Computer Science 220
Assembly Language & Comp. Architecture
Siena College
Fall 2010

Topic Notes: ISA Comparisons

With our discussion of the MIPS ISA now nearly complete, it is time to take a brief look at other ISAs and their characteristics.

We will consider the assembly language code generated from a few simple C programs for some of the ISAs we consider:

See Example:
`~jteresco/shared/cs220/examples/assembly`

RISC vs. CISC

We have used the terms RISC and CISC in passing, but now is a good time to say a bit more about them.

We will see how to implement the MIPS ISA directly in hardware, including complexities like data forwarding and hazard detection.

This is possible in an undergraduate course because of the simplicity of its design. MIPS includes a very limited number of instructions, all instructions are the same size, and there are very few addressing modes.

These are some of the features common to *Reduced Instruction Set Computer (RISC)* architectures, of which MIPS is an excellent example.

Many other architectures are too complex to be implemented directly in hardware. In these cases, an implementation might involve a simpler hardware (a *microarchitecture*) and an *interpreter* is used to execute instructions, guided by a *microprogram*.

Decoding and executing an instruction can require several steps, and the more complex the instruction (more operands, etc) the longer it will take.

These architectures are classified as *CISC* or *Complex Instruction Set Computer*.

Some characteristics of CISC architectures:

- CISC architectures have large instruction sets: many operations are supported as single machine instructions
- Individual CISC instructions may be very complex
  - the `CMPC3` and `CMPC5` *string comparison* instructions from VAX MACRO.

```
CMPC5   R5,STRING1,#^A/ /,R7,STRING2
```

This compares the character string whose starting address is specified by `STRING1` and whose length is in register `R5` to the string `STRING2`, length `R7`, using the space character to pad the shorter string for comparison purposes, using registers `R0`, `R1`, `R2` and `R3` to store information about the result of the comparison.

- Motorola 68000 `movem.l` instruction, that moves a specified subset of the 16 user-visible registers onto the call stack in a single instruction.

```
swap: link  %a6,#0
      movem.l #0x010f,-(%sp)
      ...
      movem.l (%sp)+,#0xf080
      unlk   %a6
      rts
```

  - Order for pushing: `a7..a0,d7..d0`, so the above results in `a0` and `d3..d0` being pushed onto the stack – 5 memory writes and a register modification!
  - Order for popping: `d0..d7,a0..a7` – same registers but this time 5 memory reads above.

  The code above would replace the longer code:

```
a = 8
i = 12
j = 14
swap: link  %a6,#0
      move.l %d0,-(%sp)
      move.l %d1,-(%sp)
      move.l %d2,-(%sp)
      move.l %d3,-(%sp)
      move.l %a0,-(%sp)
      ....code for swap from before...
      move.l (%sp)+,%a0
      move.l (%sp)+,%d3
      move.l (%sp)+,%d2
      move.l (%sp)+,%d1
      move.l (%sp)+,%d0
      unlk   %a6
      rts
```

- Another example from the VAX-11: the `SOBGTR` instruction – "Subtract One and Branch on GreaTeR than" – subtract 1 from the specified register and branch to the target address if the result is greater than 0. So an entire loop could be written:

```
        MOVAW   DATA,R6         ; load array ptr into R6
        MOVL    NUM,R9          ; initialize R9 with the number of elts
DOUBLE: ADDW2   (R6),(R6)+      ; double entry, increment ptr
        SOBGTR  R9,DOUBLE       ; loop control
```

- CISC instructions support a large variety of addressing modes. The 68000 has about a dozen operand modes, e.g.,
  - register direct
  - immediate
  - register indirect (pointers)
    `(%a3)`
  - register indirect with offset (structures)
    `4(%a6)`
  - register indirect with displacement and indexing (arrays)
    `4(%a4,%d1.w)`
  - address register indirect with predecrement
    `-(%a7)`
  - address register indirect with postincrement
    `(%a7)+`

