13. Stored-Program Computers

13.1 Introduction

This chapter concentrates on the low-level usage and structure of stored program computers. We focus on a particular hypothetical machine known as the ISC, describing its programming in assembly language. We show how recursion and switch statements are compiled into machine language, and how memory-mapped overlapped I/O is achieved. We also show the logic implement of the ISC, in terms of registers, buses, and finite-state machine controllers.

13.2 Programmer's Abstraction for a Stored-Program Computer

By stored-program computer, we mean a machine in which the program, as well as the data, are stored in memory, each word of which can be accessed in uniform time. Most of the high-level language programming the reader has done will likely have used this kind of computer implicitly. However, the program that is stored is not high-level language *text*. If it were, then it would be necessary to constantly parse this text, which would slow down execution immensely. Instead one of two other forms of storage is used: An abstract syntax representation of the program could be stored. The identifiers in this representation are pre-translated, and the structure is traversed dynamically as needed during execution. This is the approach used by an ***interpreter*** for the language. A second approach is to use a ***compiler*** for the language. The compiler translates the program into the very low-level language native to the machine, appropriately called ***machine language***. The native machine language acts as a least-common-denominator language for the computer. A machine that had to understand, at a native level, many different languages would be prohibitively complex and slow. Machine language is rarely programmed directly, since that would involve manipulating bits, with which it is easy to err. Instead, an equivalent symbolic form known as ***assembly language*** is employed.

In this chapter, we will build up a stored-program computer using our knowledge of finite-state machine components described earlier. But first, we describe the native language of a simple computer using assembly language. Then we "build-down" from higher-level language constructs to the assembly language to see how various algorithmic concepts get translated.

In the mid-1980's, a major paradigm shift began, from CISCs (Complex Instruction Set Computers) to RISCs (Reduced Instruction Set Computers). RISCs tried to take a "lean and mean" approach, in contrast to their predecessor CISCs, which were becoming bloated with complexity. RISCs focused on features related to speed and simplicity and consciously avoided including the "kitchen sink" in the instruction repertoire. The machine we use here for illustration is called the ISC, for Incredibly Simple Computer. It is of the RISC philosophy, but simpler than most RISCs for tutorial purposes.

The following is a terse description of the ISC. The unit of addressability of the ISC is one 32-bit word. The ISC has a 32-bit address space. This means that up to 2^{32} different words can be addressed in the memory, in principle, although a given implementation will usually contain far fewer words. Memory words are addressed by a signed integer, and negative addresses are typically used for "memory-mapped I/O", as described later. Instructions in the ISC are all one word long. Both instructions and data are stored in the memory of the computer. The instructions get into the memory by the execution of a special program known as the **loader**, which takes the output of the compiler and loads it into memory. A special part of memory known as the read-only memory (ROM) contains a primitive loader that brings in other programs from a cold-start.

Although the instructions operate on data stored in the memory, ISC instructions do not reference memory locations directly. Instead, the data in memory are brought into **registers** and the instructions specify operation on the registers. The registers also serve to hold addresses designating the locations in memory to and from which data fetching and storage occurs.

Internal to the ISC processor, but accessible by the programmer, are 32 registers, numbered 0-31. All processor state is contained in the registers and the instruction pointer (IP) (equivalent to what is sometimes called "program counter" (PC), unfortunately not a thing that counts programs). The IP contains the address of the next instruction to be executed.

The following kinds of addressing are used within ISC:

> **Register-indirect addressing** is used in all operations involving addressing, including the **jump** operations, **load**, and **store**. In other words, the memory address is contained in a register (put there earlier by the program itself) and the instruction refers to the register that contains the address.
>
> **Immediate values** are used in the **lim** and **aim** operations. The term "immediate" means that the datum comes immediately from the instruction itself, rather than from a register or memory.

In the following table, Ra, Rb, and Rc stand for register indices. The Java language equivalent is given, followed by a brief English description of the action of each instruction. In the cases of the arithmetic instructions (add, sub, mul, div), if the result does not fit into 32 bits, only the lower-order 32 bits are stored.

lim Ra C **reg[Ra] = C**
Load immediate to register *Ra* the signed 24-bit integer (or address) constant C.

aim Ra C **reg[Ra] += C**
Add immediate to register *Ra* the signed 24-bit integer (or address) constant C.

load Ra Rb **reg[Ra] = mem[reg[Rb]]**
Load into *Ra* the contents of the memory location addressed by *Rb*.

store Ra Rb **mem[reg[Ra]] = reg[Rb]**
Store into the memory location addressed by *Ra* the contents of *Rb*.

copy Ra Rb **reg[Ra] = reg[Rb]**
Copy into *Ra* the contents of register *Rb*.

add Ra Rb Rc **reg[Ra] = reg[Rb] + reg[Rc]**
Put into *Ra* the **sum** of the contents of *Rb* and the contents of *Rc*.

sub Ra Rb Rc **reg[Ra] = reg[Rb] - reg[Rc]**
Put into *Ra* the contents of *Rb* **minus** the contents of *Rc*.

mul Ra Rb Rc **reg[Ra] = reg[Rb] * reg[Rc]**
Put into *Ra* the **product** the contents of *Rb* and the contents of *Rc*.

div Ra Rb Rc **reg[Ra] = reg[Rb] / reg[Rc]**
Put into *Ra* the contents of *Rb* **divided** by the contents of *Rc*.

and Ra Rb Rc **reg[Ra] = reg[Rb] & reg[Rc]**
Put into *Ra* the contents of *Rb* **bitwise-and** the contents of *Rc*.

or Ra Rb Rc **reg[Ra] = reg[Rb] | reg[Rc]**
Put into *Ra* the contents of *Rb* **bitwise-or** the contents of *Rc*.

comp Ra Rb **reg[Ra] = ~reg[Rb]**
Put into *Ra* the **bitwise-complement** of the contents of *Rb*.

shr Ra Rb Rc **reg[Ra] = reg[Rb] >> reg[Rc]**
The contents of *Rb* is **shifted right** by the amount specified in register *Rc* and the result is stored in *Ra*. If the value in *Rc* is negative, the value is shifted left by the negative of that amount.

shl Ra Rb Rc **reg[Ra] = reg[Rb] << reg[Rc]**

The value in register *Rb* is **shifted left** by the amount specified in register *Rc* and the result is stored in *Ra*. If the value in *Rc* is negative, the value is shifted right by the negative of that amount.

jeq Ra Rb Rc
Jump to the address in *Ra* if the values in *Rb* and *Rc* are **equal**. Otherwise continue.

jne Ra Rb Rc
Jump to the address in *Ra* if the values in *Rb* and *Rc* are **not equal**. Otherwise continue.

jgt Ra Rb Rc
Jump to the address in *Ra* if the value in *Rb* is **greater than** that in *Rc*. Otherwise continue.

jgte Ra Rb Rc
Jump to the address in *Ra* if the value in *Rb* is **greater than or equal** that in *Rc*. Otherwise continue.

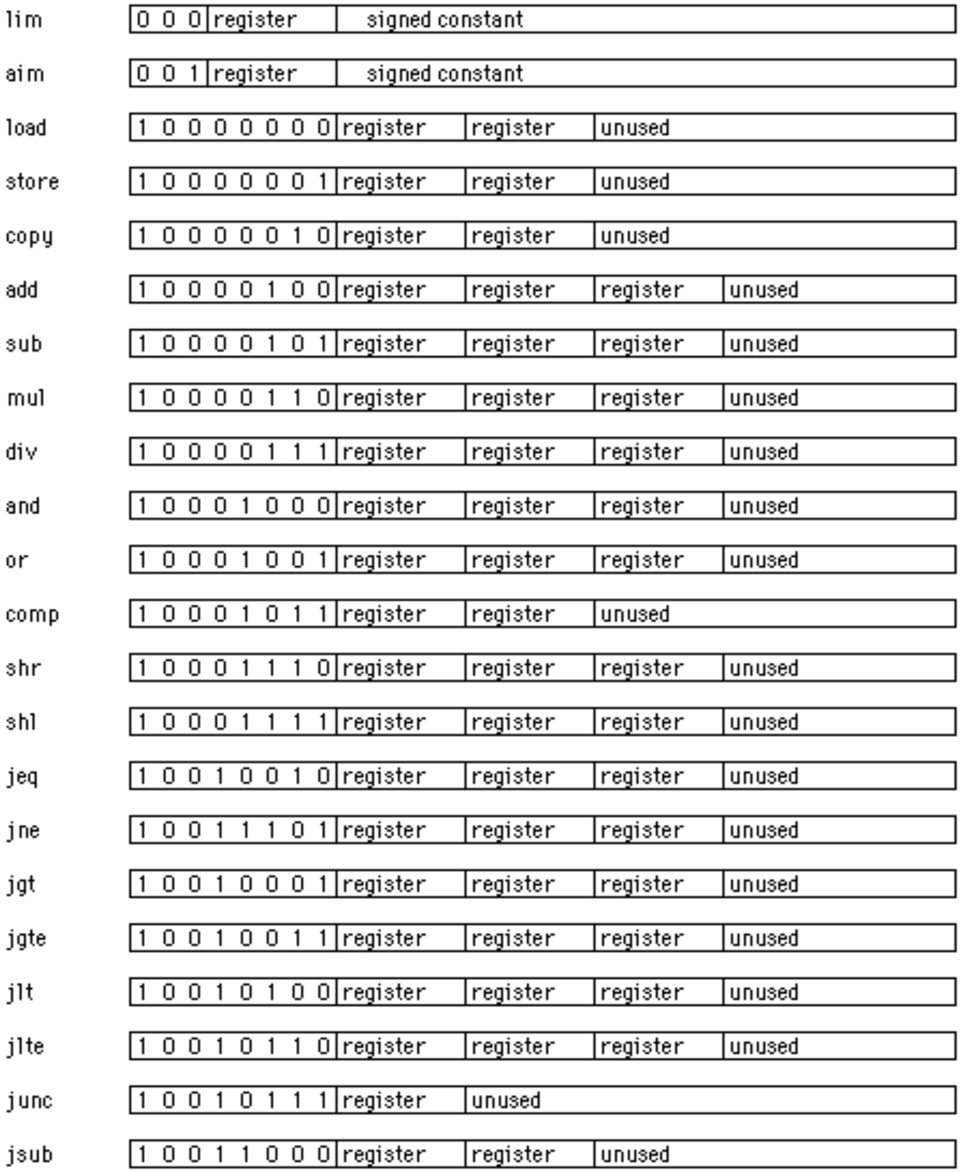
jlt Ra Rb Rc
Jump to the address in *Ra* if the value in *Rb* is **less than** that in *Rc*. Otherwise continue.

