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

Sept. 13, 2012

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.
- ISA designed for compilers, not assembly language programmers

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>Usage</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>Return value from</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>integer functions</td>
<td></td>
</tr>
<tr>
<td>t0</td>
<td></td>
<td>61</td>
</tr>
<tr>
<td>t1</td>
<td></td>
<td>62</td>
</tr>
<tr>
<td>t2</td>
<td></td>
<td>63</td>
</tr>
<tr>
<td>t3</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>t4</td>
<td></td>
<td>65</td>
</tr>
<tr>
<td>t5</td>
<td></td>
<td>66</td>
</tr>
<tr>
<td>t6</td>
<td></td>
<td>67</td>
</tr>
<tr>
<td>t7</td>
<td></td>
<td>68</td>
</tr>
<tr>
<td>s0</td>
<td></td>
<td>69</td>
</tr>
<tr>
<td>s1</td>
<td>Callee saved</td>
<td>70</td>
</tr>
<tr>
<td>s2</td>
<td></td>
<td>71</td>
</tr>
<tr>
<td>s3</td>
<td></td>
<td>72</td>
</tr>
<tr>
<td>s4</td>
<td></td>
<td>73</td>
</tr>
<tr>
<td>s5</td>
<td></td>
<td>74</td>
</tr>
<tr>
<td>s6,fp</td>
<td>Frame pointer, or</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>callee saved</td>
<td></td>
</tr>
<tr>
<td>a0</td>
<td>Temporaries (not preserved</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>across procedure calls)</td>
<td></td>
</tr>
<tr>
<td>a1</td>
<td></td>
<td>81</td>
</tr>
<tr>
<td>a2</td>
<td></td>
<td>82</td>
</tr>
<tr>
<td>a3</td>
<td></td>
<td>83</td>
</tr>
<tr>
<td>a4</td>
<td></td>
<td>84</td>
</tr>
<tr>
<td>a5</td>
<td></td>
<td>85</td>
</tr>
<tr>
<td>a6</td>
<td></td>
<td>86</td>
</tr>
<tr>
<td>a7</td>
<td></td>
<td>87</td>
</tr>
<tr>
<td>a8</td>
<td></td>
<td>88</td>
</tr>
<tr>
<td>a9</td>
<td></td>
<td>89</td>
</tr>
<tr>
<td>a10</td>
<td></td>
<td>90</td>
</tr>
<tr>
<td>a11</td>
<td></td>
<td>91</td>
</tr>
<tr>
<td>a12</td>
<td></td>
<td>92</td>
</tr>
<tr>
<td>a13</td>
<td></td>
<td>93</td>
</tr>
<tr>
<td>a14</td>
<td></td>
<td>94</td>
</tr>
<tr>
<td>a15</td>
<td></td>
<td>95</td>
</tr>
<tr>
<td>r0</td>
<td></td>
<td>96</td>
</tr>
<tr>
<td>r1</td>
<td></td>
<td>97</td>
</tr>
<tr>
<td>r2</td>
<td></td>
<td>98</td>
</tr>
<tr>
<td>r3</td>
<td></td>
<td>99</td>
</tr>
<tr>
<td>r4</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>r5</td>
<td></td>
<td>101</td>
</tr>
<tr>
<td>r6</td>
<td></td>
<td>102</td>
</tr>
<tr>
<td>r7</td>
<td></td>
<td>103</td>
</tr>
<tr>
<td>r8</td>
<td></td>
<td>104</td>
</tr>
<tr>
<td>r9</td>
<td></td>
<td>105</td>
</tr>
<tr>
<td>r10</td>
<td></td>
<td>106</td>
</tr>
<tr>
<td>r11</td>
<td></td>
<td>107</td>
</tr>
<tr>
<td>r12</td>
<td></td>
<td>108</td>
</tr>
<tr>
<td>r13</td>
<td></td>
<td>109</td>
</tr>
<tr>
<td>r14</td>
<td></td>
<td>110</td>
</tr>
<tr>
<td>r15</td>
<td></td>
<td>111</td>
</tr>
<tr>
<td>r16</td>
<td></td>
<td>112</td>
</tr>
<tr>
<td>r17</td>
<td></td>
<td>113</td>
</tr>
<tr>
<td>r18</td>
<td></td>
<td>114</td>
</tr>
<tr>
<td>r19</td>
<td></td>
<td>115</td>
</tr>
<tr>
<td>r20</td>
<td></td>
<td>116</td>
</tr>
<tr>
<td>r21</td>
<td></td>
<td>117</td>
</tr>
<tr>
<td>r22</td>
<td></td>
<td>118</td>
</tr>
<tr>
<td>r23</td>
<td></td>
<td>119</td>
</tr>
<tr>
<td>r24</td>
<td></td>
<td>120</td>
</tr>
<tr>
<td>r25</td>
<td></td>
<td>121</td>
</tr>
<tr>
<td>r26</td>
<td></td>
<td>122</td>
</tr>
<tr>
<td>r27</td>
<td></td>
<td>123</td>
</tr>
<tr>
<td>r28</td>
<td></td>
<td>124</td>
</tr>
<tr>
<td>r29</td>
<td></td>
<td>125</td>
</tr>
<tr>
<td>r30</td>
<td></td>
<td>126</td>
</tr>
<tr>
<td>r31</td>
<td></td>
<td>127</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

long int test2(long int x, long int y) {
    return (x+y+y); // Place result in $0
}
```

**Compiled to Assembly**