  These can be used with nearly any instruction!
- CISC instructions often allow memory-based data for one or more operands in most instructions, so a single instruction may make several (slow) memory accesses.
- CISC instructions can vary in length, where the length often varies according to the addressing mode (extension words on 68000).
  - larger constant/offset values allowed – just in the next memory location.
- CISC implementations are often microcoded.
- CISC implementations are heavily pipelined.
- CISC systems usually offer a relatively small numbers of user-visible registers (a dozen or two).
- CISC results in a relatively compact code footprint (each instruction accomplishes a lot).
- If you have to program directly in assembly language, CISC isn't too bad. Compilers may make use of only a subset of available instructions (for optimizations or portability of code generation).

Researchers (many at Stanford) in the 1980's advocated for the *RISC* approach.

Characteristics of RISC architectures:

- Most RISC instructions can be executed in one "cycle".

  - SPARC supports a single step of multiply, not a full multiply in one instruction. (keep the common case fast!)

- RISC instruction sets may be large, but are easily decoded.
  - small number of opcodes: MIPS has 14 arithmetic, 4 load/store, 4 ($\times 14$) conditionals, 4 misc, 3 multiple cycle *macro* instructions.
- RISC instructions support a small number of addressing modes.
- RISC memory access is only through explicit operations: `LOAD` and `STORE`.
- RISC instructions are fixed-length (no extension words) with simple instruction formats: 3-operand operations, memory reference, conditional branch, jumps.
- RISC architectures are simple enough for hardware-based decoding, but there is likely some microcoding done.
- RISC instructions are pipelined, but pipes are shorter.
  - typical pipeline: instruction fetch, instruction decode, operand decode, execution, data write
- RISC architectures offer larger numbers of registers, frequently supplanting the use of stack (maybe thousands).
- Programs in a RISC assembly are longer, so have a relatively large code footprint.
- RISC is harder to program by hand – easier for a compiler.

Sun Sparc

Sun Microsystems' Sparc architectures enjoyed a long period of success from the early 1990's to the early 2000's.

Sparc is a RISC architecture, and contains many of the same features as MIPS.

Register Windows

A RISC system may have hundreds of registers in its register file.

One way to organize these registers is to treat the register file as circular and give each routine a "view" of a limited subset of these registers at any given time.

Sparc has many, many registers but only 32 visible at any given time.

One subset of registers is designated as "globals" and are visible to all routines.

Then, each routine has three subsets of registers

- “ins” – the actual parameters
- “locals” – local variables, temporaries
- “outs” – parameters to any subroutines called by this routine

The “outs” of a routine become the “ins” of a subroutine that it calls.

Think about how this works. Much of what we could do with the stack for parameter passing and local variable storage can be replaced with this.

We eliminate some of the problems of registers getting clobbered by subroutines and we speed function calls by reducing memory accesses needed to pass parameters on the stack.

But what if we have a deep call stack and we run out of registers? We can’t just clobber things, so we’d need to keep a stack also for those cases.

The actual implementation involves “pretending” to write things to the stack, but it only actually does this (“spill” to the stack, and “fill” from the stack to restore) if necessary given the current state of the register file.

Another problem: what about functions with more parameters than can be passed in the register window?

Observation: a compiler for a system using register windows would do well to eliminate recursion when possible.

Other Sparc Features

Some other things of note about Sparc:

- register names: ins, outs, globals, locals
- load/store as in MIPS (different syntax, same effect)
- compare and branch in separate instructions – the conditional branches look at the status codes set by a previous instruction
- 1 cycle branch delay slot – note the `nop`s in the assembly code

x86/IA-32 ISA

The text also describes in Section 2.17, some details of the Intel x86/IA-32 instruction set architecture. Now that you know something about the MIPS ISA and its simplicity, that section will make for interesting reading.

