Assembly Tools

Calling Conventions

Review: $\beta$ Model of Computation

Fetch/Execute Loop:
- Fetch $<\text{PC}>$
- $\text{PC} \leftarrow <\text{pc}> + 1$
- Execute fetched instruction
- Repeat!
Review: BETA Instructions

Two 32-bit Instruction Formats:

Unused Rc Ra OPCODE Rb
OPCODE Ra 16 bit Constant Rc

Review: β ALU Operations

What the machine sees (32-bit instruction word):

What we prefer to see: symbolic ASSEMBLY LANGUAGE

ADD(ra, rb, rc) rc ← <ra> + <rb>
"Add the contents of ra to the contents of rb; store the result in rc"

SIMILARLY FOR:
- SUB, SUBC
- (optional)
- MUL, MULC
- DIV, DIVC

BITWISE LOGIC:
- AND, ANDC
- OR, ORC
- XOR, XORC

SHIFTS:
- SHL, SHR, SAR
  (shift left, right;
  shift arith right)

COMPARES
- CMPEQ, CMPLT
- CMPLE

ADD(ra, const, rc) rc ← <ra> + sext(const)
"Add the contents of ra to const; store the result in rc"
**Review: β Loads & Stores**

LD(ra, C, rc) \( \rightarrow \) \( rc \leftarrow <\text{Mem}[<ra>+\text{sext}(C)]> \)

“Fetch into \( rc \) the contents of the data memory location whose address is the contents of \( ra \) plus \( C \)”

ST(rc, C, ra) \( \rightarrow \) \( \text{Mem}[<ra>+\text{sext}(C)] \leftarrow <rc> \)

“Store the contents of \( rc \) into the data memory location whose address is the contents of \( ra \) plus \( C \)”

**NO BYTE ADDRESSES:** only 32-bit word accesses are supported.

This is similar to how Digital Signal Processors work.

It is somewhat unusual for general purpose processors, which are usually byte (8 bit) addressed.

**Review: β Branches**

**Conditional:** \( rc = <\text{PC}>+1; \) then

BRNZ(ra, label, rc) if \(<ra> \) nonzero then

\( PC \leftarrow <\text{PC}> + \text{displacement} \)

BRZ(ra, label, rc) if \(<ra> \) zero then

\( PC \leftarrow <\text{PC}> + \text{displacement} \)

**Unconditional:** \( rc = <\text{PC}>+1; \) then

BRZ(r31, label, rc) \( PC \leftarrow <\text{PC}> + \text{displacement} \)

**Indirect:** \( rc = <\text{PC}>+1; \) then

JMP(ra, rc) \( PC \leftarrow <ra> \)

*Note:* “displacement” is coded as a CONSTANT in a field of the instruction!
Review: Iterative Optimized Factorial

;assume n = 20, val = 1

(define (fact-iter n val)
  (if (= n 0)
      val
      (fact-iter (- n 1) (* n val)))

LD(n, r1) ; n in r1
LD(val, r3) ; val in r3
BRZ(r1, done)

loop:
  MUL(r1, r3, r3)
  SUBC(r1, 1, r1)
  BRNZ(r1, loop)

done:
  ST(r1, n) ; new n
  ST(r3, val) ; new val

Language Tools

The Assembler

STREAM of Words to be loaded into memory

Symbolic SOURCE text file
Translator program
Binary Machine Language

Textual Macro Pre-Processor
Symbol Memory + Expression Evaluator

How Computers Work Lecture 2 Page 7

How Computers Work Lecture 2 Page 8
Macros

Macros are parameterized abbreviations that when invoked cause TEXTUAL SUBSTITUTION

| Macro to generate 4 consecutive numbers: |
| .macro consec4(n) n n+1 n+2 n+3 |

| Invocation of above macro: |
| consec4(37) |

Is translated into:

37 37+1 37+2 37+3

Some Handy Macros

| BETA Instructions: |
| ADD(ra, rb, rc) | rc ← <ra> + <rb> |
| ADDC(ra, const, rc) | rc ← <ra> + const |
| LD(ra, C, rc) | rc ← <C + <ra>> |
| ST(rc, C, ra) | C + <ra> ← <rc> |
| LD(C, rc) | rc ← <C> |
| ST(rc, C) | C ← <ra> |
Constant Expression Evaluation

37  -3  255 \hspace{1cm} \textit{decimal (default)};
0b100101 \hspace{1cm} \textit{binary (0b prefix)};
0x25 \hspace{1cm} \textit{hexadecimal (0x prefix)};

Values can also be expressions; eg:

37+0b10-0x10 24-0x1 4*0b110-1 0xF7&0xF

\textit{generates 4 words of binary output, each with the value 23}

Symbolic Memory

\textit{We can define SYMBOLS:}

\begin{align*}
x &= 1 & | & 1 \\
y &= x + 1 & | & 2
\end{align*}

\textit{Which get remembered by the assembler. We can later use them instead of their values:}

\texttt{ADDC(x, 37, y) | R2 \leftarrow <R1> + 37}
How Are Symbols Different Than Macros?

