\]

\[Rx \leftarrow 200 \times 2^{10}\]

\[Rx \leftarrow Rx + 5\]

Note that the \(\times 2^{10}\) (i.e. shift left by 10) value comes from the SPARC word length (32) less the length of the long immediate used for SETHI (22). In our example it will be \(\times 2^{16-l}\) where \(l\) is the length of our long immediate.

To code a SETHI or LUI instruction we need fewer fields:

- \(l = 16 - n - x\)

Example: If \(n = 2\) and \(x = 4\) then \(l = 10\).

Note: \(s + l \geq 16\) for the SETHI/ADD sequence to produce a 16 bit result.
Example Coding #3

Assume \( Q_1^{RISC}: 3 \), \( Q_2^{RISC}: 3 \ (n = 2) \), \( Q_3^{RISC}: 7 \ (s = 7) \), \( Q_3A^{RISC}: YES \)

This gives \( x = 4 \) and \( l = 10 \) with the coding shown below\(^5\):

![Coding Diagram]

In this case up to 16 opcodes are supported, each of which supports either coding A (i.e. A0 and A1), coding B or coding C (which supports conditional branch).

This is just one of many possible codings (this one is loosely based on the SPARC instruction coding).

\(^5\)note that in this example short and long immediates (simm7 and simm10) are signed numbers
### Q5<sub>RISC</sub>

**What Instructions will we support?**

<table>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
<th>Mnemonic</th>
<th>Function</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>Add</td>
<td>ADD</td>
<td>Subtract</td>
<td>SUB</td>
</tr>
<tr>
<td></td>
<td>Add with Carry</td>
<td>ADDX</td>
<td>Subtract with Borrow</td>
<td>SUBX</td>
</tr>
<tr>
<td>Logic</td>
<td>Bitwise AND</td>
<td>AND</td>
<td>Bitwise OR</td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>Bitwise Exclusive OR</td>
<td>XOR</td>
<td>Logical Shift Right</td>
<td>LSR</td>
</tr>
<tr>
<td>Data Movement</td>
<td>Load</td>
<td>LD</td>
<td>Store</td>
<td>ST</td>
</tr>
<tr>
<td></td>
<td>Set High</td>
<td>SETHI</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- a complete set of common arithmetic and logical functions
  (note that multiply is too complex to be included, while shift left is accomplished by adding a number to itself).

- all arithmetic and logic functions update the condition flags:
  Zero (Z) is set if the result is zero and cleared otherwise.
  Carry (C) is updated by ADD/ADDX/SUB/SUBX/LSR and is set to zero by other logical instructions.

---

6This set has been chosen to match example coding #3
## Basic Processor Design – RISC

### Q5<sub>RISC</sub>

**What Instructions will we support?**

<table>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
<th>Mnemonic</th>
<th>SubFunction</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Transfer</td>
<td>Branch if equal (Z==1)</td>
<td>BEQ</td>
<td>Branch if not equal (Z==0)</td>
<td>BNE</td>
</tr>
<tr>
<td></td>
<td>Branch if carry set (C==1)</td>
<td>BCS</td>
<td>Branch if carry clear (C==0)</td>
<td>BCC</td>
</tr>
<tr>
<td></td>
<td>Branch and Link</td>
<td>BAL</td>
<td>Jump and Link</td>
<td>JMPL</td>
</tr>
</tbody>
</table>

- This processor (like the SPARC on which it is based) uses condition code flags (C,Z) to support conditional branch\(^7\).
- Branch instructions are PC relative with a limited range (+511/-512).
- BAL and JMPL support subroutine call by saving the old PC value in a link register. If R0 (dummy register) is specified as the link register, then BAL can be used as a simple unconditional PC relative branch while JMPL can be used for jumping a long way or for returning from a subroutine.