Some highlights:

A Brief History of the x86

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: 57 new "MMX" instructions are added, Pentium II
- 1999: The Pentium III added another 70 instructions (SSE)
- 2001: Another 144 instructions (SSE2)
- 2003: AMD extends the architecture to increase address space to 64 bits, widens all registers to 64 bits and other changes (AMD64)
- 2004: Intel capitulates and embraces AMD64 (calls it EM64T) and adds more media extensions
- 2006: SSE4: 54 new instructions from Intel
- 2007: SSE5: 170 new instructions from AMD
- 2008: Advanced Vector Extension from Intel to expand SSE register width to 256 (from 128), redefining 250 instructions, adding 128 new

---

x86/IA-32 Overview

- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes e.g., "base or scaled index with 8 or 32 bit displacement"
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow
  - much of the complexity comes from backwards compatibility and may safely be ignored by modern programmers/compilers

- 8 general purpose registers plus 8 special purpose
- Lots of restrictions and caveats

Quotes about the x86/IA-32

- “This history illustrates the impact of the ‘golden handcuffs’ of compatibility”
- “adding new features as someone might add clothing to a packed bag”
- “an architecture that is difficult to explain and impossible to love”
- “what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective”

---

x86 Basics: Registers, Data Types, and Memory

A diagram of the main registers can be found in Figure 2.36.

- modern x86/IA-32 has 8 32-bit integer registers
  - not entirely general-purpose: some instructions limit the choice of register operands to fewer than 8
- special-purpose 32-bit registers:
  - instruction pointer (program counter)
  - flags (condition codes)
  - floating-point and multimedia registers
- Registers names come from their historical special purposes:

`%eax`	accumulator (for arithmetic ops)
`%ebx`	base (address of array in memory)
`%ecx`	count (of loop iterations)
`%edx`	data (e.g., second operand for binary operations)
`%esi`	source index (for string copy or array access)
`%edi`	destination index (for string copy or array access)
`%ebp`	base pointer (base of current stack frame)
`%esp`	stack pointer (top of stack)
`%eip`	instruction pointer (program counter)
`%eflags`	flags (condition codes and other things)

- The letter “E” in the name indicates that the “extended” (32-bit) version.

- Registers can also be used to store 16- and 8-bit values
  - useful when writing smaller values to memory
  - the low 16 bits of a register are denoted by dropping the "E" from the register name, e.g., `%si`
  - the two 8-bit halves of the low 16 bits of the first four registers can be used as 8-bit registers by replacing "X" with "H" (high) or "L" (low)
- Many instructions are independent of data type, but some require that you select the proper instruction for the data types of the operands
- Memory: many x86 operations accept memory locations as operands
  - e.g., increment value in memory in one instruction (equivalent of `lw`, `addi`, `sw` in MIPS)
  - Figure 2.37 shows the valid combination (both from memory is not allowed).

---

x86 ISA

- Arithmetic operations: `ADD`, `SUB`, `NEG` (negate), `INC` (increment), `DEC` (decrement)
- Logical operations: `AND`, `OR`, `XOR`, `NOT`
- Shift operations: `SHL` (left), `SAR` (arithmetic right), `SHR` (logical right)
- Rotate operations (shift with wraparound): to the left (`ROL`) or to the right (`ROR`)
- typically specify one register as both the destination and one of the sources:

```
addl %eax,%ebx        # EBX <= EBX + EAX
```

  - we cannot use a single `ADD` instruction to put the sum of `EAX` and `EBX` into `ECX`
  - instruction name is extended with a label for the type of data, an "L" in the case above to indicate long, or 32-bit, operands (others are "W" for 16-bit (word) and "B" for 8-bit (byte) operands)
  - note destination appears last (though this is assembler-specific)
- immediate values of up to 32-bits are usually allowed (since we can have longer instructions), and values that fit into fewer bits are encoded as shorter instructions
- Immediate values are preceded with a dollar sign:

```
addl $20,%esp     # ESP <= ESP + 20
```

Beware: removing the dollar sign

```
addl 20,%esp      # ESP <= ESP + M[20]
```

which specifies the contents of memory location 20 to be added to ESP rather than the value 20

- A few more examples:

```
movl $10,%esi   # ESI <= 10
movl %eax,%ecx  # ECX <= EAX
xorl %edx,%edx  # EDX <= 0
```

- Data movement: the MOV instruction takes the place of both load and store
- Most x86 addressing modes are specific cases of the general mode:

```
displacement(SR1,SR2,scale)
```

which multiplies SR2 by scale, then adds both SR1 and displacement.

