DEPARTMENT OF
INFORMATION SYSTEMS
AND COMPUTER SCIENCE

Function Calls

Is Infinite Recursion Possible?

Lecture Time!

- Implementing Functions: Macros
- Implementing Functions: Subroutines with Static Allocation
- Implementing Functions: Using a Stack

Writing Functions

- How do we implement a reusable function in Beta assembly?

```
int fact( int n )
{
    int f = 1;
    while( n > 0 )
    {
        f *= n;
        n -= 1;
    }
    return f;
}
```

Writing Functions

- How do we implement a reusable function in Beta assembly?

```
int fact( int n ) // R1 contains n
{
    int f = 1;        // ADDC( R31, 1, R0 )
    while( n > 0 ) // loop: CMPLT( R31, R1, R2 )
    {                 // BF( R2, done, R31 )
        f *= n;       // MUL( R0, R1, R0 )
        n -= 1;       // SUBC( R1, 1, R1 )
    }                 // BR( loop )
    return f;         // done: R0 contains f
}
```

Using Macros (aka Inlining)

- Specify register mapping: input, output, and temp regs.
  - Input and output regs can be parameters of .macro.
  - *Temporary registers must be documented*, so user knows which regs are “clobbered”.
- Macros are expanded at compile-time.
  - Labels CANNOT be used within .macro in BSim.
    - This results in duplicate names.
  - Need to use relative addressing for branching.
    - Note that it is still converted to word displacement in the final 4-byte representation.

Using Macros (aka Inlining)

```
.include beta.uasm               | duh

.macro fact( Ra, Rc ) {          | Rc = fact(<Ra>)
ADDC( R31, 1, Rc )               | Rc=1
MUL( Rc, Ra, Rc )                | loop:
SUBC( Ra, 1, Ra )
BT( Ra, .-8, R31 )               | branch to loop
}                                | note: not error-free

| how to use the macro
main:                            | int main( void )
CMOVE( 6, R1 )                   | ADDC( R31, 6, R1 )
CMOVE( 3, R2 )
fact( R1, R0 )                   | R0=fact(6)
fact( R2, R3 )                   | R3=fact(3)
HALT()
```

Using Macros (aka Inlining)

- Advantages
  - Simple and straightforward
  - Faster than other methods: no overhead
- Disadvantages
  - Compiled binary code grows the more you call macros
    - Did you try running the sample code? What did you notice?
  - Need to be careful about “clobbered” registers
    - What if the macro needed to use a register not included in its arguments for temporary storage?
  - Hard to debug
  - No recursion
- In general, inlining is good for *short* code.
  - Often used by C/C++ compilers for optimizing speed.

Subroutines with Static Allocation

- Only one copy of function code.
- Linkage Pointer (LP=R28) holds return address:
  - From main program, BR(fact,LP) to call.
  - From subroutine, JMP(LP) to return to where the function was called.
- You still need to specify register mapping: input, output, and temp regs.
  - *Input and output regs are now fixed and must be documented* in addition to the temporary regs.

Subroutines with Static Allocation

```
main:
CMOVE( 6, R1 )
CMOVE( 3, R2 )
BR( fact, LP ) | call fact( 6 )
| at this point, R0 contains fact( R1 )
MOVE( R2, R1 )
BR( fact, LP ) | call fact( 3 )
MOVE( R0, R3 )
| at this point, R3 contains fact( R2 )
HALT()

fact: | takes input in R1, puts output in R0
ADDC( R31, 1, R0 )
loop: MUL( R0, R1, R0 )
SUBC( R1, 1, R1 )
BT( R1, loop, R31 ) | note: not error-free
JMP( LP ) | return
```

Subroutines with Static Allocation

- Alternatively, we can put inputs and outputs in main memory instead of registers.
  - Useful when we use more than 32 variables.
  - Or if we don't want to clobber too many registers.
- Just use a few registers, then LD and ST to them as necessary.
  - Let's make code for that now! (and fix the BT issue also)
  - Hint: Documentation should say something like: "takes input in factN, puts output in factF, clobbers R0 and R1".

