ECE 290: Introduction to Computer Engineering **Fall 2000**
Notes from 29 November and 4 and 6 December Lectures

Instruction Set Architecture

Tradeoffs and design elements of instruction set architectures (ISA's) form the last component of our class. As you perhaps recall, an ISA defines the interface between software and hardware, abstracting the capabilities of a computer's datapath and standardizing the format of instructions to utilize those capabilities. Successful ISA's are rarely discarded, as success implies the existence of large amounts of software built to use the ISA. Rather, they are extended, and their original forms must be supported for decades (consider, for example, the IBM 360 and the Intel x86). Employing sound design principles is thus imperative in an ISA.

Those of you who go on to take ECE 291, ECE 311, and/or ECE 312 will further develop the ideas presented here in tandem with the computer architecture designs that we developed in the last few sets of notes.

Learning Objectives

This section outlines what you should expect to know, and what I'll expect you to know, after reading these notes. As I regularly indicate during lecture, the lists in class are slightly abbreviated due to the limited space available on a slide; the full list for this topic is given here.

Terminology and Circuits

- instruction format and fields
  - mode field
  - fixed-length instructions
  - variable-length instructions
  - implicit operands
  - general-purpose registers
  - special-purpose registers: stack pointer (SP), program counter (PC), processor status register (PSR) or processor status word (PSW), zero
  - 3-address, 2-address, 1-address, and 0-address formats
- addressing architectures: memory-to-memory, load-store (register-to-register), register-memory, accumulator, stack
- reduced instruction set computers (RISC)
- complex instruction set computers (CISC)
- instruction types
  - stack: push, pop
  - control flow operations: call, return, trap
  - control flow conditions: comparisons, bit tests
  - I/O: in, out
- procedure, system call: calling convention, callee saved, caller saved
- interrupt, exception
  - handler
  - vector table
  - interrupt chaining
- spill code
- I/O: ports, independent I/O, memory-mapped I/O

Tradeoffs

- fixed- versus variable-length instructions
- general-purpose versus special-purpose registers
- independent versus memory-mapped I/O

Comprehension/Integration

- Understand basic tradeoffs in ISA design.

Formats and Fields

The example architecture that we have developed in the last two months employs fixed-length instructions and a load-store architecture, two aspects that help to reduce the design space to a manageable set of choices. In a general ISA design, many other options exist for instruction formats.

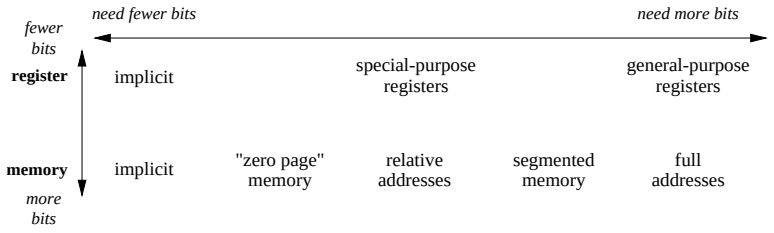
Recall the idea of separating the bits of an instruction into (possibly non-contiguous) fields. One of the fields must contain an opcode, which specifies the type of operation to be performed by the instruction. In our example architecture, the opcode specified both the type of operation and the types of arguments to the operation. In a more general architecture, many addressing modes are possible for each operand (as you learned in section), and we can think of the bits that specify the addressing mode as a separate field, known as the **mode** field. With regard to our example architecture, the first bit of the ALU/shift operations can be called a mode bit: when the bit has value 0, the instruction uses register-to-register mode; when the bit has value 1, the instruction uses immediate mode.

Several questions must be answered in order to define the possible instruction formats for an ISA. First, are instructions fixed-length or variable-length? Second, how many addresses are needed for each instruction, and how many of the addresses can be memory addresses? Finally, what forms of addresses are possible for each operand, *e.g.*, full memory addresses or only limited offsets relative to a register?

The answer to the first question depends on many factors, but several clear advantages exist for both answers. **Fixed-length instructions** are easy to fetch and decode. A processor knows in advance how many bits must be fetched to fetch a full instruction; fetching the opcode and mode fields in order to decide how many more bits are necessary to complete the instruction often takes more than one cycle, as did the two-word instructions in our example architecture. Fixing the time necessary for instruction fetch also simplifies pipelining. Finally, fixed-length instructions simplify the datapath by restricting instructions to the size of the bus and always fetching properly aligned instructions. As an example of this simplification, note that our 16-bit example architecture did not need to support addressing for individual bytes, only 16-bit words.