jlte Ra Rb Rc
Jump to the address in *Ra* if the value in *Rb* is **less than or equal** that in *Rc*. Otherwise continue.

junc Ra
Jump to the address in *Ra* **unconditionally**.

jsub Ra Rb
Jump to subroutine in the address in *Ra*. The value of the *IP* (i.e. what would have been the next instruction) is put into *Rb*. Therefore this can be used for jumping to a subroutine. If the return address is not needed, some register not in use should be specified.

Although it is not critical for the current explanation, the following shows a plausible formatting of the ISC instructions into 32-bit words, showing possible assignment of op-code bits. Each register field uses five bits. It appears that there is wasted space in many of the instructions. There are techniques, such as having instructions of different lengths, for dealing with this. We are using the present form for simplicity.



lim	0 0 0	register	signed constant		
aim	0 0 1	register	signed constant		
load	1 0 0 0 0 0 0 0	register	register	unused	
store	1 0 0 0 0 0 0 1	register	register	unused	
copy	1 0 0 0 0 0 1 0	register	register	unused	
add	1 0 0 0 0 1 0 0	register	register	register	unused
sub	1 0 0 0 0 1 0 1	register	register	register	unused
mul	1 0 0 0 0 1 1 0	register	register	register	unused
div	1 0 0 0 0 1 1 1	register	register	register	unused
and	1 0 0 0 1 0 0 0	register	register	register	unused
or	1 0 0 0 1 0 0 1	register	register	register	unused
comp	1 0 0 0 1 0 1 1	register	register	unused	
shr	1 0 0 0 1 1 1 0	register	register	register	unused
shl	1 0 0 0 1 1 1 1	register	register	register	unused
jeq	1 0 0 1 0 0 1 0	register	register	register	unused
jne	1 0 0 1 1 1 0 1	register	register	register	unused
jgt	1 0 0 1 0 0 0 1	register	register	register	unused
jgte	1 0 0 1 0 0 1 1	register	register	register	unused
jlt	1 0 0 1 0 1 0 0	register	register	register	unused
jlte	1 0 0 1 0 1 1 0	register	register	register	unused
junc	1 0 0 1 0 1 1 1	register	unused		
jsub	1 0 0 1 1 0 0 0	register	register	unused	

Figure 281: Plausible ISC instruction formatting

13.3 Examples of Machine-Level Programs

Example Add the values in registers 0, 1, and 2 and put the result into register 3:

```
add 3 0  1  // register 3 gets sum of registers 0 and 1
add 3 2  3  // register 3 gets sum of registers 2 and 3
```

Here we use register 3 to hold a temporary value, which is used as an operand in the second instruction.

Example Suppose x is stored in register 0, and y in register 1. Compute the value of (x + y) * (x - y) and put it in register 3. Assume register 4 is available for use, if needed.

```
add 3 0 1   // register 3 gets x + y
sub 4 0 1   // register 4 gets x - y
mul 3 3 4   // register 3 gets (x + y)(x - y)
```

Example Add the contents of memory locations 1000 and 1001 and put the result into 1002. Assume registers 0 and 1 are available.

```
lim 0 1000  // get addresses of operands into registers 0
lim 1 1001  //   and 1
load 0 0    // overlay addresses with operands
load 1 1
add 1 0 1   // put sum  in register 1
lim  0 1002 // re-use register 0 for address of result
store 0 1   // store the value in register 1 into 1002
```

Example Assume that register 0 contains the address of the first location of an array in memory and register 1 contains the number of locations in the array. Add up the locations and leave the result in register 2. Assume that registers 0 and 1 can be changed in the process and that registers 3 through 8 can be used for temporaries. Assume that the program starts in location 0.

```
lim 2 0     // initialize sum
lim 3 0     // comparison value
lim 6 10    // address of instruction following this code
lim 7 4     // address of next instruction
jlte 6 1 3  // jump to location 10 if the count is <= 0
load 5 0    // load register 5 from the next memory location
add 2 5 2   // add the next number to the sum
aim 0 1     // add 1 to the array address
aim 1 -1    // add -1 to the count
junc 7      // go back to location 4 and compare
```

Note that location 10 is the next location following this program fragment. This was determined from our assumption that the first instruction is in location 0 and instructions are one word long each. Similarly, the jump unconditionally back to 4 (the address in register 7) is for the next iteration of the loop.

Exercises

1 •• Show how the following could be evaluated using ISC machine language:

The sum of the squares of four numbers in registers.

The sum of the squares of numbers in an array.

2 • Show how an *xor* (exclusive-OR) instruction could be added to the ISC.

3 •• Show how a *mim* (multiply-immediate) instruction could be added to the ISC.

4 •• Show how a *jim* (jump-immediate) instruction could be added to the ISC.

Assembly Language

A reader who has worked through a simple example such as the above will no doubt immediately realize a need to invent a symbolic notation within which to construct programs. When constructing the preceding example program, at the third instruction, we did not know initially to put the 10 into `lim 6 10`, since we did not know where the next instruction following would be. Instead, we put in a **symbol**, say xx, to be resolved later. Once all the instructions were in place, we counted to find that the value of xx should be 10. This kind of record keeping becomes tedious with even modest size programs. For this reason, a computer program called an **assembler** is usually used to do this work for us. In an assembler, we can use symbolic values that either we equate to actual values or, as in the case of the address 10 above, the assembler will equate automatically for us. The assembler, not the programmer, does the counting of locations. This eliminates many possible errors in counting and is of exceptional benefit if the program needs to be changed. In the latter case, we would have to go back and track down any uses of addresses. We call the assembly language for the ISC **ISCAL** (ISC Assembly Language). The previous program in ISCAL might appear as:

```
      lim 2 0            // initialize sum
      lim 3 0            // comparison value
      lim 6 done_loc     // address of instruction following this code
      lim 7 loop_loc     // address of next instruction
label loop_loc           // implicitly define label 'loop'
      jlte 6 1 3         // jump to location 10 if the count <= 0
      load 5 0           // load register 5 from the next location
      add 2 5 2          // add the next number to the sum
      aim 0 1            // add 1 to the array address
      aim 1 -1           // add -1 to the count
      junc 7             // go back and compare
label done_loc           // implicitly define label 'done'
```

The readability of the code is also considerably improved through the use of **mnemonic labels** in place of absolute addresses. Note that, in contrast to the other instructions, the lines beginning with label are not executable instructions, but rather merely **directives** that define the labels *loop_loc* and *done_loc*. The general term for such directives in the jargon is **pseudo-op**, for "pseudo-operation". The label pseudo-op equates the identifier following the label to the address of the next instruction. This allows us to use that label as an address and load a register with, in preparation for jumping to that instruction.

Other pseudo-ops of immediate interest in ISCAL are:

origin Location Indicates that the following code is to be loaded into successive locations starting at Location.

define Identifier Value Causes the assembly-time value of Identifier to be equated to the integer value given.

