Hayden Miedema and Conrad Toney

CIS 451

Professor Kurmas

Lab 5: Intel Machine Language

1. Explain what each of the assembly language instructions in exampleIML-1b.s does *and why*.

**pushq %rbp** - save frame pointer %rbp on the stack. This marks the beginning of the memory space.

**movq %rsp, %rbp -** store the value of stack pointer %rsp in %rbp. You move the value into another register so you can save the starting point and have a pointer to move throughout the memory space

**subq $16, %rsp -** subtract 16 from stack pointer. Save the value of four ints and an int takes up 4 bytes. 4 bytes \* 4 ints we need to move 16 bytes down.

**call main -**  calls to look for our main method in code. Then stores its return value into %eax

**movl $1324, -16(%rbp) -** move the constant 1324 into the memory address 4 bytes down from the frame pointer. This is where the first local variable is stored. We move down 4 bytes because an int takes up 4 bytes.

**movl $5657, -8(%rbp)** - move the constant 5657 into the memory address 8 bytes down from the frame pointer. This is where the second local variable is stored. It’s located 4 bytes away from the first local variable because the int is 4 bytes

**movl $9876, -12(%rbp)** - move the constant 9876 into the memory address 12 bytes down from the frame pointer. This is where the third local variable is stored. It’s located 4 bytes away from the second local variable because the int is 4 bytes

**movl $2221, -16(%rbp)** - move the constant 2221 into the memory address 16 bytes down from the frame pointer. This is where the fourth local variable is stored. It’s located 4 bytes away from the third local variable because the int is 4 bytes.

**movl -8(%rbp), %eax** - move the value at the memory address 8 bytes down from the frame pointer into register eax (general purpose register). We are going to do some subtraction and the value at 12(%rbp) is one of the values needed. So we move that value into a general purpose register.

**movl -4(%rbp), %edx** - move the value at the memory address 4 bytes down from the frame pointer into register edx. We are going to do some subtraction and the value at -4(%rbp) is one of the values needed. So we move that value into a general purpose register.

**subl %eax, %edx** - subtract the contents of eax into edx, the result is in edx. We do this so we can store the answer of a - b in the same register.

**movl %edx, %eax -** move the contents of edx into eax. We then store the the subtracted into another general purpose register.

**movl %eax, -12(%rbp)** - move the contents of eax into the stack at the memory address 12 bytes down from the frame pointer. We do this so we can update the value at -12(%rbp) to the new answer after the subtraction was made.

**movl -12(%rbp), %ecx** - move the value at the memory address 12 bytes down from the frame pointer into register ecx. This is setting up for the printf, because we use the value at -12(%rbp) in the printf.

**movl -8(%rbp), %edx** - move the value at the memory address 8 bytes down from the frame pointer into register edx. This is setting up for the printf, because we use the value at -8(%rbp) in the printf

**movl -4(%rbp), %eax** - move the value at the memory address 4 bytes down from the frame pointer into register eax. This is setting up for the printf, because we use the value at -4(%rbp) in the printf

**movl %eax, %esi** - move the contents of eax into esi. Moves the value into a register used for parameters

**movl $.LC0, %edi** - move the constant at the label .LC0 into register edi. We do this to load in the string for the printf

**movl $0, %eax** - move the constant 0 into register eax. You do this to zero out the register so there are no values in it

**call printf -** calls the C function printf. So we can print the message to the screen. It the stores its return value into %eax

**movl %eax, -16(%rbp)** - move the contents of register eax into the memory address 16 bytes down from the frame pointer. We do this so we can store the return value of the printf in the memory location -16(%rbp)

**movl -12(%rbp), %eax** - move the value at the memory address 12 bytes down from the frame pointer into the register eax. We need to return difference so we move the value of difference (-12(%rbp)) into the %eax so we can return it later.

**leave** - the leave instruction is short for mov %rsp %rbp then pop %rbp. This releases the whole frame, everything between %rsp and %rbp is gone, and the frame pointer is reset to the previous value.

**ret** - pops a location from hardware memory stack then jumps to that

2. Using Table 2-2, identify the addressing mode that corresponds to each of the four possible values of Mod. Hint: Look at the operands in the "Effective Address" column. If you saw EBP or %ebp in an assembly instruction, where would the data for that instruction come from? What addressing mode does that correspond do? How about if you saw either [EBP]-16 (which gcc writes as -16(%ebp))?

[EAX] - This is register indirect

[EAX]+disp8 - This is register indirect + displacement. Overall displacement

[EAX]+disp32 - This is register indirect + displacement. Overall displacement

EAX/AX/AL/MMO/XMMO - This is register direct, we can tell because the value is not surrounded with brackets, meaning we can directly access the value.

If we saw EBP or %ebp in an assembly instruction it would mean that the data for that instruction is coming directly from a register. This corresponds to register direct addressing. If we saw [EBP]-16 or 16(%ebp) this would be a displacement. It would get the memory address in the register and subtract 16 bytes from it to get the new memory address. Rather than direct addressing, this is a form of indirect addressing u

3. List the machine instruction for each of the instructions marked with a number, and identify the meaning of each byte. Clearly indicate how the source(s) and destination are specified. Some hints and sample output appear below.

