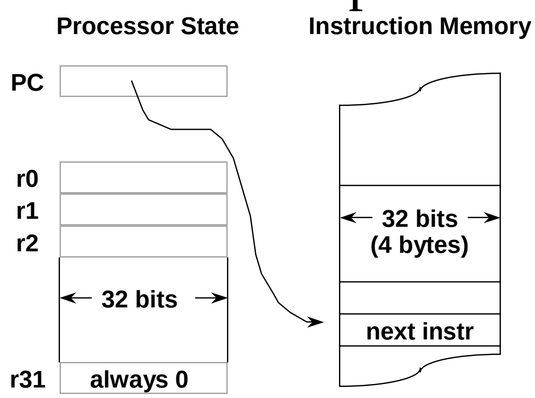
How Computers Work
Lecture 2

Assembly Tools
Calling Conventions

Review: β Model of Computation

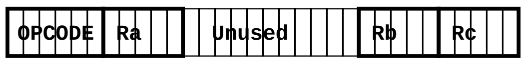


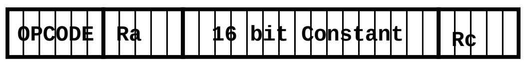
Fetch/Execute Loop:

- **Fetch <PC>**
- **PC ← *<pc> + 1***
- **Execute fetched instruction**
- **Repeat!**

Review: BETA Instructions

Two 32-bit Instruction Formats:

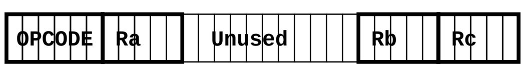




How Computers Work Lecture 2 Page 3

Review: β ALU Operations

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

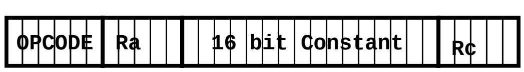


What *we* prefer to see: symbolic ASSEMBLY LANGUAGE

ADD(ra, rb, rc) **rc ← <ra> + <rb>**

"Add the contents of ra to the contents of rb; store the result in rc"

Alternative instruction format:



ADDC(ra, const, rc) **rc ← <ra> + sext(const)**

"Add the contents of ra to const; 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**

How Computers Work Lecture 2 Page 4

Review: β Loads & Stores

LD(ra, C, rc) **rc ← < Mem[<ra> + 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) **Mem[<ra> + sext(C)] ← <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 usual byte (8 bit) addressed

Review: β Branches

Conditional: **rc = <PC>+1; then**

BRNZ(ra, label, rc) **if <ra> nonzero then**
PC <- <PC> + displacement

BRZ(ra, label, rc) **if <ra> zero then**
PC <- <PC> + displacement

Unconditional: **rc = <PC>+1; then**

BRZ(r31, label, rc) **PC <- <PC> + displacement**

Indirect: **rc = <PC>+1; then**

JMP(ra, rc) **PC <- <ra>**

Note: "displacement" is coded as a CONSTANT in a field of the instruction!

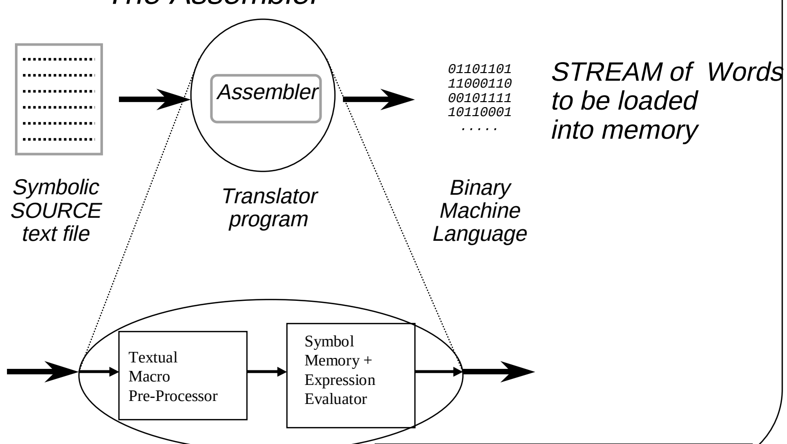
Review: Iterative Optimized Factorial

```
;assume n = 20, val = 1

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