We can take the idea of symbolic names a step further by allowing symbolic names for *registers* in place of the absolute register names. Let us agree to call the registers by the following names in this example:

0	array_loc
1	count
2	sum
3	zero (for comparing against)
5	value (one of the array elements)
6	done
7	loop

One way to equate the symbolic names to the register numbers is through the use of the **register** pseudo-op. Using this pseudo-op, the code would then appear as:

```
      register array_loc 0
      register count 1
      register sum 2
      register zero 3
      register value 5
      register done 6
      register loop 7
      ...
      lim sum  0                     // initialize sum
      lim zero  0                    // comparison value
      lim done done_loc              // address of instruction following
      lim loop loop_loc              // address of next instruction
label loop_loc
      jlte done count zero           // jump if <= 0
      load value array_loc           // load register next array value
      add sum value sum              // add the next number to the sum
      aim array_loc 1                // add 1 to the array address
      aim count -1                   // add -1 to the count
      junc loop                     // go back and compare
label done_loc
```

Note that *array_loc* is assumed to be initialized before we get to the executable code, e.g. this takes place somewhere within “...” . In order to use a jump instruction, we would normally expect to see a preceding *lim* instruction that loads an address into a jump target register. Above, both done and loop are used as jump target registers. Note that the *lim* instruction need not be immediately before the jump, although it often is. In the case of loops, for example, the target is sometimes left in its own register that is only loaded once, at the beginning of the loop sequence.

In the code above, the computation, for the most part, did not depend on specific registers being used. To avoid manually assigning register indices to registers when it doesn't matter, the ISC assembler provides another pseudo-op to automatically manage register indices. This is the **use** pseudo-op. When the assembler encounters the use pseudo-op, it attempts to allocate a free register of its choice to the identifier. Registers that have not been identified in register pseudo-ops, or in previous use pseudo-ops, are assumed to be free for this purpose. Furthermore, a register, once used, can be released by naming it in the **release** pseudo-op. Keep in mind that *use* and *release* are not executable instructions. They are interpreted in a purely textual fashion when the assembler input is scanned.

Let's rewrite the preceding code using *use* and *release*. We will assume that *array_loc*, *count*, and *sum* are to be kept as fixed registers, since they must be used to communicate with other code, i.e. they are not arbitrary.

```
register array 0
register count 1
register sum 2

use loop
use zero                    // register to hold zero
use value
use done
    lim sum  0              // initialize sum
    lim zero 0              // comparison value
    lim done done_loc       // address of instruction following
    lim loop loop_loc       // address of next instruction
label loop_loc
    jlte done count zero    // jump if <= 0
    load value array        // load register next array value
    add sum value sum       // add the next number to the sum
    aim array 1             // add 1 to the array address
    aim count -1            // add -1 to the count
    junc loop               // go back and compare
label done_loc
```

Procedures and Calling Conventions

It is common to have specific calling conventions with respect to registers used for procedure entry and exit. This helps standardize the compilation process. An example might be:

Use register 0 for the return address.
Use register 1 for the returned result.
Use register 2 for the first argument.
Use register 3 for the second argument.
....

up to some convened number of arguments. A procedure having more than this number of arguments would transfer the remaining ones through some sort of memory structure. The registers beyond this number are assumed to be available for internal use within the procedure.

Here is an example of calling a factorial procedure using this convention. There is only one argument.

```
// register definitions

register return 0       // standard return address reg
register result 1       // standard result register
register arg1   2       // first argument register

...

// calling sequence
// get argument in arg1

lim jump_target fac
jsub jump_target return

// use result from result

// procedure definition

label fac     // iterative factorial routine
              // initializes counter 'count' with argument value 'arg'
              // initializes an accumulator with value 1
              // repeats as long as counter greater than 0
              //    multiply accumulator by counter
              //    decrement counter
use zero
      lim zero 0
      lim result 1              // seed result with 1
      lim jump_target test      // set up for loop
label test
      jlte return arg1 zero     // return if arg is 0 or less
      mul  result result arg1   // multiply acc value by counter
      aim arg1 -1               // subtract 1 from the down counter
      junc jump_target          // jump back to the test
```

If such a convention is to be observed, then additional care must be taken when nested procedure calls are present. For example, if a main program calls procedure A, the return address to the point in main is put in register *return*. If A calls B, then the return address to the point in A is put into *return*. Thus, before calling B, A should save the contents of *return* somewhere, e.g. another register or a special location in memory. Following the return from B, and before returning, A should either return to the alternate register or restore *return* to what it was before B was called.

The following code demonstrates return address saving in a procedure that calls fac twice: given argument x, it computes fac(fac(x)). [Note: "nested refers here to fac_fac calling fac, not to the nesting fac(fac(x)).]

```
label fac_fac                 // calls fac(fac(arg))
use return2                   // return2 avoids clobbering return reg
      copy return2 return     // save original return
      lim jump_target fac     // call fac the first time (original arg)
      jsub jump_target return

      copy arg1 result        // copy result to argument register

      lim jump_target fac
      jsub jump_target return // call fac on the result

      junc return2
```

In the example above, we had no need to save the original argument of fac_fac. However, in some cases, we will need to use the original argument again after making the inner call. In this event, the *argument* too must be saved, much in the same manner as the return address.

Recursive Procedures in Machine Language

When a procedure is recursive, the technique described above has to be extended. There is generally no a priori limit on the number of levels of nesting. Thus no fixed number of registers nor special memory locations will suffice to store the return addresses and arguments. In this case, we must use some form of stack. There are two ways in which a stack could be used: The argument and return address could be put on the stack by the caller, or they could be put there by the callee, when and if it makes a nested call. In the following code, we use the latter method: data are not stacked unless a nested call is made. In either case, the stack itself must be set up beforehand. Once we are in the procedure, it is too late, as the procedure assumes the stack is present if needed.

A stack here will be implemented simply as an array in some otherwise unused area of memory. The code below does not check for stack overflow. Adding appropriate code for this is left as an exercise.

```
// set up stack
      lim stack_pointer save_area_loc  // initialize stack pointer
      aim stack_pointer     -1         // always point to top of stack
...

label fac     // recursive factorial routine

      lim result 1               // basis is 1
      jlte return arg zero       // return if count is 0 or less

      aim  stack_pointer +1      // increment stack pointer
      store stack_pointer return // save return address on stack

      aim  stack_pointer +1      // increment stack pointer
      store stack_pointer arg    // save argument on stack

      aim arg -1                 // subtract 1 from argument

      jsub jump_target return    // call recursively

      load arg stack_pointer     // restore original arg
      aim  stack_pointer -1

      load return stack_pointer  // restore original return address
      aim  stack_pointer -1

      mul  result result arg     // multiply by original arg

      junc return                // return to caller

...

label save_area_loc              // first location in save area
```

There are many ways to optimize the code above. But the purpose of the code is to exemplify recursive calling, not to give the best way to compute factorial.

The following diagram shows the stack growth in the case of calling fac with argument 4.

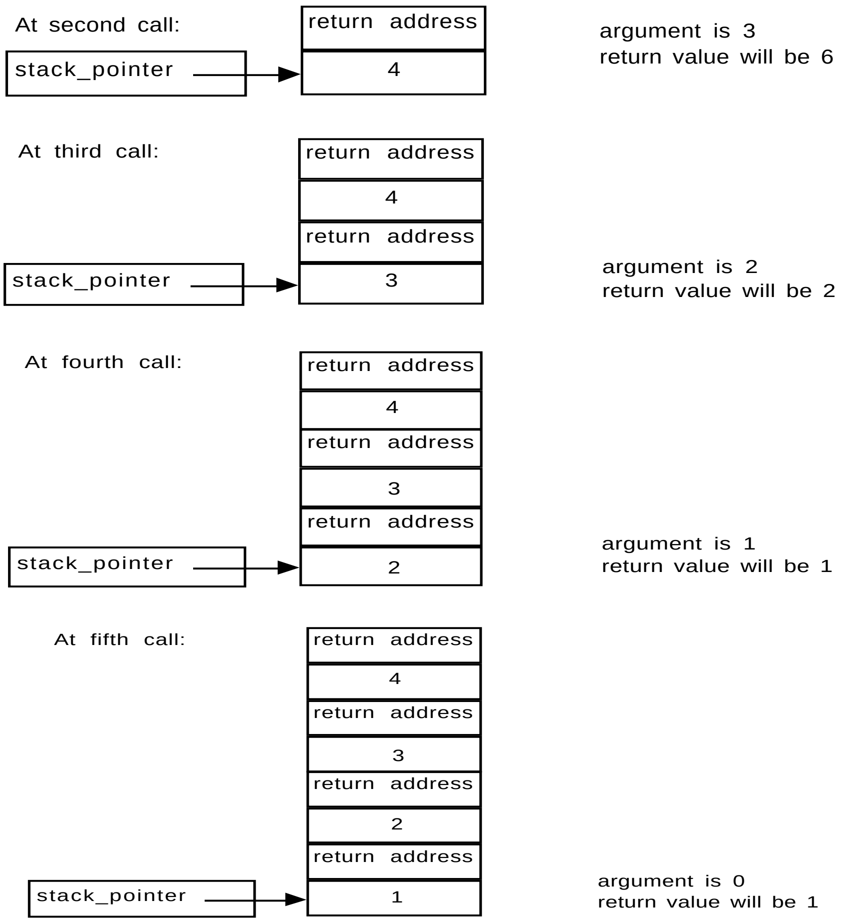


Figure 282: Snapshots of the stack in computing recursive factorial on the ISC

Exercises

1 •• Implement the recursive version of the Fibonacci function in ISCAL. Note: Unlike the case of fac above, the return address values will not always be the same.