This complex addressing supports array accesses generated by high-level programs.

For example, to access the i^{th} element of an array of 32-bit integers, one could put a pointer to the base of the array into EBX and the index i into ESI, and execute

```
 movw (%ebx,%esi,4),%eax # EAX <= M[EBX + ESI * 4]
```

If the array started at the 28^{th} byte of a structure, and EBX instead held a pointer to the structure, one could still use this form by adding a displacement:

```
movw 28(%ebx,%esi,4),%eax   # EAX <= M[EBX + ESI * 4 + 28]
```

scale can be 1, 2, 4, or 8, defaults to 1.

- Examples of simpler cases of this addressing mode:

```
movb (%ebp),%al           # AL <= M[EBP]
movb -4(%esp),%al         # AL <= M[ESP - 4]
movb (%ebx,%edx),%al      # AL <= M[EBX + EDX]
movb 13(%ecx,%ebp),%al    # AL <= M[ECX + EBP + 13]
movb (,%ecx,4),%al        # AL <= M[ECX * 4]
movb -6(,%edx,2),%al      # AL <= M[EDX * 2 - 6]
movb (%esi,%eax,2),%al    # AL <= M[ESI + EAX * 2]
movb 24(%eax,%esi,8),%al  # AL <= M[EAX + ESI * 8 + 24]
```

Figure 2.38 also shows addressing modes with MIPS equivalents.

- direct addressing mode: the address to be used is specified as an immediate value in the instruction

```
movb 100,%al               # AL <= M[100]
movb label,%al             # AL <= M[label]
movb label+10,%al          # AL <= M[label+10]
movb 10(label),%al         # NOT LEGAL!
movb label(%eax),%al       # AL <= M[EAX + label]
movb 13+8*8-35+label(%edx),%al # AL <= M[EDX + label + 42]
movw $label,%eax           # EAX <= label
movw $label+10,%eax        # EAX <= label+10
movw $label(%eax),%eax     # NOT LEGAL!
```

- Condition codes: there are many, but we will look at 5 condition codes
  - sign flag (SF): was last result negative
  - zero flag (ZF): was last result 0
  - carry flag (CF): did last result generate a carry (among other uses)
  - overflow flag (OF): did last operation overflow
  - parity flag (PF): even parity of last operation

  Note: most, but not all instructions affect all flags

- Conditional branches: 8 basic branch conditions and their inverses are available plus the unconditional branch JMP
  - jo: overflow – OF is set
  - jp: parity – PF is set (even parity)
  - js: sign – SF is set (negative)
  - je: equal – ZF is set
  - jb: below – CF is set
  - jbe: below or equal – CF or ZF is set
  - jl: less – SF != OF
  - jle: less or equal – (SF != OF) or ZF is set

  All except can be inverted by inserting an "N" after the initial "J": JNB jumps if the carry flag is clear.

- Subroutine control: the CALL and RET instructions

  CALL pushes EIP, and its target can come from one of many addressing modes:

```
call printf         # (push EIP), EIP <= printf
call *%eax          # (push EIP), EIP <= EAX
call *(%eax)        # (push EIP), EIP <= M[EAX]
call *fptr          # (push EIP), EIP <= M[fptr]
call *10(%eax,%edx,2) # (push EIP), EIP <= M[EAX + EDX*2 + 10]
```

`RET` (return) pops the return address off the stack and into `EIP`

- Some typical instruction formats are shown in Figure 2.41.

---

x86 Wrapup

Just how complex is the x86 ISA today?

See Figure 2.43 for a summary.