Subroutines with Static Allocation

- Advantages:
  - Single copy of code, which saves space and makes it easier to debug.
  - Can use memory for storage.
    - Macros can too, but creates many copies and wastes enough space already.
- Disadvantages:
  - Still clobbers registers.
  - No nested function calls.
  - Still no recursion.
- But we're getting somewhere!

Analyzing Recursive Functions

- Consider the following code:

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

- What are the storage requirements?
  – Why can't we use recursion or even nested function calls for subroutines with static allocation?
  – Is there an issue with local variable n? What about the return value?

The Solution: Stacks

- Needs a special pointer, the Stack Pointer (R29).

```
SP=R29
.macro PUSH(rx) {
ADDC(SP, 4, SP)
ST(rx, -4, SP)
}

.macro POP(rx) {
LD(SP, -4, rx)
SUBC(SP, 4, SP)
}

| assumes SP initially contains a high value
| (is there a problem if it contains 0?)
| and these macros are already in beta.uasm!
```

Possible Issue With Pop?

- Push a few values on to the stack, then pop one.
  - What do the macros actually do?
    - You may want to try it out in BSim.
  - Look at the stack closely. Is there a possible issue?
    - More importantly, will it be an issue?
    - Hint: When formatting your hard drive, what is the difference between quick format and full format?
    - Hint #2: While a Java program is still running, what does garbage collection do?

Stack State

- Given a stack's current state:
  - X pushes followed by X pops (assuming # of pops NEVER exceeds # of pushes at any given time) should return the stack to that state.
- Example:
  - Stack contains 2 elements B and T. After 3 pushes, 2 pops, 4 pushes, 1 pop, 1 push, and 5 pops, what does the stack contain?

Stack Frames

- Storage requirements of each function call is placed in a *stack frame* which contains at least the following:
  - Return address (old LP)
  - Arguments (example: x, y, z in f(x, y, z))
  - Temp/Local variables
  - Original values of clobbered registers
- The stack is used for this storage, of course!
  - Grow (PUSH) stack for each function call.
  - Shrink (POP) stack on return.
  - Each call gets its own “stack frame”.