2 ••• Implement Ackermann’s function in ISCAL.

3 ••• Try to get rid of some of the recursions in Ackermann's function by converting them to iterations. Can you get rid of all recursion? [Ackermann's function is an example of a function that can be proved to be non-primitive-recursive.]

4 ••• Implement Quicksort in ISCAL.

Switch Statement Equivalents

While we are discussing machine language, it would be worthwhile to see how Java switch statements are compiled to take advantage of the linear addressing principle, as discussed earlier. As mentioned, the idea is that switches over a dense set of values are compiled to an array of jumps. Let us illustrate with an example. Consider the Java code

```
      int i, x, y, z;
      ....
      switch( i )
        {
        case 0:   x = y + z;  break;
        case 1:   x = y - z;  break;
        case 2:   x = y * z;  break;
        case 3:   x = y / z;  break;
        default:  x = 0;      break;
        }
```

An ISC equivalent of this code is shown below. The structure should be understood carefully, as it exemplifies the structure that could be used for any switch statement. There is an **initial** part where outlying cases, those corresponding to the default, are handled. Then there is a **dispatch** part where a jump address is computed by adding to a base jump address an appropriate multiple (in this case 2) of the integer upon which we are switching. Then there are branches, one for each different case and the default. Finally, there is a final part, to which each branch converges.

```
      use temp
      use zero
      use jump_target
      use converge
      lim converge converge_loc      // set up location for converging

// initial part
      lim zero 0
      lim jump_target default_branch
      jlt jump_target i zero         // handle i < 0
      lim temp 3
      jgt jump_target i temp         // handle i > 3

// dispatch part
      lim jump_target branch_array   // set up jump address
      add jump_target i jump_target
      add jump_target i jump_target  // add twice i
      junc jump_target               // jump to branch_array+2*i

label branch_array                   // dispatching array of jumps
                                     // each 2 locations
      lim jump_target branch_0       // case 0
      junc jump_target

      lim jump_target branch_1       // case 1
      junc jump_target

      lim jump_target branch_2       // case 2
      junc jump_target

      lim jump_target branch_3       // case 3
      junc jump_target

// one branch for each default and switch case

label default_branch                 // default case
      lim x 0
      junc converge

label branch_0                       // case 0
      add x y z
      junc converge

label branch_1                       // case 1
      sub x y z
      junc converge

label branch_2                       // case 2
      mul x y z
      junc converge

label branch_3                       // case 3
      div x y z
      junc converge

// converge here
label converge_loc                   // statements after switch
```

13.4 Memory-mapped I/O

There is a notable absence of any I/O (input/output) instructions in the ISC. While I/O instructions were included in early machines, modern architectures prefer to move such capabilities outside the processor itself. Part of the motivation for doing so includes:

> I/O devices are typically slower than computational speeds, so there is a hesitancy to provide instructions that would encourage tying up the processor waiting for I/O.
>
> The wide variety of I/O devices makes it difficult to provide for all possibilities in one processor architecture.

Instead of providing specific I/O instructions, modern architectures use the memory addressing mechanism to deal with I/O devices. These devices are identified with various memory locations, hence the term "memory-mapped I/O". When writing to those locations occurs, detection logic on the memory bus will interpret the contents as intended for the I/O device, rather than as an actual memory write. Thus the variety of I/O devices is essentially unlimited and the processor does not have to take such devices into account.

The most straightforward way to memory map I/O would be to assume a sequential or stream-oriented devices and have one location for input and one location for output. Whenever a read from the input location is issued, the next word in the input is read. Similarly, whenever a write to the output location is issued, a word is sent to the output device. As simple as it is, this picture is slightly undesirable, due to the disparity in speeds between typical I/O devices and processors. If the processor tried to read the location and the device was not ready to send anything, there would have to be a long wait for that memory access to return, during which time the processor is essentially idle. By providing a little more sophistication, there are ways to use this otherwise-idle time. A processor can separate the request for input and checking of whether the next word is ready to be transferred. In the intervening interval, other work could be done in principle. We achieve this effect by having two words per device, one for the datum being transferred and one for the status of the device.

Below we describe one possible memory mapping of an input device and an output device. These would be serial devices, such as a keyboard and monitor.

Location -1 (called **input_word**) is the location from which a word of input is read by the program. Location -2 (called **input_status**) controls the reading of data from the input device and serves as a location that can be tested for input status (e.g. normal vs. end-of-file). In order to read a word, input_status is set to 0 by the program. A write of 0 to this location triggers the input read. When the word has been input and is ready to be read by the program, the computer will change input_status to a non-zero value. The value 1 is used for normal input, while -1 is used for end-of-file condition.

The program should only set input_status to 0 if it is currently 1. If input_status is 0, then a read is already in progress and could be lost due to lack of synchronization. It can be assumed that input_status is initially 1, indicating the readiness of the input device. So a possible input sequence will be something like:

```
input_status = 0;             // start first read
end_of_file = 0;

while( ! end_of_file )
{
  ......                      // other processing can go on here
  while( input_status == 0 )  // wait for read complete
    {}
  switch( input_status )
    {
    case 1:
      use input_word;
      input_status = 0;       // start next read
      break;

    case -1:
      end_of_file = 1;        // indicate done
    }
}
```

Below we show a simpler input reader in ISCAL. This reader can be called as a procedure by the programmer to transfer the next input word. It assumes that the first input word has been requested by setting input_status to 0 earlier on.

```
define input_word_loc    -1 // fixed location for input word
define input_status_loc  -2 // fixed location for input status
use input_status            // register to hold input_status

register return 0       // standard return address reg
register result 1       // standard result register
register arg1   2       // first argument register

      lim input_status  input_status_loc  // setup input status reg
      store input_status zero             // request input
....

label input     // input routine, returns result in register 'result'
use input_word  // register to hold input_word_loc
use jump_target
use zero
use temp                           // temporary register
      lim zero 0
      lim input_word input_word_loc // memory-mapped input
      lim jump_target input_loop    // set up to loop back
label input_loop
      load  temp input_status       // get input status
      jeq   jump_target temp zero   // loop if previous input not ready
      jlt   halt temp zero          // quit if -1 (end-of-file)
      load  result input_word       // load from input word
      store input_status zero       // request next input
      junc  return
```

Output in the ISC gets a similar, although not identical, treatment. The routines are not identical because, unlike input, we cannot request output before the program knows the word to be output. Location -3 (called **output_word**) is the location to which a word of input is written by the program. Location -4 (called **output_status**) controls the writing of data to the output device and serves as a location that can be tested for output status. In order to write a word, output_status is set to 0 by the program, which in turn triggers the output write. When the word has been output, the computer will change output_status to a non-zero value.

It is important that output_status be tested to see that it is not already 0 before changing it. Otherwise, an output value can be lost. It can be assumed that output_status is 1 when the machine is started. So the normal output sequence will be something like:

```
      while( more to be written )
        {
        ......                        // other processing can go on here
        while( output_status == 0 )   // wait for write complete
          {}
        output_word = next word to write;
        output_status = 0;
        }
```

A procedure for output of one word using this scheme in ISCAL is:

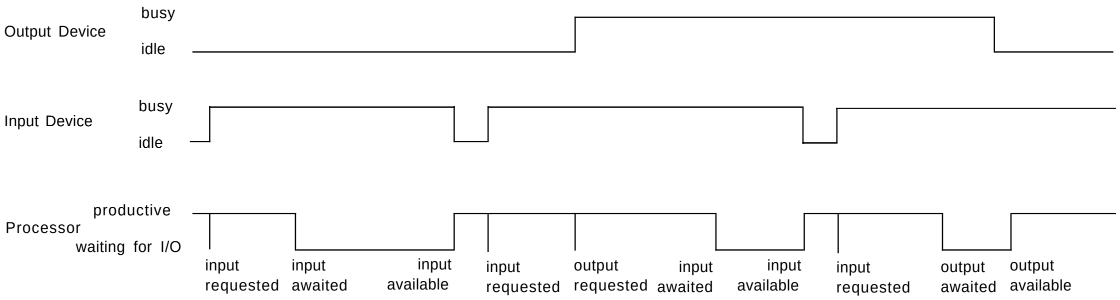
```
define output_word_loc   -3 // fixed location for output word
define output_status_loc -4 // fixed location for output status

register return 0       // standard return address reg
register result 1       // standard result register
register arg1   2       // first argument register

use output_status       // register to hold output_status

label output     // output routine, outputs word in register 'arg1'
use output_word  // register to hold output_word_loc
use jump_target
use zero
use temp                             // temporary register
      lim output_word output_word_loc // memory-mapped output
      lim zero 0
      lim jump_target output_loop     // set up loop address
label output_loop
      load  temp output_status        // get output status in temp
      jeq   jump_target temp zero     // jump back if output not ready
      store output_word arg1          // set up for output of result
      store output_status zero        // request output
      junc  return
```

The timing diagram below shows how these two routines could be called in a loop to keep both the input and output device busy, by requesting input in advance and only waiting when the processor cannot proceed without input or output being complete. This is an example of *overlapped I/O*, that is, input-output overlapped with processing.