\(^7\)Other RISC processors such as MIPS save some complexity by not supporting condition code flags

1010
Basic Processor Design – RISC

### Q5A<sub>RISC</sub>

**Define Assembly Language Syntax**<sup>8</sup> and Semantics.

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Mnemonic</th>
<th>Format</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>A</td>
<td>ADD Rs1,Op2,Rd</td>
<td>Rd ← Rs1 + Op2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>where Op2 is either</strong></td>
<td>Rs2 or simm7</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>A</td>
<td>SUB Rs1,Op2,Rd</td>
<td>Rd ← Rs1 - Op2</td>
<td></td>
</tr>
<tr>
<td>ADDX</td>
<td>A</td>
<td>ADDX Rs1,Op2,Rd</td>
<td>Rd ← Rs1 + Op2 + C</td>
<td></td>
</tr>
<tr>
<td>SUBX</td>
<td>A</td>
<td>SUBX Rs1,Op2,Rd</td>
<td>Rd ← Rs1 - Op2 - C</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logic</th>
<th>Mnemonic</th>
<th>Format</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>A</td>
<td>AND Rs1,Op2,Rd</td>
<td>Rd ← Rs1 &amp; Op2</td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>A</td>
<td>OR Rs1,Op2,Rd</td>
<td>Rd ← Rs1</td>
<td>Op2</td>
</tr>
<tr>
<td>XOR</td>
<td>A</td>
<td>XOR Rs1,Op2,Rd</td>
<td>Rd ← Rs1 ^ Op2</td>
<td></td>
</tr>
<tr>
<td>LSR</td>
<td>A</td>
<td>LSR Rs1,Rd</td>
<td>Rd ← Rs1 &gt;&gt;1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Movement</th>
<th>Mnemonic</th>
<th>Format</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>A</td>
<td>LD [Rs1+Op2],Rd</td>
<td>Rd ← mem(Rs1 + Op2)</td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>A</td>
<td>ST Rd,[Rs1+Op2]</td>
<td>mem(Rs1 + Op2) ← Rd</td>
<td></td>
</tr>
<tr>
<td>SETHI</td>
<td>B</td>
<td>SETHI simm10,Rd</td>
<td>Rd ← simm10 &lt;&lt;6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control Transfer</th>
<th>Mnemonic</th>
<th>Format</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ</td>
<td>C</td>
<td>BEQ simm10</td>
<td>if (Z == 1) then PC ← PC + simm10</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>C</td>
<td>BNE simm10</td>
<td>if (Z == 0) then PC ← PC + simm10</td>
<td></td>
</tr>
<tr>
<td>BCS</td>
<td>C</td>
<td>BCS simm10</td>
<td>if (C == 1) then PC ← PC + simm10</td>
<td></td>
</tr>
<tr>
<td>BCC</td>
<td>C</td>
<td>BCC simm10</td>
<td>if (C == 0) then PC ← PC + simm10</td>
<td></td>
</tr>
<tr>
<td>BAL</td>
<td>B</td>
<td>BAL simm10,Rd</td>
<td>Rd ← PC; PC ← PC + simm10</td>
<td></td>
</tr>
<tr>
<td>JMPL</td>
<td>A</td>
<td>JMPL Rs1+Op2,Rd</td>
<td>Rd ← PC; PC ← Rs1 + Op2</td>
<td></td>
</tr>
</tbody>
</table>

<sup>8</sup>operand order follows SPARC convention  
*marked commands update the flags (C,Z)
Basic Processor Design – RISC

Q5B<sub>RISC</sub>
What Opcodes will be Assigned?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td>ADD</td>
<td>ADDX</td>
<td>AND</td>
<td>OR</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>SUB</td>
<td>SUBX</td>
<td>XOR</td>
<td>LSR</td>
</tr>
<tr>
<td>11</td>
<td>Cond[0]</td>
<td>LD</td>
<td>ST</td>
<td>–</td>
<td>JMPL</td>
</tr>
<tr>
<td>10</td>
<td>Cond[1]</td>
<td>SETHI</td>
<td>–</td>
<td>B...</td>
<td>BAL</td>
</tr>
</tbody>
</table>

Instructions are grouped so that decoding is simple.

Unfortunately this example processor does not meet your specification...
Basic Processor Design

Consider CISC\(^9\) features that might be included in the design.

- **Variable instruction length**
  - allows full 16-bit values to be included in your instructions

- **Ability to act on data in memory**
  - gives reduced instruction count

- **Registers with specific roles**
  - e.g. 8 registers but only 2 may act as address registers for load and store - leads to shorter register address fields

- **More addressing modes**
  - e.g. store with pre-decrement or load with post-increment

- **More complex instructions**
  - e.g. `CALL` and `RETURN` instructions which store and retrieve the return address on the stack

\(^9\)CISC = Complex Instruction Set Computer