Variable-length instructions also have benefits, however. Variable-length encodings allow more efficient encodings, saving both memory and disk space. A register transfer operation, for example, clearly requires fewer bits than addition of values at two direct memory addresses for storage at a third. Fixed-length instructions must be fixed at the length of the longest possible instruction, whereas variable-length instructions can use lengths appropriate to each mode. The same tradeoff has another form in the sense that fixed-length ISA's typically eliminate many addressing modes in order to limit the size of the instructions. Variable-length instructions thus allow more flexibility; indeed, extensions to a variable-length ISA can incorporate new addressing modes that require longer instructions without affecting the original ISA.

Moving to the last of the three questions posed for instruction format definition, we explore a range of answers developed over the last few decades. Answers are usually chosen based on the number of bits necessary, and we use this metric to organize the possibilities. The figure below separates approaches into two dimensions: the vertical dimension divides addressing into registers and memory, and the horizontal dimension into varieties within each type.



As a register file contains fewer registers than a memory does words, the use of register operands rather than memory addresses reduces the number of bits required to specify an operand. Our example architecture used only register operands to stay within the limit imposed by the decision to use only 16-bit instructions. Both register and memory addresses, however, admit a wide range of implementations.

Implicit operands of either type require no additional bits for the implicit address. A typical procedure call instruction, for example, pushes a return address onto the stack, but the stack pointer can be named implicitly, without the use of bits in the instruction beyond the opcode bits necessary to specify a procedure call. Similarly, memory addresses can be

implicitly equated to other memory addresses; an increment instruction operating on a memory address, for example, implicitly writes the result back to the same address. The opposite extreme provides full addressing capabilities, either to any register in the register file or to any address in the memory. As addressing decisions are usually made for classes of instructions rather than individual operations, I have used the term **general-purpose registers** to indicate that the registers are used in any operation.

Special-purpose registers, in contrast, split the register file and allow only certain registers to be used in each operation. For example, the Motorola 680x0 series, used until recently in Apple Macintosh computers, provides distinct sets of address and data registers. Loads and stores use the address registers; ALSU operations use the data registers. As a result, each instruction selects from a smaller set of registers and thus requires fewer bits in the instruction to name the register for use.

As full memory addresses require many more bits than full register addresses, a wider range of techniques has been employed to reduce the length. "Zero page" addresses, as defined in the 6510 (6502) ISA used by Commodore PET's,* C64's,† and VIC 20's, prefixed a one-byte address with a zero byte, allowing shorter instructions when memory addresses fell within the first 256 memory locations. Assembly and machine language programmers made heavy use of these locations to produce shorter programs.

Relative addressing appeared in the context of control flow instructions of our example architecture, but appears in many modern architectures as well. The Alpha, for example, has a relative form of procedure call with a 21-bit offset (plus or minus a megabyte). The x86 architecture has a "short" form of branch instructions that uses an 8-bit offset.

Segmented memory is a form of relative addressing that uses a register (usually implicit) to provide the high bits of an address and an explicit memory address (or another register) to provide the low bits. In the x86 architecture, for example, 20-bit addresses are found by adding a 16-bit segment register extended with four zero bits to a 16-bit offset.

Addressing Architectures

One question remains for the definition of instruction formats: how many addresses are needed for each instruction, and how many of the addresses can be memory addresses? The first part of this question usually ranges from zero to three, and is very rarely allowed to go beyond three. The answer to the second part determines the **addressing architecture** implemented by an ISA. We now illustrate the tradeoffs between five distinct addressing architectures through the use of a running example, the assignment $X = AB + C/D$.