Overlapped I/O timing

13.5 Finite-State Control Unit of a Computer Processor

Up to this point, we have seen the ISC primarily from the programmer's viewpoint. Next we look at a possible internal structure, particularly the various finite-state machine components that comprise it. Viewed from the processor, the instructions of the stored program are also a form of "data". The computer reads the instructions as if data from memory and **interprets** them as instructions. In this sense, a computer is an interpreter, just as certain language processors are interpreters.

A typical memory abstraction employed in stored-program computers is as follows: The address of a word to be read or written is put into the MAR (memory address register). If the operation is a *write*, the word to be written is first put into the MDR (memory data register). On command from the sequencer of the computer, the memory is directed to write and transfers the word in the MDR into the address presented by the MAR. If the operation is a *read*, then the word read from the location presented in the MAR is put into the MDR by control external to the processor.

Instructions are normally taken from successive locations in memory. The register IP (instruction pointer) maintains the address of the location for the next instruction. Only when there is a "jump" indicated by an instruction does the IP value deviate from simply going from one location to the next. While the instruction is being interpreted, it is kept in the IR (instruction register). The reason that it cannot be kept in the MDR is because the MDR will be used for other purposes, namely reading or writing data to memory, during the instruction's execution. Unlike the numbered registers used in programming, the registers MAR, MDR, IP, and IR are not directly visible or referenceable by the programmer.

Refer to the ISC diagram below, showing a bus encircling most of the registers in the processor. (In actuality, multiple buses would probably be used, but this version is used for simplicity at this point.) This bus allows virtually any register shown to be gated into any other. At the lower left-hand corner, we see the control sequencer, a finite state machine that is responsible for the overall control of the processor. The control sequencer achieves its effect by selectively enabling the transfer of values from one register to another. The inputs to the sequencer consist of the value in the instruction register and the ALU test bit. Based on these, the sequencer goes through a series of state changes. In each state, certain transfers are enabled.

Every instruction executed undergoes the following **instruction fetch cycle** to obtain the instruction from memory (using Java notation):

```
MAR = IP;      // load the MDR with the address of next instruction
read_memory(); // get the instruction from that address into the MDR
IP++;          // set up for next instruction
IR = MDR;      // move the instruction to the IR
```

The portion of the sequencer for the instruction fetch cycle simply consists of four states. In the first state, the bus is used to gate IP into MAR. If we were to look at a lower level, this would mean that a set of 3-state buffers on the output of the IP is enabled, and a set of AND-gates on the input of the MAR is enabled. In the next state, read_memory is enabled (signaled to the memory controller). In the next state the IP register is incremented (we can build this logic into the register itself, similar to our discussion regarding shift registers), and in the last state of the cycle, the output of the MDR is enabled onto the bus while the input to the IR is enabled.

The description above is simplified. If we had a fast memory, it would pay to do `IP++` at the same time as `read_memory()`, i.e. ***in parallel***, so that we used one fewer clock time for instruction fetch. More likely, we might have a slow memory that takes multiple clock cycles just to read a word. In this case, we would have additional **wait states** in the sequencer to wait until the memory read is done before going on. This would show up in our state diagram as a loop from the memory access state to itself, conditioned on memory not being finished.

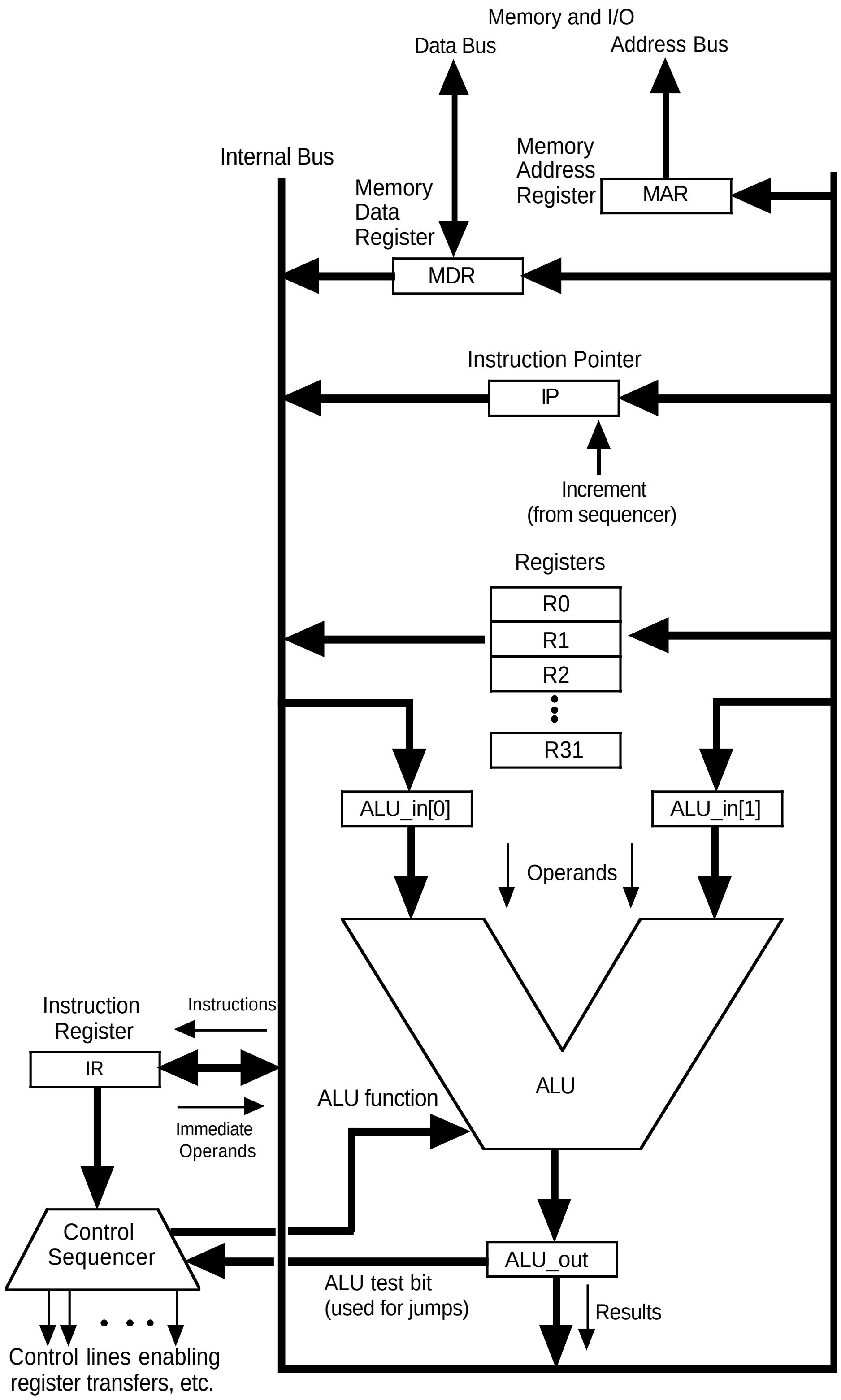


Figure 283: Possible ISC Internal Structure (unoptimized)

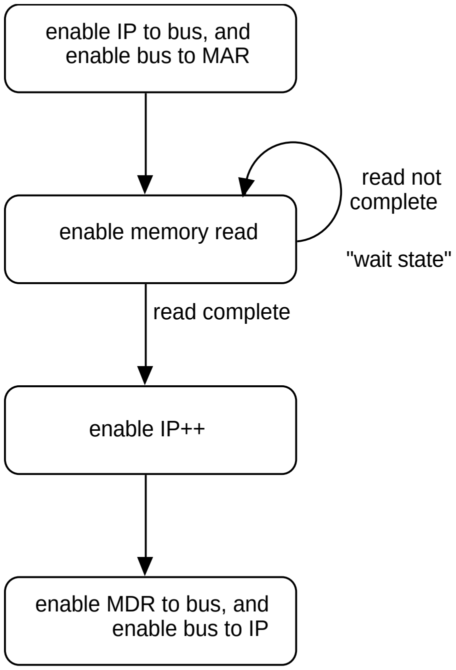


Figure 284: A possible state diagram for the instruction-fetch cycle of the ISC

Once the instruction to be interpreted is in the IR, a different cycle of states is used depending on the bits in the instruction. We give just a couple of examples here:

> If the instruction were **add Ra Rb Rc**, the intention is that we want to add the values in Rb and Rc and put the result in Ra. The sequence would be:

```
ALU_in[0] = Ra;
ALU_in[1] = Rb;
            // add done by the ALU here
Rc = ALU_out;
```

The registers Ra, Rb, and Rc are selected by **decoding** the binary register indices in the instruction and using it to drive 3-state selection (in the case of Ra and Rb) or and-gates (in the case of Rc), as per earlier discussion. We assume here that the combinational addition can be done in one clock interval. If not, additional states would be inserted to allow enough time for the addition to complete.