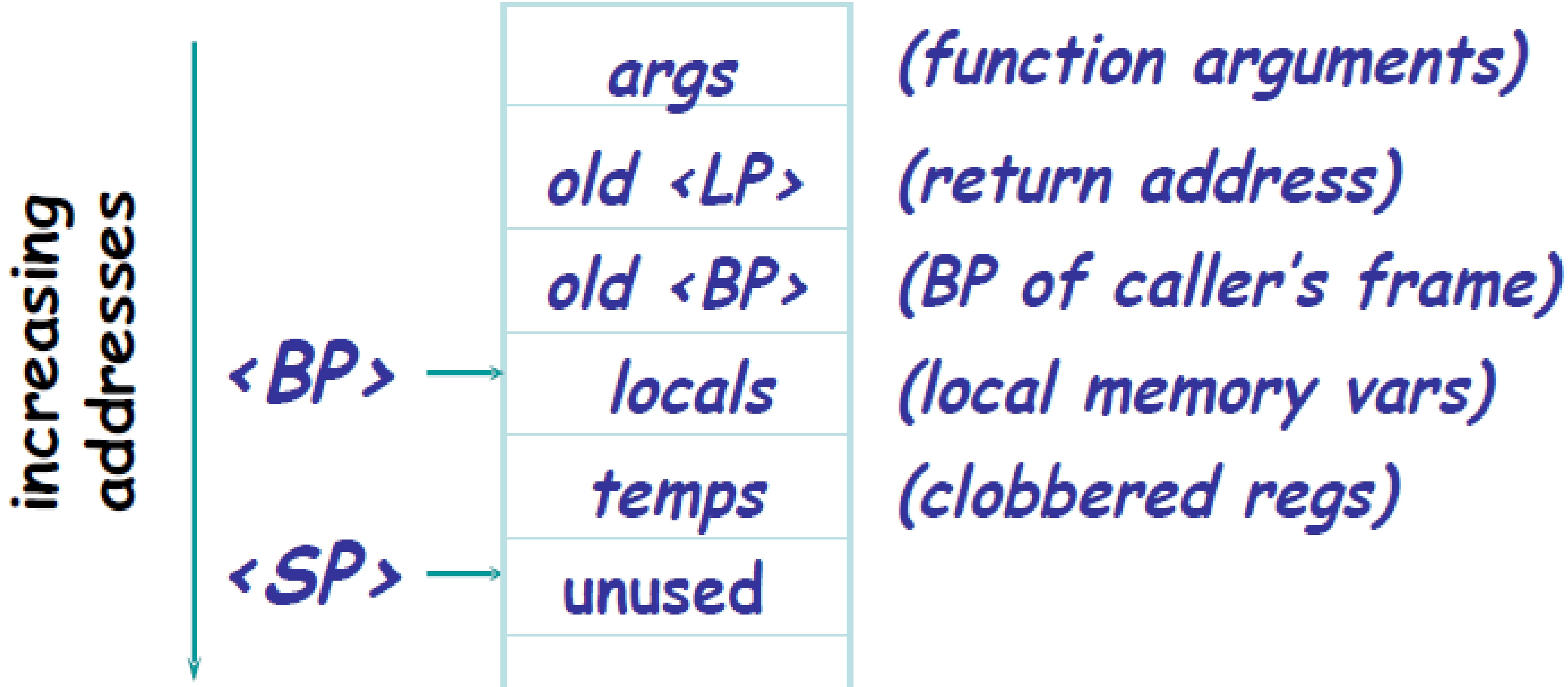
Stack State + Stack Frames

- *At the end of a function call, the state of the stack is the same as right before that function call.*
  - In other words, SP should be in the same position before and after the function call.
  - Example using the recursive version of fact() (go back a few slides):
    - From main, call fact(2) (push stack frame A)
    - From fact(2), call fact(1) (push stack frame B)
    - From fact(1), call fact(0) (push stack frame C)
    - Return fact(0) (pop stack frame C)
    - Return fact(1) (pop stack frame B)
    - Return fact(2) (pop stack frame A)

So Many Pointers!

- BP (R27) – Base Pointer, contains the address of the first local variable in a function's stack frame.
- LP (R28) – Linkage Pointer, contains the address of the line to return to when the function is finished.
- SP (R29) – Stack Pointer, contains the address of the line above the top of the stack (or points at the first unused location, if you prefer it that way).
- XP (R30) – Exception Pointer, not yet discussed. Therefore, not yet important.

Stack Frame Details



Stack Frame Details

- Per stack frame, for BSim:
  - Function arguments appear in reverse order: the LAST arg has the LOWEST address while the FIRST arg has the HIGHEST address.
  - The LP in the frame is the function's LP that will be used for the JMP(LP) at the end.
  - The BP in the frame is NOT the function's BP. It is the BP of whatever called this function.
  - To minimize number of clobbered registers, stack space can be used for local variables.
  - Temps are the original contents of the registers that will be clobbered in this function, and are to be restored before JMP(LP).

The Stack Discipline

- Caller's part :
  - Push args in stack in reverse order.
  - BR(callee, LP)
  - Should go back to next line (after branch instruction) after callee's JMP(LP)...
  - ... where it should now remove args from stack (can just subtract 4*NumberOfArgs from SP).

The Stack Discipline

- Callee’s part:
  - Save LP and BP on stack, then set BP to SP (so BP will point at first local, if any).
  - PUSH(LP) and PUSH(BP), then MOVE(SP, BP).
  - Allocate mem for local vars on stack so each call has its own memory.
  - Perform computation and leave final result in R0.
  - Save and Restore clobbered registers.
  - Restore stack to original state before entry.
  - JMP(LP)

Accessing the Stack Frame

- BP is base for accessing the current stack frame.
- Within one function call:
  - BP points after args, old LP, and old BP in stack frame.
  - <BP> points to local var #1, <BP>–12 points to arg#1.
- Notice that caller must push arguments in reverse order so arg #1 is always in the same place!
- So, given the function's BP and a register Rx:
  - How do we access argument #N?
  - How do we access local variable #N?
  - Note: For both, we have to be able to read from and write to the variable (hint: memory access).

Exercises

- Implement the recursive factorial function in BSim.
- Create a function that averages two integers. Then, use it to get the average of three integers.
  - average(x, average(y, z));
- Create a function that returns the square of an integer. Then, use it to replace each element in an array with its square.