A binary operator requires two source operands and one destination operand, for a total of three addresses. The ADD instruction, for example, has a **3-address** format:

```
      ADD  A,B,C      ; M[A] ← M[B] + M[C]
or    ADD  R1,R2,R3   ; R1 ← R2 + R3
```

If all three addresses can be memory addresses, the ISA is dubbed a **memory-to-memory architecture**. Such architectures may have small register sets or even lack a register file completely. To implement the assignment, we assume the availability of two memory locations, T1 and T2, for temporary storage:

```
MUL  T1,A,B      ; T1 ← M[A] * M[B]
DIV  T2,C,D      ; T2 ← M[C]/M[D]
ADD  X,T1,T2     ; X ← M[T1] + M[T2]
```

The assignment requires only three instructions to implement, but each instruction contains three full memory addresses, and is thus very long.

At the other extreme is the **load-store architecture** used by the ISA that we developed earlier. In a load-store architecture, only loads and stores can use memory addresses; all other operations use only registers. As most instructions use only registers, this type of addressing architecture is also called a **register-to-register architecture**. The example assignment translates to the code at the top of the next page, which assumes that R1, R2, and R3 are free for use.

*My computer in junior high school.

†My computer in high school.

```
LD    R1,A        ; R1 ← M[A]
LD    R2,B        ; R2 ← M[B]
MUL   R1,R1,R2    ; R1 ← R1 * R2
LD    R2,C        ; R2 ← M[C]
LD    R3,D        ; R3 ← M[D]
DIV   R2,R2,R3    ; R2 ← R2/R3
ADD   R1,R1,R2    ; R1 ← R1 + R2
ST    R1,X        ; M[X] ← R1
```

Eight instructions are necessary, but no instruction requires more than one full memory address, and several use only register addresses, allowing the use of shorter instructions. The need to move data in and out of memory explicitly, however, also requires a reasonably large register set, as is available in the Sparc, Alpha, and IA-64 architectures.

Architectures that use other combinations of memory and register addresses with 3-address formats are not named. Unary operators and transfer operators require only one source operand, thus can use a 2-address format (*e.g.*, NOT A,B). Binary operations can also use **2-address** format if one operand is implicit, as in the following instructions:

```
     ADD  A,B      ; M[A] ← M[A] + M[B]
or   ADD  R1,B     ; R1 ← R1 + M[B]
```

The second instruction, in which one address is a register and the second is a memory address, defines a **register-memory architecture**. As shown below, such architectures strike a balance between the two architectures just discussed.

```
LD    R1,A        ; R1 ← M[A]
MUL   R1,B        ; R1 ← R1 * M[B]
LD    R2,C        ; R2 ← M[C]
DIV   R2,D        ; R2 ← R2/M[D]
ADD   R1,R2       ; R1 ← R1 + R2
ST    R1,X        ; M[X] ← R1
```

The assignment requires six instructions using at most one memory address each; like memory-to-memory architectures, register-memory architectures use relatively few registers. Note that two-register operations are also allowed. Intel's x86 ISA is a register-memory architecture.

Several ISA's of the past[‡] used a special-purpose register called the accumulator for ALU operations, and are called **accumulator architectures**. The accumulator in such architectures is implicitly both a source and the destination for any such operation, allowing a **1-address** format for instructions, as shown below.

```
     ADD  B        ; ACC ← ACC + M[B]
or   ST   E        ; M[E] ← ACC
```

Accumulator architectures strike the same balance as register-memory architectures, but use fewer registers. Note that memory location X is used as a temporary storage location as well as the final storage location in the following code:

```
LD    A           ; ACC ← M[A]
MUL   B           ; ACC ← ACC * M[B]
ST    X           ; M[X] ← ACC
LD    C           ; ACC ← M[C]
DIV   D           ; ACC ← ACC/M[D]
ADD   X           ; ACC ← ACC + M[X]
ST    X           ; M[X] ← ACC
```

The last addressing architecture that we discuss is rarely used for modern general-purpose processors, but is perhaps the most familiar to you because of its use in scientific and engineering calculators for the last fifteen to twenty years. A **stack architecture** maintains a stack of values and draws all ALU operands from this stack, allowing these

[‡]The 6510/6502 as well, if memory serves, as the 8080, Z80, and Z8000, which used to drive parlor video games.

instructions to use a **0-address** format. A special-purpose stack pointer (SP) register points to the top of the stack in memory. and operations analogous to load (**push**) and store (**pop**) are provided to move values on and off the stack. To implement our example assignment, we first transform it into postfix notation (also called reverse Polish notation):

A B * C D / +

The resulting sequence of symbols transforms on a one-to-one basis into instructions for a stack architecture:

PUSH	A	; $SP \leftarrow SP - 1, M[SP] \leftarrow M[A]$	A		
PUSH	B	; $SP \leftarrow SP - 1, M[SP] \leftarrow M[B]$	B	A	
MUL		; $M[SP + 1] \leftarrow M[SP + 1] * M[SP], SP \leftarrow SP + 1$	AB		
PUSH	C	; $SP \leftarrow SP - 1, M[SP] \leftarrow M[C]$	C	AB	
PUSH	D	; $SP \leftarrow SP - 1, M[SP] \leftarrow M[D]$	D	C	AB
DIV		; $M[SP + 1] \leftarrow M[SP + 1]/M[SP], SP \leftarrow SP + 1$	C/D	AB	
ADD		; $M[SP + 1] \leftarrow M[SP + 1] + M[SP], SP \leftarrow SP + 1$	AB+C/D		
POP	X	; $M[X] \leftarrow M[SP], SP \leftarrow SP + 1$			

The values to the right are the values on the stack, starting with the top value on the left and progressing downwards, *after the completion of each instruction*.

Common Special-Purpose Registers

This section illustrates the uses of special-purpose registers through a few examples.

The **stack pointer (SP)** points to the top of the stack in memory. Most older architectures support push and pop operations that implicitly use the stack pointer. Modern architectures assign a general-purpose register to be the stack pointer and reference it explicitly, although an assembler may support instructions that appear to use implicit operands but in fact translate to machine instructions with explicit reference to the register defined to be the SP.

The **program counter (PC)** points to the next instruction to be executed. Some modern architectures expose it as a general-purpose register, although its distinct role in the implementation keeps such a model from becoming as common as the use of a general-purpose register for the SP.

The **processor status register (PSR)**, also known as the **processor status word (PSW)**, contains all status bits as well as a mode bit indicating whether the processor is operating in user mode or privileged (operating system) mode. Having a register with this information allows more general access than is possible solely through the use of control flow instructions.

The **zero register** appears in modern architectures of the RISC variety (defined in the next section of these notes). The register is read-only and serves both as a useful constant and as a destination for operations performed only for their side-effects (*e.g.*, setting status bits). The availability of a zero register also allows certain opcodes to serve double duty. A register-to-register add instruction becomes a register move instruction when one source operand is zero. Similarly, an immediate add instruction becomes an immediate load instruction when one source operand is zero.

Reduced Instruction Set Computers

By the mid-1980's, the VAX architecture dominated the workstation and minicomputer markets, which included most universities. Digital Equipment Corporation, the creator of the VAX, was second only to IBM in terms of computer sales. VAXen, as the machines were called, used microprogrammed control units and supported numerous addressing modes as well as very complex instructions ranging from "square root" to "find roots of polynomial equation."

The impact of increasingly dense integrated circuit technology had begun to have its effect, however, and in view of increasing processor clock speeds, more and more programmers were using high-level languages rather than writing assembly code. Although assembly programmers often made use of the complex VAX instructions, compilers were usually unable to recognize the corresponding high-level language constructs and thus were unable to make use of the instructions.

Increasing density also led to rapid growth in memory sizes, to the point that researchers began to question the need for variable-length instructions. Recall that variable-length instructions allow shorter codes by providing more efficient

instruction encodings. With the trend toward larger memories, code length was less important. The performance advantage of fixed-length instructions, which simplifies the datapath and enables pipelining, on the other hand, was very attractive.

Researchers leveraged these ideas, which had been floating around the research community (and had appeared in some commercial architectures) to create **reduced instruction set computers**, or **RISC** machines. The competing VAXen were labeled **CISC** machines, which stands for **complex instruction set computers**.

RISC machines employ fixed-length instructions and a load-store architecture, allowing only a few addressing modes and small offsets. This combination of design decisions enables deep pipelines and multiple instruction issues in a single cycle (termed superscalar implementations), and for years, RISC machines were viewed by many researchers as the proper design for future ISA's. However, companies such as Intel soon learned to pipeline microoperations after decoding instructions, and CISC architectures now offer competitive if not superior performance in comparison with RISC machines. The VAXen are dead, of course,[§] having been replaced by the Alpha.