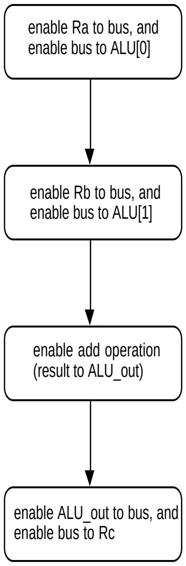


Figure 285: Portion of the ISC state diagram corresponding to the add operation

The ALU is capable of multiple functions: adding, subtracting, multiplying, shifting, AND-ing, shifting, etc. Exactly which function is performed is determined by bits provided by the IR. Most of the instructions involving data processing follow the same pattern as above.

If the instruction were **jeq Ra Rb Rc**, this is an example where the next instruction might be taken from a different location. The sequence would be:

```
ALU_in[0] = Rb;
ALU_in[1] = Rc;
            // comparison is done by the ALU here
if( the result of comparison is equal )
IP = Ra
```

Here Ra contains the address of the next instruction to be used in case Rb and Rc are equal. Otherwise, the current value of IP will just be used.

Overall, then, the behavior of the machine can be depicted as:

```
for( ; ; )
  {
  instruction_fetch();
  switch( IR OpCode bits )
    {
    case add:     add_cycle;        break;
    case sub:     subtract_cycle:   break;
    .
    .
    case jeq:     jeq_cycle;        break;
    .
    .
    }
    }
```

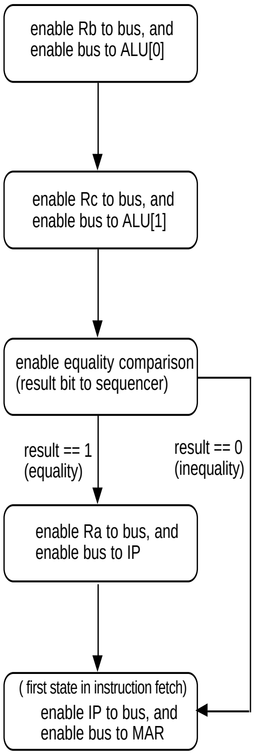


Figure 286: Portion of the ISC state diagram for the jeq instruction

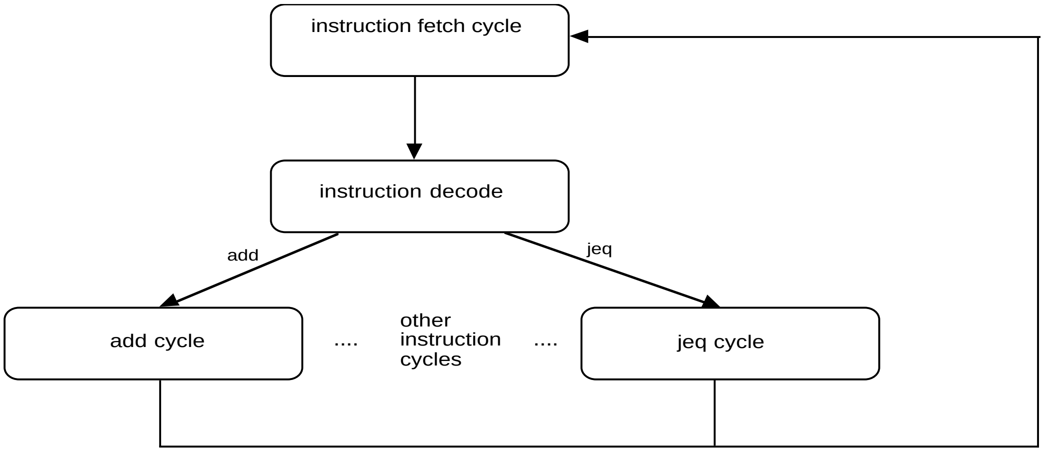


Figure 287: Overall state behavior of the ISC

Exercises

1 •• Based on the above discussion, estimate the number of states in each of the cycles for the various instructions in the ISC instruction set. Obtain an estimate of the number of states in the instruction sequencer. Assume that all memory operations and ALU operations take one clock period.

2 •••• How many flip-flops (in addition to those in the IR) would be sufficient to implement the sequencer? Give a naive estimate based on the preceding question, then a better estimate based on a careful assignment analysis of how functionality in the sequencer can be shared.

3 ••• By using more than one bus, some register transfers that would have been done in sequence can be done concurrently, or "in parallel". For example, in the add cycle, both Rb and Rc need to be transferred. This could, in principle, be done concurrently, but two buses would be required. Go through the instruction set and determine where parallelism is possible. Then optimize the ISC register-transfer structure so as to reduce the number of cycles required by as many instructions as possible.

13.6 Forms of Addressing in Other Machines

As mentioned earlier, the ISC uses register-indirect and immediate addressing only. The following diagrams abstract these two general forms.

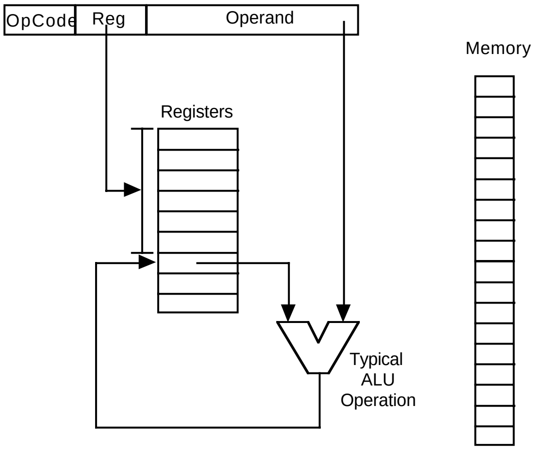


Figure 288: Immediate operand

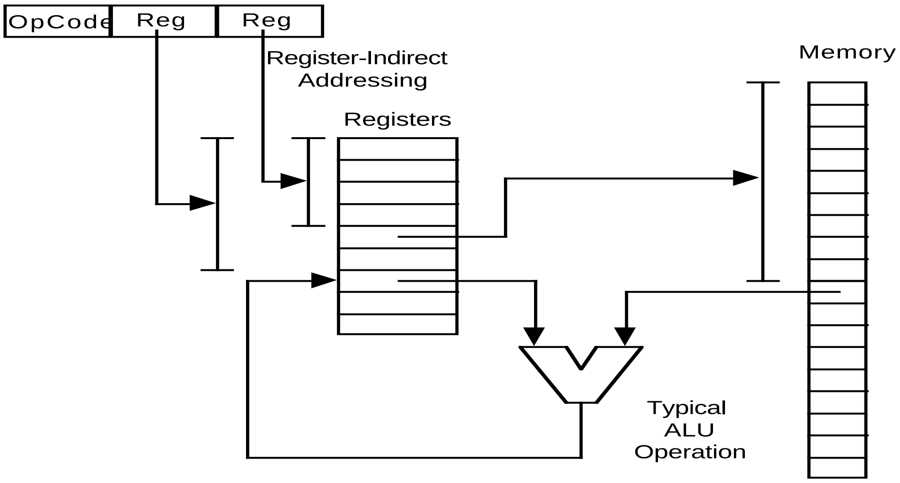


Figure 289: Register indirect addressing

For contrast, other machines might employ some or all of the following types of addressing:

direct addressing – The address of a datum is in the instruction itself. It is not necessary to load a register with the address. The problem with this mode of addressing is that addresses can be very large, making it difficult for a single instruction to directly address all of memory. For example, the ISC's address space is 32-bits, the same size as an instruction. This would leave no space in the instruction for op-code information.

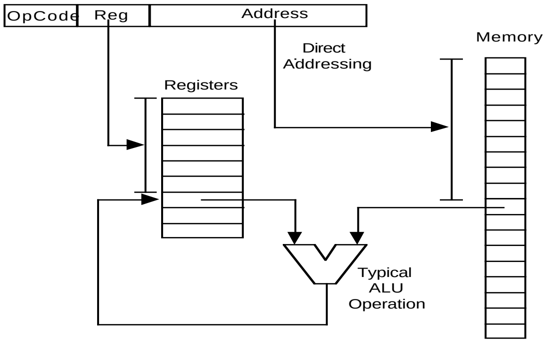


Figure 290: Direct addressing

indirect addressing – The address of the datum is in a word in memory. The instruction contains the address of the latter word. An example of the use of this type of addressing is pointer dereferencing. In a C++ statement

```
x = *p;
```

The address of p would be in the instruction. The contents of p is interpreted as a memory address. The contents of the latter address is stored into a register. The contents of the register would be stored into x by a subsequent instruction (unless the instruction can contain two addresses, one of which would be the address of x).