```
n:      20
val:    1

    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



Macros

Macros are parameterized abbreviations that when invoked cause TEXTUAL SUBSTITION

```
| 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`	*decimal (default);*
`0b100101`	*binary (0b prefix);*
`0x25`	*hexadecimal (0x prefix);*

Values can also be expressions; eg:

```
37+0b10-0x10   24-0x1   4*0b110-1   0xF7&0x1F
```

generates 4 words of binary output, each with the value 23

Symbolic Memory

We can define SYMBOLS:

```
x = 1         | 1
y = x + 1     | 2
```

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

```
ADDC(x, 37, y) | R2 ← <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:

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

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

Address Tags

`x:` *is an abbreviation for* `x =.` *-- leading to programs like*

```
x:     0

buzz: LD(x, r0)                   do { x = x-1; }
      ADDC(r0, -1, r0)
      ST(r0, x)
      BRNZ(r0, buzz)              while (x > 0);
      ...
```


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:

BR(label, rc)	**rc ← <PC>+4; then** **PC ← <PC> + displacement**
BR(label)	**PC ← <PC> + displacement**

by the definitions

```
.macro BR(lab, rc) BRZ (r31,lab, rc)


.macro BR(lab)     BR(lab,r31)
```

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));
}
```

Suppose caller:

1. Stores nin agreed-on location (say, r1)

2. Calls fact using, say,

BR(fact, r28)

Then fact:

1. Computes value, leaves (say) in r0.

2. Returns via

JMP(r28, r31)

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

Does this really work?

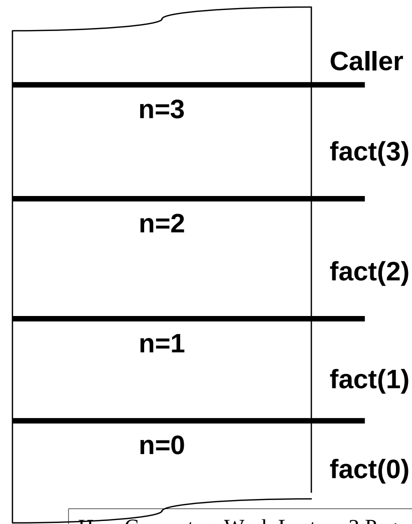
```
int fact(int n)
{
    if (n == 0)
        return (1);
    else
        return (n * fact(n-1));
}
```

```
fact:
        ...

        BR(fact, LP)
        ...
        (put result in r0)
```

We need a STACK ! ! !

Recursion...

fact(3) ...



How Computers Work Lecture 2 Page 19

Stack Implementation

one of several possible implementations

Conventions:

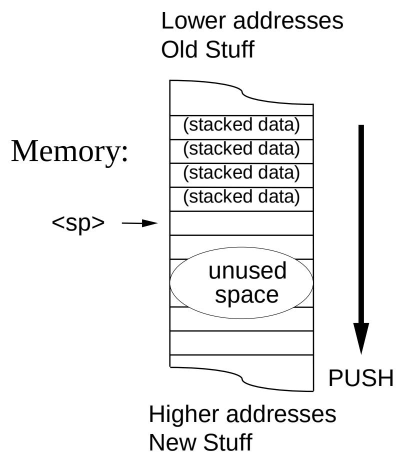
- Builds UP on push
- SP points to first UNUSED location.

To push <x>:

sp ← <sp> + 1;
Mem[<sp> - 1] ← <x>

To pop() a value into x:

x ← <Mem[<sp> - 1]>
sp ← <sp> - 1;



How Computers Work Lecture 2 Page 20

Support Macros for Stacks

`sp = r29`

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

```
ADDC(sp, 1, sp)
ST(rx, -1, sp)
```

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

```
LD(rx, -1, sp)
ADDC(SP, -1, sp)
```

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

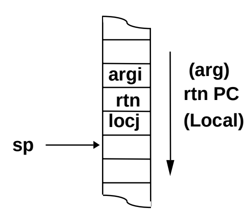
```
ADDC(SP, k, sp)
```

`deallocate(k)` - give back k WORDS

```
SUBC(SP, k, sp)
```


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:

```
BR(f, lp)
```

And code procedure f like:

```
f:      PUSH(lp)                  | SAVE lp
        <perform computation>     | (may trash lp)
        POP(lp)                   | RESTORE lp
        JMP(lp)                   | Return to caller
```


Recursion with Register-Passed Arguments

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

fact:   PUSH(lp)             | Save linkage pointer

        BRZ(r1,fact1)        | terminal case?
        PUSH(r1)             | Save n,
        ADDC(r1,-1,r1)       | compute fact(n-1).
        BR(fact, lp)         | recursive call to fact.
        POP(r1)              | restore arg,
        MUL(r1,r0,r0)        | return n*fact(n-1)

factx:  POP(lp)              | restore linkage pointer
        JMP(lp)              | and return to caller.

fact1:  MOVC(1,r0)           | fact(0) => 1
        BR(factx)

.macro MOV(rsrc,rdest)       ADD (rsrc, r31, rdest)
.macro MOVC(csrc,rdest)      ADDC (r31, csrc, rdest)
```

A Generic Stack Frame Structure: The 6.004 Stack Discipline

RESERVED REGISTERS:

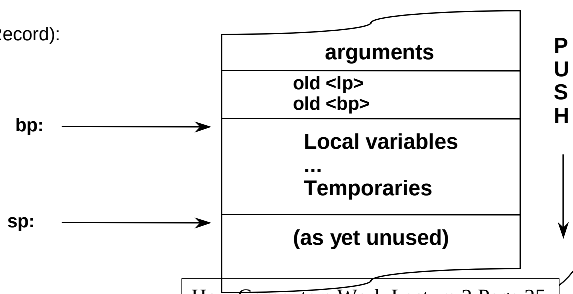
bp = r27. Base ptr, points to 1st local.

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):



How Computers Work Lecture 2 Page 25

6.004 Stack Discipline Procedure Linkage

Calling Sequence:

```
        PUSH(arg_n)          | push args, in
        ...                  | RIGHT-TO-LEFT
        PUSH(arg_1)          | order!
        BR(f, lp)            | Call f.
        DEALLOCATE(n)        | Clean up!
        ...                  | (returned value now in r0)
```

Entry Sequence:

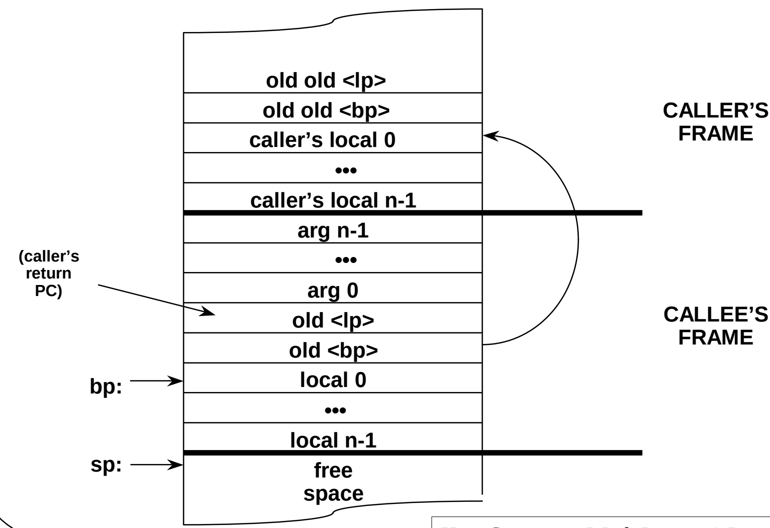
```
f:      PUSH(lp)             | Save <LP>, <BP>
        PUSH(bp)             |   for new calls.
        MOV(sp,bp)           | set bp=frame base
        ALLOCATE(locals)     | allocate locals
        (push other regs)    | preserve regs used
        ...
```

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.
```

How Computers Work Lecture 2 Page 26

Stack Frame Detail



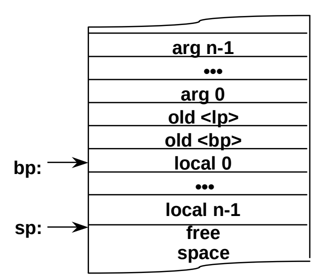
Access to Arguments & Local Variables

To access j^{th} local variable (j >= 0)

LD(BP, (j, rx)

or

ST(rx, j, bp)



To access j^{th} 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(rtn)

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,sp)         | Why?
         POP(bp)            | restore links
         POP(lp)
         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