Procedure and System Calls

A **procedure** is a sequence of instructions that executes a particular task. Procedures are used as building blocks for multiple, larger tasks. The concept of a procedure is fundamental to programming, and appears in some form in every high-level language as well as in most low-level designs.[¶] For our purposes, the terms procedure, subroutine, function, and method are synonymous, although they usually have slightly different meanings from the linguistic point of view. Procedure calls are supported through **call** and **return** control flow instructions. The first instruction in the code below, for example, transfers control to the procedure "DoSomeWork," which presumably does some work, then returns control to the instruction following the call.

```
loop:         CALL   DoSomeWork
              CMP    R6,#1          ; compare return value in R6 to 1
              BEQ    loop           ; keep doing work until R6 is not 1

DoSomeWork:   ...                   ; set R6 to 0 when all work is done, 1 otherwise
              RETN
```

The procedure also places a return value in R6, which the instruction following the call compares with immediate value 1. Until the two are not equal (*i.e.*, all work is done), the branch returns control to the call and executes the procedure again.

As you may recall, the call and return use the stack pointer to keep track of nested calls. Sample RTL for these operations appears below.

call RTL
$$SP \leftarrow SP - 1$$
$$M[SP] \leftarrow PC$$
$$PC \leftarrow \text{procedure start}$$

return RTL
$$PC \leftarrow M[SP]$$
$$SP \leftarrow SP + 1$$

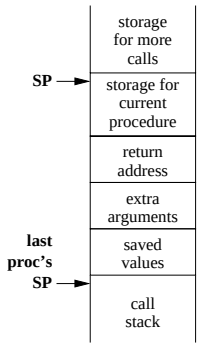
While an ISA provides the call and return instructions necessary to support procedures, it does not specify how information is passed to or returned from a procedure. A standard for such decisions is usually developed and included in descriptions of the architecture, however. This **calling convention** specifies how information is passed between a caller and a callee. In particular, it specifies the following: where arguments must be placed, *i.e.*, in registers or in specific stack memory locations; which registers can be used or changed by the procedure; and where any return value must be placed.

The term "calling convention" is also used in the programming language community to describe the convention for deciding what information is passed for a given call operation. For example, are variables passed by value, by pointers to values, or in some other way? However, once the things to be sent are decided, the architectural calling convention that we discuss in this class is used to determine where to put the data in order for the callee to be able to find it.

[§] Unless you talk with customer support employees, for whom no machine ever dies.

[¶] The architecture that you used in the labs allowed limited use of procedures in its microprogram.

Calling conventions for architectures with large register sets typically pass arguments in registers, and nearly all conventions place the return value in a register. A calling convention also divides the register set into **caller saved** and **callee saved** registers. Caller saved registers can be modified arbitrarily by the called procedure, whereas any value in a callee saved register must be preserved. Similarly, before calling a procedure, a caller must preserve the values of any caller saved registers that are needed after the call. Registers of both types usually saved on the stack by the appropriate code (caller or callee).



A typical stack structure appears in the figure to the right. In preparation for a call, a caller first stores any caller saved registers on the stack. Arguments to the procedure to be called are pushed next. The procedure is called next, implicitly pushing the return address (the address of the instruction following the call instruction). Finally, the called procedure may allocate space on the stack for storage of callee saved registers as well as local variables.

As an example, the following calling convention can be applied to our example architecture: the first three arguments must be placed in R0 through R2 (in order), with any remaining arguments on the stack; the return value must be placed in R6; R0 through R2 are caller saved, as is R6, while R3 through R5 are callee saved; R7 is used as the stack pointer. The code fragments below use this calling convention to implement a procedure and a call of that procedure.

```
int add3 (int n1, int n2, int n3) {
    return (n1 + n2 + n3);
}
. . .
printf ("%d", add3 (10, 20, 30));
```

by convention:
- n1 is in R0
- n2 is in R1
- n3 is in R2
- return value is in R6

```
add3:   ADD     R0,R0,R1
        ADD     R6,R0,R2
        RETN
        . . .
        PUSH    R4          ; save the value in R4
        LDI     R0,#10      ; marshal arguments
        LDI     R1,#20
        LDI     R2,#30
        CALL    add3
        MOV     R1,R6       ; return value becomes second argument
        LDI     R0,"%d"     ; load a pointer to the string
        CALL    printf
        POP     R4          ; restore R4
```