• Answer:
  – A **macro’s** value at any point in a file is the last previous value it was assigned.
      • Macro evaluation is purely textual substitution.
  – A **symbol’s** value throughout a file is the very last value it is assigned in the file.
      • Repercussion: we can make “forward” references to symbols not yet defined.
      • Implementation: the assembler must first look at the entire input file to define all symbols, then make another pass substituting in the symbol values into expressions.

Dot, Addresses, and Branches

*Special symbol “.” (period) changes to indicate the address of the next output byte.*

*We can use .* to define branches to compute RELATIVE address field:*

```assembly
.macro BRNZ(ra, loc) betaopc(0x1E, ra, (loc-.)-1, r31)

loop = .
  ADDC(r0, 1, r0)  | “loop” is here...
  ...  
  BRNZ(r3, loop)   | Back to addc instr.
```

How Computers Work Lecture 2 Page 13

How Computers Work Lecture 2 Page 14
Address Tags

\( x: \) is an abbreviation for \( x = . \) -- leading to programs like

\[
x: 0
\]

buzz:
\[
\begin{align*}
LD(x, r0) & \quad \text{do } \{ x = x-1; \} \\
ADDC(r0, -1, r0) & \\
ST(r0, x) & \\
BRNZ(r0, buzz) & \quad \text{while } (x > 0); \\
\ldots
\end{align*}
\]

Macros Are Also Distinguished by Their Number of Arguments

We can extend our assembly language with new macros. For example, we can define an UNCONDITIONAL BRANCH:

\[
\begin{align*}
\text{BR(label, rc)} & \quad \text{rc} \leftarrow \text{<PC>+4}; \text{then} \\
\text{PC} & \leftarrow \text{<PC> + displacement} \\
\text{BR(label)} & \quad \text{PC} \leftarrow \text{<PC> + displacement}
\end{align*}
\]

by the definitions

\[
\begin{align*}
\text{.macro BR(lab, rc)} & \quad \text{BRZ (r31,lab, rc)} \\
\text{.macro BR(lab)} & \quad \text{BR(lab,r31)}
\end{align*}
\]
OK, How about recursive fact?

(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1))))
)

int fact(int n)
{
    if (n == 0)
        return (1);
    else
        return (n * fact(n-1));
}

R28 Convention: We call it the **Linkage Pointer**
(just like scheme RML's `continue` register)

Does this really work?

```c
int fact(int n)
{
    if (n == 0)
        return (1);
    else
        return (n * fact(n-1));
}
```

We need a STACK !!!!
Recursion...

fact(3) ...

<table>
<thead>
<tr>
<th>n=0</th>
<th>n=1</th>
<th>n=2</th>
<th>n=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>fact(0)</td>
<td>fact(1)</td>
<td>fact(2)</td>
<td>fact(3)</td>
</tr>
</tbody>
</table>

Caller

Stack Implementation

one of several possible implementations

Conventions:
- Builds UP on push
- SP points to first UNUSED location.

To push <x>:
\[
\text{sp} \leftarrow \text{sp} + 1;
\text{Mem}[\langle \text{sp} \rangle - 1] \leftarrow \langle x \rangle
\]

To pop() a value into x:
\[
\text{x} \leftarrow \text{Mem}[\langle \text{sp} \rangle - 1]
\text{sp} \leftarrow \text{sp} - 1;
\]
Support Macros for Stacks

\[ \text{sp} = r29 \]

**push (rx)** - pushes 32-bit value onto stack.

\[
\begin{align*}
    \text{ADDC} & (s_p, 1, s_p) \\
    \text{ST} & (r_x, -1, s_p)
\end{align*}
\]

**pop (rx)** - pops 32-bit value into rx.

\[
\begin{align*}
    \text{LD} & (r_x, -1, s_p) \\
    \text{ADDC} & (s_p, -1, s_p)
\end{align*}
\]

**allocate (k)** - reserve k WORDS of stack

\[
\begin{align*}
    \text{ADDC} & (s_p, k, s_p)
\end{align*}
\]

**deallocate (k)** - give back k WORDS

\[
\begin{align*}
    \text{SUBC} & (s_p, k, s_p)
\end{align*}
\]

---

The Stack

- **STACK** as central storage-management mechanism...
  - SIMPLE, EFFICIENT to implement using contiguous memory and a "stack pointer"
  - ORIGINAL USE: subroutine return points - push/pop STACK DISCIPLINE follows natural order of call/return nesting.
  - EXPANDED USE: "automatic" or "dynamic" allocation of local variables.
- **REVOLUTIONARY IMPACT:**
  - ALL modern machines, to varying extents;
  - ALL modern languages

STACK DISCIPLINE: 
most recently allocated location is next location to be deallocated.

