RISC vs. CISC Instruction Sets
15-740/18-740

Sept. 10, 2009

Topics
- Alpha instruction set

Alpha Processors

Reduced Instruction Set Computer (RISC)
- Simple instructions with regular formats
- Key Idea: make the common case fast!
  - infrequent operations can be synthesized using multiple instructions

Assumes compiler will do optimizations
- e.g., scalar optimization, register allocation, scheduling, etc.

A 2nd Generation RISC Instruction Set Architecture
- Designed for superscalar processors (i.e. >1 inst per cycle)
  - avoids some of the pitfalls of earlier RISC ISAs (e.g., delay slots)
- Designed as a 64-bit ISA from the start

Very high performance machines in their day

32 General Purpose Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>s6, fp</td>
<td>Frame pointer, callee saved</td>
</tr>
<tr>
<td>a6</td>
<td>Return value from integer functions</td>
</tr>
<tr>
<td>t10</td>
<td>Temporaries (not preserved across procedure calls)</td>
</tr>
<tr>
<td>s11</td>
<td>Callee saved</td>
</tr>
<tr>
<td>s12</td>
<td>AT</td>
</tr>
<tr>
<td>s13</td>
<td>gp</td>
</tr>
<tr>
<td>s14</td>
<td>sp</td>
</tr>
<tr>
<td>s15</td>
<td>Always zero</td>
</tr>
<tr>
<td>a2</td>
<td>Integer arguments</td>
</tr>
<tr>
<td>a1</td>
<td>Temporaries</td>
</tr>
<tr>
<td>a0</td>
<td>Return address</td>
</tr>
<tr>
<td>ra</td>
<td>Current proc addr or Temp</td>
</tr>
</tbody>
</table>

Instruction Formats

Arithmetic Operations:
- all register operands
  - addq $1, $7, $5
- with a literal operand
  - addq $1, 15, $5

Branches:
- a single source register
  - bne $1, label

Jumps:
- one source, one dest reg
  - jsr $26, $1, hint

Loads & Stores:
- ldq $1, 16($30)
Returning a Value from a Procedure

C Code

```c
long int test2(long int x, long int y)
{
    return (x+x+x) - (y+y+y);
}
```

Compiled to Assembly

```assembly
.globl test2
.globl test2
.test2:
.frame $30,0,$26,0
.prologue
.addq $16,$16,$1
.addq $16,$16,$1
.addq $17,$17,$0
.addq $0,$17,$0
.subq $0,$17,$0
.addq $0,$17,$0
.addq $0,$17,$0

ret $31,($26),1
```

Place result in $0

Pointer Examples

C Code

```c
long int test2(long int x, long int y)
{
    return (x+x+x) - (y+y+y);
}
```

Annotated Assembly

```assembly
.globl test2
.globl test2
.test2:
.frame $30,0,$26,0
.prologue
.addq $16,$16,$1
.addq $16,$16,$1
.addq $17,$17,$0
.addq $0,$17,$0
.subq $0,$17,$0
.addq $0,$17,$0
.addq $0,$17,$0
.addq $0,$17,$0

ret $31,($26),1
```

Array Indexing

C Code

```c
long int arefl(long int a[], long int i)
{
    return a[i];
}
```

Annotated Assembly

```assembly
.globl arefl
.arefl:
.addq $17,$16,$17 # $17 = 8*i + &a[0]
.idq $0,0($16) # return val = a[0]
.addq $0,0($17) # return val = a[i]
.addq $0,0($17) # return val = a[i]
.addq $0,0($17) # return val = a[i]
.addq $0,0($17) # return val = a[i]

ret $31,($26),1 # return
```

Branches

Conditional Branches

- **bCond Ra, label**
  - **Cond**: branch condition, relative to zero
    - **beq**: Equal
      - $Ra == 0$
    - **bne**: Not Equal
      - $Ra != 0$
    - **bgt**: Greater Than
      - $Ra > 0$
    - **bge**: Greater Than or Equal
      - $Ra >= 0$
    - **blt**: Less Than
      - $Ra < 0$
    - **ble**: Less Than or Equal
      - $Ra <= 0$

Unconditional Branches

- **br label**
Conditional Branches

Comparison Instructions

- **Format:**  
  \texttt{cmp Cond Ra, Rb, Rc}
  
  - **Cond:** comparison condition, Ra relative to Rb
    
    \texttt{cmpeq}  \hspace{1cm} \text{Equal}  \hspace{1cm} \text{Rc} = (Ra == Rb)
    
    \texttt{cmplt}  \hspace{1cm} \text{Less Than}  \hspace{1cm} \text{Rc} = (Ra < Rb)
    
    \texttt{cmple}  \hspace{1cm} \text{Less Than or Equal}  \hspace{1cm} \text{Rc} = (Ra \leq Rb)
    
    \texttt{cmpult}  \hspace{1cm} \text{Unsigned Less Than}  \hspace{1cm} \text{Rc} = (uRa < uRb)
    
    \texttt{cmpule}  \hspace{1cm} \text{Unsigned Less Than or Equal}  \hspace{1cm} \text{Rc} = (uRa \leq uRb)

\begin{verbatim}
long int condbr(long int x, long int y){
    long int v = 0;
    if (x > y) v = x+x+x+y;
    return v;
}
\end{verbatim}

C Code Annotated Assembly

\begin{verbatim}
long int
condbr(long int x, long int y) {
    long int v = 0;
    if (x > y) v = x+x+x+y;
    return v;
}
\end{verbatim}

Conditional Move Instructions

Motivation:
- conditional branches tend to disrupt pipelining & hurt performance

Basic Idea:
- conditional moves can replace branches in some cases
  - avoids disrupting the flow of control

Mechanism:
  \texttt{cmov Cond Ra, Rb, Rc}
  
  - **Cond:** comparison condition, Ra compared with zero
  - same conditions as a conditional branch (eq, ne, gt, ge, lt, le)
  
  - if (Ra Cond zero), then copy Rb into Rc

Psuedo-code example:

\begin{verbatim}
if (x > 0) z = y;
\Rightarrow
cmovgt x, y, z
\end{verbatim}

Jumps

Characteristics:
- transfer of control is unconditional
- target address is specified by a register

Format:

\texttt{jmp Ra, (Rb), Hint}

- Rb contains the target address
- for now, don’t worry about the meaning of Ra or “Hint”
- synonyms for jmp: \texttt{jsr, ret}

\begin{verbatim}
caller:
  \ldots
  0x800 \texttt{bsr} $26, \textit{callee}
\textit{callee}:
  \ldots
  0x8ff \texttt{ret} $31, ($26), 1
\end{verbatim}

Procedure Calls & Returns

Maintain the return address in a special register ($26)

Procedure call:

- \texttt{bsr} $26, label
  - save return addr in $26, branch to label
- \texttt{jsr} $26, (Ra)
  - save return addr in $26, jump to address in Ra

Procedure return:

- \texttt{ret} $31, ($26)
  - jump to address in $26

\begin{verbatim}
caller: \ldots
  0x800 \texttt{bsr} $26, \textit{callee}  \texttt{# save return addr (0x804) in}
  0x804 \texttt{...  \texttt{# \# jumps to callee}
\textit{callee}: \ldots
  0xe018 \texttt{bsr} $31,5,50  \texttt{# return value = 5}
  0xe01c \texttt{ret} $31, ($26), 1  \texttt{# jump to addr in $26}
\end{verbatim}