The add3 procedure takes three integers as arguments, adds them together, and returns the sum. The procedure is called with the constants 10, 20, and 30, and the result is printed. By the calling convention, when the call is made, R0 must contain the value 10, R1 the value 20, and R2 the value 30. We assume that the caller wants to preserve the value of R4, but does not care about R3 or R5. In the assembly language version on the right, R4 is first saved to the stack, then the arguments are marshaled into position, and finally the call is made. The procedure itself needs no local storage and does not change any callee saved registers, thus must simply add the numbers together and place the result in R6. After add3 returns, its return value is moved from R6 to R1 in preparation for the call to printf. After loading a pointer to the format string into R0, the second call is made, and R4 is restored, completing the translation.

System calls are almost identical to procedure calls. As with procedure calls, a calling convention is used: before invoking a system call, arguments are marshaled into the appropriate registers or locations in the stack; after a system call returns, any result appears in a pre-specified register. The calling convention used for system calls need not be the same as that used for procedure calls. Rather than a call instruction, system calls are usually initiated with a **trap** instruction, and system calls are also known as traps. With many architectures, a system call places the processor in privileged or kernel mode, and the instructions that implement the call are considered to be part of the operating system. The term system call arises from this fact.

Interrupts and Exceptions

Unexpected processor interruptions arise both from interactions between a processor and external devices and from errors or unexpected behavior in the program being executed. The term **interrupt** is reserved for asynchronous interruptions generated by other devices, including disk drives, printers, network cards, video cards, keyboards, mice, and any number of other possibilities. **Exceptions** occur when a processor encounters an unexpected opcode or operand.

An undefined instruction, for example, gives rise to an exception, as does an attempt to divide by zero. Exceptions usually cause the current program to terminate, although many operating systems will allow the program to catch the exception and to handle it more intelligently. The table below summarizes the characteristics of the two types and compares them to system calls.

type	generated by	example	asynchronous	unexpected
interrupt	external device	packet arrived at network card	yes	yes
exception	invalid opcode or operand	divide by zero	no	yes
trap/system call	deliberate, via trap instruction	print character to console	no	no

Interrupts occur asynchronously with respect to the program. Most designs only recognize interrupts between instructions, *i.e.*, the presence of interrupts is checked only after completing an instruction rather than in every cycle. In pipelined designs, however, instructions execute simultaneously, and the decision as to which instructions occur "before" an interrupt and which occur "after" must be made by the processor. Exceptions are not asynchronous in the sense that they occur for a particular instruction, thus no decision need be made as to instruction ordering. After determining which instructions were before an interrupt, a pipelined processor discards the state of any partially executed instructions that occur "after" the interrupt and completes all instructions that occur "before." The terminated instructions are simply restarted after the interrupt completes. Handling the decision, the termination, and the completion, however, significantly increases the design complexity of the system.

The code associated with an interrupt, an exception, or a system call is a form of procedure called a **handler**, and is found by looking up the interrupt number, exception number, or trap number in a table of functions called a **vector table**. Separate vector tables exist for each type (interrupts, exceptions, and system calls). Interrupts and exceptions share a need to save all registers and status bits before execution of the corresponding handler code (and to restore those values afterward). Generally, the values—including the status word register—are placed on the stack. With system calls, saving and restoring any necessary state is part of the calling convention. A special return from interrupt instruction is used to return control from the interrupt handler to the interrupted code; a similar instruction forces the processor back into user mode when returning from a system call.

Interrupts are also interesting in the sense that typical computers often have many interrupt-generating devices but only a few interrupts. Interrupts are prioritized by number, and only an interrupt with higher priority can interrupt another interrupt. Interrupts with equal or lower priority are blocked while an interrupt executes. Some interrupts can also be blocked in some architectures by setting bits in a special-purpose register called an interrupt mask. While an interrupt number is masked, interrupts of that type are blocked, and can not occur.

As several devices may generate interrupts with the same interrupt number, interrupt handlers can be **chained** together. Each handler corresponds to a particular device. When an interrupt occurs, control is passed to the handler for the first device, which accesses device registers to determine whether or not that device generated an interrupt. If it did, the appropriate service is provided. If not, or after the service is complete, control is passed to the next handler in the chain, which handles interrupts from the second device, and so forth until the last handler in the chain completes. At this point, registers and processor state are restored and control is returned to the point at which the interrupt occurred.