IMPACT: BLOCK STRUCTURE.
Call / Return Linkage

lp = r28  
sp = r29

Using these macros and r28 as a “linkage pointer”, we can call f by:

\[
\text{BR}(f, \ lp)  
\]

And code procedure f like:

\[
\begin{align*}
f: & \quad \text{PUSH}(lp) \quad | \quad \text{SAVE} \ lp  \\
& \quad <\text{perform computation}> \quad | \quad (\text{may trash} \ lp)  \\
& \quad \text{POP}(lp) \quad | \quad \text{RESTORE} \ lp  \\
& \quad \text{JMP}(lp) \quad | \quad \text{Return to caller} \\
\text{BR}(f, \ lp) & 
\end{align*}
\]

Recursion with Register-Passed Arguments

| Compute Fact(n) \n| n passed in r1, result returned in r0

\[
\begin{align*}
fact: & \quad \text{PUSH}(lp) \quad | \quad \text{Save linkage pointer}  \\
& \quad \text{BRZ}(r1, fact1) \quad | \quad \text{terminal case?}  \\
& \quad \text{PUSH}(r1) \quad | \quad \text{Save n,}  \\
& \quad \text{ADDC}(r1,-1,r1) \quad | \quad \text{compute fact(n-1).}  \\
& \quad \text{BR}(\text{fact}, \ lp) \quad | \quad \text{recursive call to fact.}  \\
& \quad \text{POP}(r1) \quad | \quad \text{restore arg,}  \\
& \quad \text{MUL}(r1,r0,r0) \quad | \quad \text{return n*fact(n-1)}  \\
\text{factx:} & \quad \text{POP}(lp) \quad | \quad \text{restore linkage pointer}  \\
& \quad \text{JMP}(lp) \quad | \quad \text{and return to caller.}  \\
\text{fact1:} & \quad \text{MOV}(1,r0) \quad | \quad \text{fact(0) => 1}  \\
& \quad \text{BR}(\text{factx}) \\
\text{.macro MOV(rs,rd) \quad ADD \ (rs, r31, rd)\n}  \\
\text{.macro MOV(rsrc,rdest) \quad ADDC \ (r31, rsrc, rdest)\n}  
\end{align*}
\]
A Generic Stack Frame Structure: The 6.004 Stack Discipline

**Reserved Registers:**
- `bp = r27`. Base ptr, points to 1st local.
- `lp = r28`. Linkage pointer, saved <PC>.
- `sp = r29`. Stack ptr, points to 1st unused word.
- `xp = r30`. Exception pointer, saved <PC>

**Stack Frame**
(a.k.a. Function Activation Record):

**6.004 Stack Discipline Procedure Linkage**

**Calling Sequence:**
- `PUSH(arg n)` | push args, in
  - `...` | RIGHT-TO-LEFT
  - `PUSH(arg1)` | order!
  - `BR(f, lp)` | Call f.
  - `DEALLOCATE(n)` | Clean up!
  - `...` | (returned value now in r0)

**Entry Sequence:**
- `f:`
  - `PUSH(lp)` | Save <LP>, <BP>
  - `PUSH(bp)` | for new calls.
  - `MOV(sp, bp)` | set bp=frame base
  - `ALLOCATE(locals)` | allocate locals
  - `(push other regs)` | preserve regs used
  - `...`

**Return Sequence:**
- `(pop other regs)` | restore regs
  - `MOV(val, r0)` | set return value
  - `MOV(bp, sp)` | strip locals, etc
  - `POP(bp)` | restore linkage
  - `POP(lp)` | (the return <PC>)
  - `JMP(lp)` | return.
Stack Frame Detail

Access to Arguments & Local Variables

To access jth local variable (j >= 0)
LD(BP, (j, rx)
or
ST(rx, j, bp)

To access jth argument (j >= 0):
LD(BP, 3-j, rx)
or
ST(rx, 3-j, bp)

QUESTION: Why push args in REVERSE order???
Procedure Linkage: The Contract

The caller will:
- Push args onto stack, in reverse order.
- Branch to callee, putting return address into lp.
- Remove args from stack on return.

The callee will:
- Perform promised computation, leaving result in r0.
- Branch to return address.
- Leave all regs (except lp, r0) unchanged.

Recursive factorial with stack-passed arguments

(define (fact n)
 (if (= n 0) 1 (* n (fact (- n 1))))
)

fact: PUSH(lp)  | save linkages
PUSH(bp)      
MOV(sp, bp)   | new frame base
PUSH(r1)      | preserve regs
LD(bp, -3, r1) | r0 ← n
BRNZ(r1, big) | if n>0
MOVC(1, r0)   | else return 1;
BR(stn)

big: SUBC(r1, 1, r1)  | r1 ← (n-1)
PUSH(r1) | arg1
BR(fact, lp) | fact(n-1)
DEALLOCATE(1) | (pop arg)
LD(bp, -3, r1) | r0 ← n
MUL(r1, r0, r0) | r0 ← n*fact(n-1)

rtn: POP(r1) | restore regs
MOV(bp, ap) | Why?
PPOP(bp) | restore links
PPOP(lp) | return
JMP(lp) | return.
What did we Learn Today?

• How to call functions
• How to do recursive factorial
• The 6.004 Stack Discipline
• How to retrieve arguments and local variables

Next In Section

• Practice with Stack Discipline

C Tutorial

http://www.csc.lsu.edu/tutorial/ten-commandments/bwk-tutor.html