Assembly vs. Low IR

- **Assembly code:**
  - Finite set of registers
  - Variables in memory
  - Variables accessed differently: global, local, heap, args, etc.
  - Uses a runtime stack
  - Calling sequences: special sequences of instructions for function calls and returns
  - Instruction set of target machine
  - Special instructions for accessing the runtime stack

- **Low IR code:**
  - Variables (and temporaries)
  - No runtime stack
  - No calling sequences
  - Some abstract set of instructions

Low IR to Assembly Translation

- **Variables:**
  - Register Allocation: map the variables to registers
  - Translate accesses to specific kinds of variables (globals, locals, arguments, etc.)

- **Calling sequences:**
  - Translate function calls and returns into appropriate sequences which pass parameters, save registers, and give back return values
  - Consists of push/pop operations on the runtime stack

- **Instruction set:**
  - Account for differences in the instruction set
  - **Instruction selection:** maps sets of low level IR instructions to instructions in the target machine

Run-Time Stack

- A frame (or activation record) for each function execution
  - Represents execution environment of the function
  - Includes: local variables, parameters, return value, etc.
  - Different frames for recursive function invocations

- Run-time stack of frames:
  - Push frame of f on stack when program calls f
  - Pop stack frame when f returns
  - Top frame - frame of currently executed function

- This mechanism is necessary to support recursion
  - Different activations of the same recursive function have different stack frames

Stack Pointers

- Usually run-time stack grows downwards
  - Address of top of stack decreases

- Values on current frame (i.e., frame on top of stack) accessed using two pointers:
  - Stack pointer (sp): points to frame top
  - Frame pointer (fp): points to frame base

- Variable access: use offset from fp (sp)

- When do we need two pointers?
  - If stack frame size not known at compile time
  - Example: alloc (dynamic allocation or stack)
Hardware Support

- Hardware provides:
  - Stack registers
  - Stack instructions
- Pentium:
  - Register for stack pointer: esp
  - Register for frame pointer: ebp
  - Push instructions: push, push, pushad etc
  - Pop instructions: pop, popa, popad etc
  - Call instruction: call
  - Return instruction: ret

Anatomy of a Stack Frame

Previous frame
  - (responsibility of the caller)
    - Param 1
    - Param n
    - Return address
  - Previous fp
  - Current frame
    - (responsibility of the callee)
      - Local 1
      - Local n
      - Param 1
      - Param n
      - Return address

Static Links

- Problem for languages with nested functions (Pascal, ML):
  - How do we access local variables from other frames?
- Need a static link: a pointer to the frame of enclosing function
- Previous fp = dynamic link, i.e. pointer to the previous frame in the current execution

Example Nested Procedures

procedure f(i : integer)
  var a : integer;
  procedure h(j : integer)
    begin a = j end
  procedure g(k : integer)
    begin h(k*k) end
  begin g(i+2) end

Saving Registers

- Problem: execution of invoked function may overwrite useful values in registers
- Generated code must:
  - Save registers when function is invoked
  - Restore registers when function returns
- Possibilities:
  - Caller saves and restores registers
  - Caller saves and restores registers
  - ... or both

Calling Sequences

- How to generate the code that builds the frames?
- Generate code which pushes values on stack:
  1. Before call instructions (caller responsibilities)
  2. At function entry (callee responsibilities)
- Generate code which pops values from stack:
  3. After call instructions (caller responsibilities)
  4. At return instructions (callee responsibilities)
- Calling sequences = sequences of instructions performed in each of the above 4 cases
Push Values on Stack
- Code before call instruction:
  - Push each actual parameter
  - Push caller-saved registers
  - Push static link (if necessary)
  - Push return address (current program counter) and jump to caller code
- Prologue = code at function entry
  - Push dynamic link (i.e., current fp)
  - Old stack pointer becomes new frame pointer
  - Push caller-saved registers
  - Push local variables

Pop Values from Stack
- Epilogue = code at return instruction
  - Pop (restore) callee-saved registers
  - Store return value at appropriate place
  - Restore old stack pointer (pop callee frame)
  - Pop old frame pointer
  - Pop return address and jump to that address
- Code after call
  - Pop (restore) caller-saved registers
  - Use return value

Example: Pentium
- Consider call foo(3, 5), callee-saved registers
- Code before call instruction:
  push $3  // push first parameter
  push $5  // push second parameter
  sub $8, $esp // make room for return value
  call_foo  // push return address and jump to callee
- Prologue:
  push %ebp  // push old fp
  mov %esp, %ebp // compute new fp
  push %ebp  // push callee saved registers
  sub $16, %esp // push 2 integer local variables

Example: Pentium
- Epilogue:
  pop %ebx  // restore callee-saved registers
  mov %eax, %ebx  // store return value
  mov %ebp, %esp // pop callee frame
  pop %ebp // restore old fp
  ret // pop return address and jump
- Code after call instruction:
  mov %eax(%ebp), %eax // use return value
  add $24, %esp // pop callee locals

Accessing Stack Variables
- To access stack variables: use offsets from fp
- Example:
  \[ fp + 8 \] = return value
  \[ fp + 24 \] = parameter 1
  \[ fp - 4 \] = local 1
- Translate low-level code to take into account the frame pointer:
  \[ a = fp + 1 \]
  \[ => (fp - 4) = (fp + 16) + 1 \]

Accessing Other Variables
- Global variables
  - Are statically allocated
  - Their addresses can be statically computed
  - Don’t need to translate low IR
- Heap variables
  - Are unnamed locations
  - Can be accessed only by dereferencing variables which hold their addresses
  - Therefore, they don’t explicitly occur in low-level code
Big Picture: Memory Layout

Run-time Support

- Code to maintain stack frames = run-time mechanism
- Array bounds checks: if v holds the address of an array element, insert array bounds checking code for v before each load (...=[v]) or store ([v] = ...)
  - Use type information from symbol table to see if v points to an array element
- Garbage collection: insert code which automatically deallocates heap objects when they are no longer referenced