```
.long int
.globl test2
.etype test2
.test2:
.align 3
.frame $30,0,$26,0
.prologue 0
.add $16,$16,$1
.add $1,$16,$1
.add $17,$17,$0
.addq $0,$17,$0
.subq $1,$0,$2
.ret $31,($26),1
.end test2
```

### Array Indexing

**C Code**

```c
long int

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

**Annotated Assembly**

```
arefl:
.saddq $17,$16,$17 # $17 = 8*i + &a[0]
.ldq $0,0($17) # return val = a[i]
.ret $31,($26),1 # return
```

### Pointer Examples

**C Code**

```c
long int

long int iaddp(long int *xp, long int *yp) {
    int x = *xp;
    int y = *yp;
    return x + y;
}
```

**Annotated Assembly**

```
iaddp:
.ldq $1,0($16) # $1 = *xp
.ldq $0,0($17) # $0 = *yp
.addq $1,$0,$0 # return with a value x+y
```

```c
void

void incr(long int *sum, long int v) {
    long int old = *sum;
    long int new = old + v;
    *sum = new;
}
```

**Annotated Assembly**

```
incr:
.ldq $1,0($16) # $1 = *sum
.addq $1,$17,$1 # $1 += v
.stq $1,0($16) # *sum = $1
.ret $31,($26),1 # return
```

### Branches

#### Conditional Branches

- `bCond Ra, label`
  - `cond`: branch condition, relative to zero
    - `beq`: Equal
    - `bne`: Not Equal
    - `bgt`: Greater Than
    - `bge`: Greater Than or Equal
    - `blt`: Less Than
    - `ble`: Less Than or Equal
  - `Ra`: register value
  - `label`: branch label

  - Register value is typically set by a comparison instruction

#### Unconditional Branches

- `br label`
Conditional Branches

Comparison Instructions
- **Format:** `cmp <Cond> Ra, Rb, Rc`
- `Cond:` comparison condition, Ra relative to Rb
  - `cmpeq`: Equal
    - `Rc = (Ra == Rb)`
  - `cmplt`: Less Than
    - `Rc = (Ra < Rb)`
  - `cmpue`: Less Than or Equal
    - `Rc = (Ra <= Rb)`
  - `cmple`: Unsigned Less Than
    - `Rc = (Ra < uRb)`
  - `cmple`: Unsigned Less Than or Equal
    - `Rc = (Ra <= uRb)`

C Code
```c
long int condbr(long int x, long int y)
{
  long int v = 0;
  if (x > y)
    v = x + x + y;
  return v;
}
```

Annotated Assembly
```assembly
    long int condbr(long int x, long int y) {
        long int v = 0;
        if (x > y) { v = x + x + y; }
        return v;
    }
```

C Code
```c
condbr:
```
Annotated Assembly
```assembly
condbr:
    bsc $31, $31, 0, 0 # v = 0
    cmpbl $16, $17, 0 # x > y?
    bl $1, $45 # if so, branch
    addq $16, $16, 0 # v += x
    addq $0, $16, 0 # v += x
    addq $0, $17, 0 # v += y
    ret $31, ($26), 1 # return v
```

Jumps

Characteristics:
- Transfer of control is unconditional
- Target address is specified by a register

Format:
```assembly
    jmp Ra, (Rb), Hint
```
- `Rb` contains the target address
- For now, don’t worry about the meaning of `Ra` or “Hint”
- Synonyms for jmp: `jsr`, `ret`

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:
```assembly
cmov <Cond> Ra, Rb, Rc
```
- `Cond:` comparison condition, Ra is 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:
```c
if (x > 0) z = y;
    => cmovgt x, y, z
```

Procedure Calls & Returns

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

Procedure call:
```c
    bsr $26, label
    jsr $26, (Ra)
```
- Save return address in $26, branch to label
- Save return address in $26, jump to address in Ra

Procedure return:
```c
    ret $31, ($26)
```
- Jump to address in $26

C Code
```c
long int caller()
{
    return callee();
}
long int callee()
{
    return 5L;
}
```

Annotated Assembly
```assembly
caller:
```
RISC Ideas Today

- RISC principles can be seen in Intel processors:
  - x86 translated to uops (micro-ops) in Pentium Pro and successors
  - x86-64 has more registers, register-based calling convention, etc.

- Streamlined design makes sense when designing an ISA from scratch
  - processors that don’t need to run legacy codes, etc.

- How does optimizing for power (rather than performance) change the design goals?
  - fewer registers?
  - more compact code size?