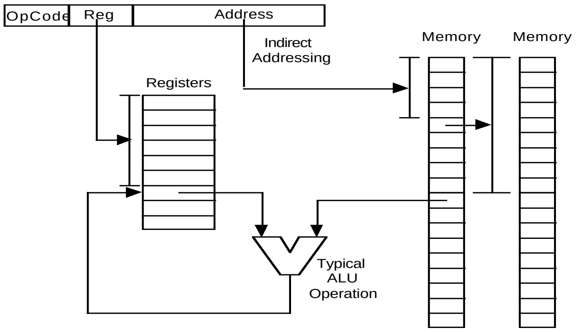


Figure 291: Indirect addressing

indexed addressing – The address of the datum is formed by adding the address in the instruction to the contents of a register, called the **index register**. This sum is called the effective address and is used as the address of the actual datum. In Java or C++, indexed addressing would be useful in indexing arrays. For example, in the statement

```
x = a[i];
```

the instruction could contain the base address, &a[0] and the index register could contain the value of i.

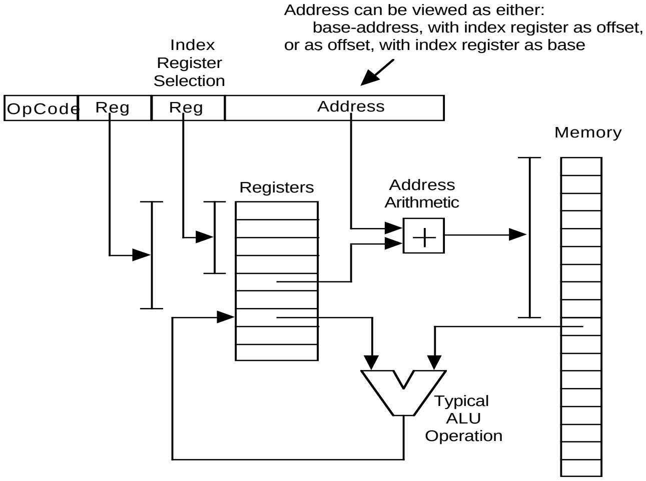

Address can be viewed as either:
base-address, with index register as offset,
or as offset, with index register as base
Index Register Selection
OpCode
Reg
Reg
Address
Memory
Registers
Address Arithmetic
+
Typical ALU Operation

Figure 292: Indexed addressing

based addressing – This is similar to indexed addressing, except that the base address &a[0] is contained in a register. The value of i, called an **offset**, is contained in the word addressed by the instruction.

based indexed addressing – This uses two registers, one containing a base address and one containing an index. The instruction specifies an offset. The effective address is obtained by adding the base address, the index, and the offset. An example of a statement using such addressing would be

```
x = a[i+5];
```

where 5 would be the offset.

There are, of course, other possible combinations of addressing. Machines such as the ISC that do not have all of these forms of addressing must achieve the same effect by a sequence of instructions.

Exercises

1 •• Consider adding a *lix* (load-indexed) instruction to the ISC. This is similar to the load instruction, except that there is an additional register, called the index register. The address of the word in memory to be loaded (called the effective address) is formed by taking the sum of the address register, as in the current load instruction, plus the index register. Show how this instruction could be added to the ISC. Then suggest a possible use for the instruction. To retain symmetry, what other indexed instructions would be worthwhile?

2 •• Explain why a *jix* (jump-indexed) instruction might be useful.

3 •• How could indexing be useful in implementing recursion?

4 •• Give a diagram that abstracts based indexed addressing.

13.7 Processor-Memory Communication

Our diagram of the ISC internal structure omitted details of how the processor and memory interact. We indicated the presence of a data bus for communicating data to and from the memory and an address bus for communicating the address, but other details have been left out. There are numerous reasons for not including the memory in the same physical unit as the processor. For one thing, the processor will fit on a single VLSI chip, whereas a nominal-sized memory will not, at least not with current technology. It is also common for users to add more memory to the initial system configuration, necessitating a more modular approach to memory. Another reason for separation is that memory technology is generally slower than processors. Moderately-priced memory cannot deliver data at the rate demanded by sophisticated processors. However, the memory industry keeps making memory faster, opening the possibility of an upgrade in speed. This is another reason not to tie down the processor to a particular memory speed.

Let us take a look at the control aspects of processor-memory communication. The processor and memory can be regarded as separate agents. When the processor needs data from the memory, it sends a request to the memory. The memory can respond when it has fulfilled the request. This type of dialog is called **handshaking**. The key components in handshaking, assuming the processor is making a read request, are:

a. Processor asserts address onto address bus.

b. Processor tells memory that it has a read request.

c. Memory performs the read.

d. Memory asserts data onto data bus.

e. Memory tells processor that data is there.

f. Process tells memory that it has the data.

g. Memory tells processor that it is ok to present the next request.

The following timing diagram indicates a simple implementation of handshaking along these lines. The transitions are labeled to correspond to the events above. However, step c is not shown because it is implicitly done by the memory, without communication. The **strobe** signal is under control of the processor to indicate initiation of a read. The ack signal is under control of the memory.

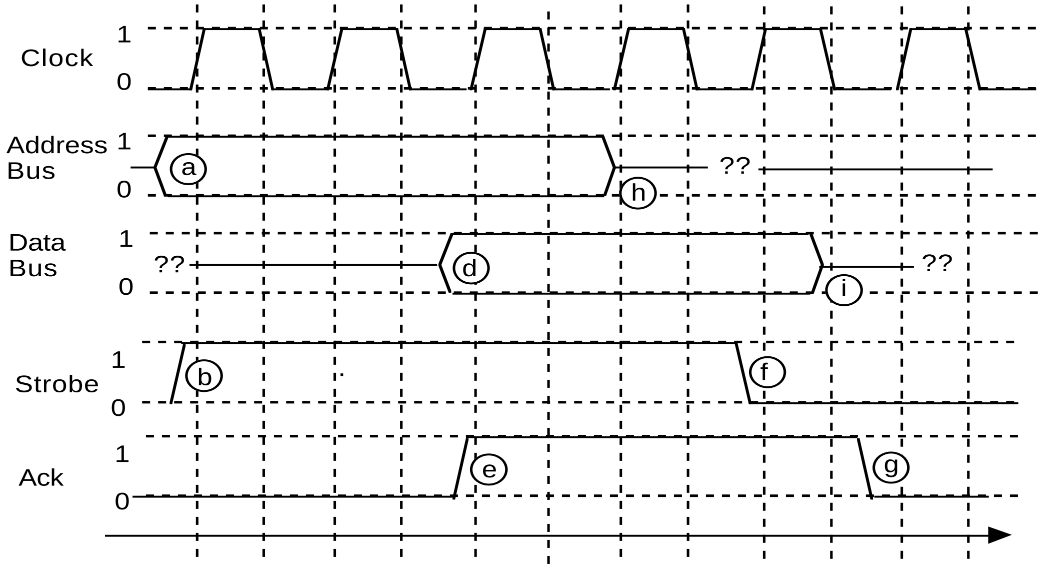


Figure 293: Handshaking sequence for a memory read

The address and data bus lines are shown as indicating both 0 and 1 during some part of the cycle. This means that the values could be either 0 or 1. Since addresses and data consist of several lines in parallel, some lines will typically be each. When the signal is shown mid-way, it means that it is not important what the value is at that point.

Events shown as *h* and *i* in the diagram are of less importance. Event h indicates that once the memory has read the data (indicated by event *e*), the address lines no longer need to be held.

The advantage of the handshaking principle is that it is effective no matter how long it takes for the memory to respond: The period between events *b* and *e* can just be lengthened accordingly. Meanwhile, if the processor cannot otherwise progress without the memory action having been completed, it can stay in a **wait state**, as shown in earlier diagrams. This form of communication is called **semi-asynchronous**. It is not truly asynchronous, since the changes in signals are still supposed to occur between clock signal changes.

The sequence for a memory write is similar. Since reads and writes typically share the same buses to save on hardware, it is necessary to have another signal so that the processor can indicate the type of operation. This is called the **read/write strobe**, and is indicated as **R/W**, with a value of 1 indicating read and a value of 0 indicating a write.

The following table and diagram shows the timing of a write sequence.

a. Processor asserts address onto address bus.

b. Processor asserts data onto data bus.

c. Processor tells memory that it has a request.

d. Memory performs the write.

e. Memory tells processor that write is performed.

f. Processor acknowledges previous signal from memory.

g. Memory tells processor that it is ok to present the next request.