Control Flow Conditions

Control flow instructions may change the PC, loading it with an address specified by the instruction. Although any addressing mode can be supported, the most common specify an address directly in the instruction, use a register as an address, or use an address relative to a register.

Unconditional control flow instructions typically provided by an ISA include procedure calls and returns, traps, and jumps. Conditional control flow instructions are branches, and are logically based on status bits set by two types of instructions: **comparisons** and **bit tests**. Comparisons subtract one value from another to set the status bits, whereas bit tests use an AND operation to check whether certain bits are set or not in a value.

Many older architectures, such as that used for your labs, implement status bits as special-purpose registers and implicitly set them for certain instructions. A branch based on R2 being less or equal to R3 can then be written as shown on the next page. The status bits are set by subtracting R3 from R2 with the function unit.

```
CMP     R2,R3     ; R2 < R3 : CNZ ← 110, R2 = R3 : CNZ ← 001, R2 > R3 : CNZ ← 000
BLE     R1        ; Z XOR C = 1 : PC ← R1
```

The status bits are not always implemented as special-purpose registers; instead, they may be kept in general-purpose registers or not kept at all. For example, the Alpha ISA stores the results of comparisons in general-purpose registers, and the same branch is instead implemented as follows:

```
CMPLE   R4,R2,R3  ; R2 ≤ R3 : R4 ← 1, R2 > R3 : R4 ← 0
BNE     R4,R1     ; R4 ≠ 0 : PC ← R1
```

Finally, status bits can be calculated, used, and discarded within a single instruction, in which case the branch is written as follows:

```
BLE     R1,R2,R3  ; R2 ≤ R3 : PC ← R1
```

The three approaches have advantages and disadvantages similar to those discussed in the section on addressing architectures: the first has the shortest instructions, the second is the most general and simplest to implement, and the third requires the fewest instructions.

Stack Operations

Two types of stack operations are commonly supported. Push and pop are the basic operations in many older architectures, and values can be placed upon or removed from the stack using these instructions. In more modern architectures, in which the SP becomes a general-purpose register, push and pop are replaced with indexed loads and stores, *i.e.*, loads and stores using the stack pointer and an offset as the address for the memory operation. Stack updates are performed using the ALU, subtracting and adding immediate values from the SP as necessary to allocate and deallocate local storage.

Stack operations serve three purposes in a typical architecture. The first is to support procedure calls, as illustrated in a previous section. The second is to provide temporary storage during interrupts, as mentioned earlier.

The third use of stack operations is to support **spill code** generated by compilers. Compilers first translate high-level languages into an intermediate representation much like assembly code but with an extremely large (theoretically infinite) register set. The final translation step translates this intermediate representation into assembly code for the target architecture, assigning architectural registers as necessary. However, as real ISA's support only a finite number of registers, the compiler must occasionally spill values into memory. For example, if ten values are in use at some point in the code, but the architecture has only eight registers, spill code must be generated to store the remaining two values on the stack and to restore them when they are needed.

I/O

As a final topic for the course, we now consider how a processor connects to other devices to allow input and output. We have already discussed interrupts, which are a special form of I/O in which only the signal requesting attention is conveyed to the processor. Communication of data occurs through instructions similar to loads and stores. A processor is designed with a number of **I/O ports**—usually read-only or write-only registers to which devices can be attached with opposite semantics. That is, a port is usually written by the processor and read by a device or written by a device and read by the processor.

The question of exactly how I/O ports are accessed is an interesting one. One option is to create special instructions, such as the **in** and **out** instructions of the x86 architecture. Port addresses can then be specified in the same way that memory addresses are specified, but use a distinct address space. Just as two sets of special-purpose registers can be separated by the ISA, such an **independent I/O** system separates I/O ports from memory addresses by using distinct instructions for each class of operation.

Alternatively, device registers can be accessed using the same load and store instructions as are used to access memory. This approach, known as **memory-mapped I/O**, requires no new instructions for I/O, but demands that a region of the memory address space be set aside for I/O. The memory words with those addresses, if they exist, can not be accessed during normal processor operations.