* Remember that instructions have different lengths. Don't try to incorporate too much into an instruction.
* Remember that within each word, the least significant bits are on the right. This makes the instructions feel "backwards", but the constants appear "forward".
* Ignore the "l" on the end of mnemonics (i.e., "movl" means "mov").
* When necessary, see Table 3-1 for the meaning of the +rw in the opcode.

|  |  |
| --- | --- |
| Assembly Instruction | push %rbp (main+0) |
| Machine Instruction (hex) | 0x55 |
| Byte Number | 1 |
| Field Name | opcode |
| Field Value | 0x55 |
| Field Meaning | Operation + register |
| Info Source | Page 4-274 and Table 3.1 |

|  |  |  |  |
| --- | --- | --- | --- |
| Assembly Instruction | mov %rsp,%rbp (main+1) | | |
| Machine Instruction (hex) | 0x4889e5 | | |
| Byte Number | 1 | 2 | 3 |
| Field Name | REX.W prefix | opcode | Mod R/M |
| Field Value | 0x48 | 0x89 | 0xe5 |
| Field Meaning | 64-bit operand | move | destination [RBP] + offset |
| Info Source | Table 2.4 | Page-3-508 | Table 2.2 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Assembly Instruction | movl $0x1619,-0x8(%rbp) (main+15) | | | |
| Machine Instruction (hex) | 0xc745f819 160000 | | | |
| Byte Number | 1 | 2 | 3 | 4-7 |
| Field Name | Opcode | Mod R/M | offset | Immediate value |
| Field Value | 0xc7 | 0x45 | 0xf8 | 0x19160000 |
| Field Meaning | move | destination [RBP] + offset | Subtract from offset | Immediate Value |
| Info Source | Page 3-508 | Table 2.2 | Table 2.2 | Page 3-508 |

|  |  |  |  |
| --- | --- | --- | --- |
| Assembly Instruction | mov -0x8(%rbp),%eax (main+36) | | |
| Machine Instruction (hex) | 0x8b45f8 | | |
| Byte Number | 1 | 2 | 3 |
| Field Name | opcode | Mod R/M | offset |
| Field Value | 0x8b | 0x45 | 0xf8 |
| Field Meaning | move | destination [RBP] + offset | Offset -8 bits |
| Info Source | Page 3-508 | Table 2.2 | Page 3-508 |

|  |  |  |  |
| --- | --- | --- | --- |
| Assembly Instruction | mov %eax,-0xc(%rbp) (main+46) | | |
| Machine Instruction (hex) | 0x8945f4 | | |
| Byte Number | 1 | 2 | 3 |
| Field Name | opcode | Mod R/M | Offset |
| Field Value | 0x89 | 0x45 | 0xf4 |
| Field Meaning | move | destination [RBP] + offset | Offset - 4 bits |
| Info Source | Page 3-508 | Table 2.2 | Page 3-508 |

|  |  |  |
| --- | --- | --- |
| Assembly Instruction | mov $0x0,%eax (main+65) | |
| Machine Instruction (hex) | 0xb8000000 | |
| Byte Number | 1 | 2 |
| Field Name | opcode | immediate |
| Field Value | 0xb8 | 0x0 |
| Field Meaning | move | zero |
| Info Source | Page 3-508 and Table 3-1 | Table 2.2 |

|  |  |
| --- | --- |
| Assembly Instruction | Leaveq (main+81) |
| Machine Instruction (hex) | 0xc9 |
| Byte Number | 1 |
| Field Name | opcode |
| Field Value | 0xc9 |
| Field Meaning | Leave and release stack frame |
| Info Source | Page 3-461 |

4. Notice that the push instruction is only one byte long. How did the designers squeeze both the opcode and the operator into one byte?

* The top 4 bits are used for the operation (being push) then the register being used is signified with the bottom 4 bits.

5. When using Table 2-2, sometimes the y-axis refers to the source operand, and sometimes it refers to the destination. How can you tell whether the y-axis refers to the first or second operator? Hint: Compare instructions main+36 and main+46.

* The opcode is used to differentiate here. The 0x8b vs 0x89 opcode signify different forms to be used. 0x8b signifies Move r/m32 to r32, while 0x89 signifies Move r32 to r/m32. The difference here is seen in the order of either register, immediate value, or immediate value to register.

6. How/where does instruction main+15 encode that one of the parameters is an immediate value? How is the R/M byte for this instruction used?

* The offset is used in order to signify that a 32 bit immediate value is coming, rather than a register. Also, the opcode shows MOV r/m32, imm32

Extra Credit: Explain how the IA64 machine language encodes the "r" registers (r8d, r9d, ..., r15d). Your explanation should include a table for this instruction: sub %r11d, %r14d