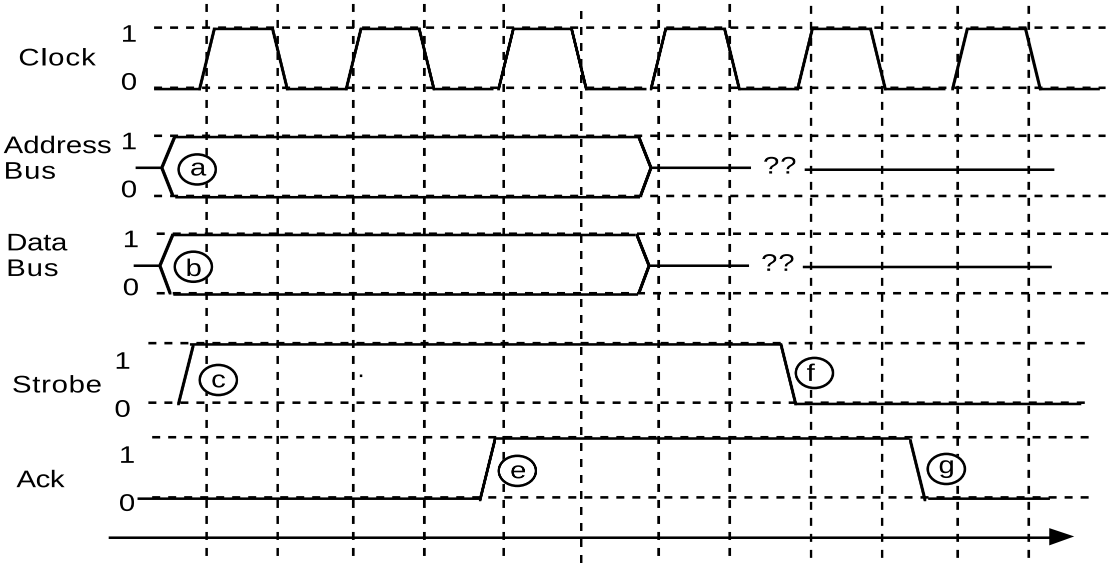


Figure 294: Handshaking sequence for a memory write

Again, it is the responsibility of the RW strobe to convey the type of request to the memory and thereby determine which of the above patterns applies. The handshaking principle is usable whenever it is necessary to communicate between independent sub-systems, not just between processor and memory. The general setup for such communication is shown by the following diagram, where *function strobe* is, for example, the RW line. The sub-system initiating the communication is called the *master* and the sub-system responding is called the *slave*.

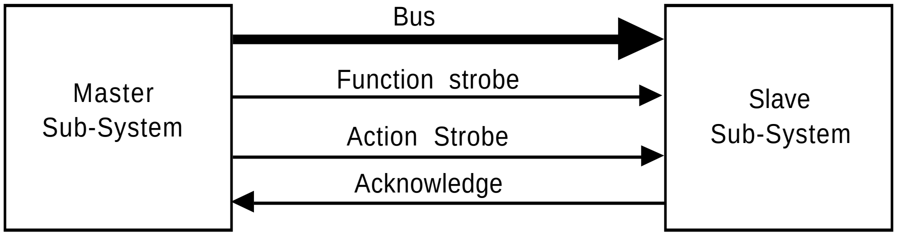


Figure 295: Set-up for communication using handshaking

13.8 Interrupts

When a processor wishes to initiate communication with the outside world, it can use the approach taken here for input/output: writing to certain special memory locations is interpreted by the processor's environment as a directive to carry out some action, such as start an i/o device. It is also necessary to provide a way for the environment to get the attention of the processor. Without such a method, the processor would have to explicitly "poll" the environment to know whether certain events have taken place, e.g. the completion of an i/o operation. The problems with exclusive reliance on polling are the following:

- It is often unclear where the best place is in the program to insert polling instructions. If polling is done too often, time can be wasted. If it is done too seldom, then critical events can be left waiting the processor's attention too long.

- An end-user's program cannot be burdened with the insertion of polling code.

- If the program is errant, then polling might not take place at all.

The concept of "interrupt" is introduced to solve such problems. An interrupt is similar to a procedure call in that there is a return to the point where the program was interrupted (i.e. to where the procedure was called). However, an interrupt is different in that the call is not done explicitly by the interrupted code but rather by some external condition.

The fact that the call is not explicit in the code raises the issue of where the procedure servicing the interrupt is to reside, so that the processor can go there and execute instructions. Typically, there are preset agreed upon locations for this purpose. These locations are aggregated in a small array known as the **interrupt vector**. Typically a special register indicates where this vector is in memory. The interrupt vector is indexed by an integer that indicates the cause of the interrupt. For example, if there are four different classes of devices that can interrupt, there might be four locations in the interrupt vector. The locations within the interrupt vector are address of routines called **interrupt service routines**.

The sequence of actions that take place at an interrupt is:

> The cause of interrupt is translated by the hardware into an index, used to access the interrupt vector.
>
> The current value of the instruction pointer (IP register) is saved in a **special interrupt save location**. This provides the return address later on.
>
> The IP register is loaded with the address specified at the indexed position within the interrupt vector.
>
> Execution at this point is within the interrupt service routine.

> At the end of the interrupt service routine, a **return-from-interrupt** instruction causes the IP to be reloaded with the address in the interrupt save location.
>
> Execution is now back within the program that was interrupted in the first place.

The addition of interrupts has thus necessitated the introduction of one new instruction, return-from-interrupt, to the repertoire of the processor. It also requires a new processor register to point to the base of the interrupt vector. Finally, there needs to be a way to get to the interrupt save location. One scheme for doing this might be to interleave the save locations with the addresses in the interrupt vector. In this way, no additional register is needed to point to the interrupt save location. Furthermore, we have one such location per interrupt vector index. This is useful, since it should be possible for a higher priority interrupt to interrupt the service routine of a lower priority interrupt. Finally, we don't want to allow the converse, i.e. a lower priority interrupt to interrupt a higher priority one. To achieve this, there would typically be an **interrupt mask register** in the processor that indicates which class of interrupts is enabled. The interrupt mask register contents is changed automatically by the processor when an interrupt occurs and when a return-from-interrupt instruction is executed.

Interrupts vs. Traps

Communication with the environment is not the only need for an interrupt mechanism. There are also needs internal to the processor, which correspond to events that we don't want to have to test repeatedly but which nonetheless occur. Examples include checking for arithmetic overflow within registers and for memory protection violations. The latter are designed to keep an errant program from over-writing itself. Sometimes these internal causes are distinguished from interrupts by calling them "**traps**". Traps are also used for debugging and for communicating with the operating system. It is unreasonable to simply allow a user program to jump to the operating system code; the latter must have special privileges that the user program does not. The only way to provide the transfer from an unprivileged domain to a privileged one is through a trap, which causes a change in a set of mask registers that deal with privileges.

13.9 Direct Memory Access I/O

While interrupts assist in the ability for a processor to communicate with input/output devices at high speed, it is often too slow to have an interrupt deal with every word transferred to or from a device. Some devices demand such great attention that it would slow down the executing program significantly to be interrupted so often. To avoid such slow down, special secondary processors are often introduced to handle the flow of data to and from high-speed devices. These are variously known as **DMA** (direct memory access) **channels** (or simply "channels") or **peripheral processors**. A channel competes with the processor for memory access. It transfers an entire array of locations in one single interaction from the processor, maintaining its own pointer to a word in memory that is next to be transferred. Rather than interrupting the processor at every word transfer, it only interrupts the processor on special events, such as the completion of an array transfer.

13.10 Pipelining within the Processor

In order to gain an additional factor in execution speed, modern processors are designed for "pipelined" execution. This means that multiple, rather than a single, instructions are being executed concurrently, albeit at different stages within their execution cycles. For example, instruction n can be executing an add instruction while instruction n+1 is fetching some data from memory. In order for pipelining to work, there must be additional internal registers and control provided that make the net result for pipelined execution be the same as for sequential execution. To give a detailed exposition of pipelining is beyond the scope of the present text. The reader may wish to consult a more advanced text or the tutorial article [Keller 1975].

13.11 Virtual Memory

Virtual memory is a scheme that simplifies programming by allowing there to be more accessible words than there is physical memory space. This is accomplished by "swapping" some of the memory contents to a secondary storage device, such as a disk. The hardware manages the record-keeping of what is on disk vs. in main memory. This is accomplished by translating addresses from the program to physical addresses through a "page table". Memory is divided up into blocks of words known as pages, which contain sets of contiguous storage locations. When the processor wants to access a word, it uses the higher-order so many bits to access the page table first. The page table indicates where the page, either in main memory or secondary storage, and where it is. If the page is in main memory, the processor can get the word by addressing relative to the physical page boundary. If it is on disk, the processor issues an i/o command to bring the page in from disk. This may also entail issuing a command to write the current contents of some physical memory page to disk, to make room for the incoming page. The page idea also alleviates certain aspects of main memory allocation, since physical pages are not required to be contiguous across page boundaries. This is an example of the linear-addressing principle being applied at two levels: once to find the page and a second time to find the word within the page. There is a constant-factor net slow-down in access as a result, but this is generally considered worth it in terms of the greater convenience it provides in programming.

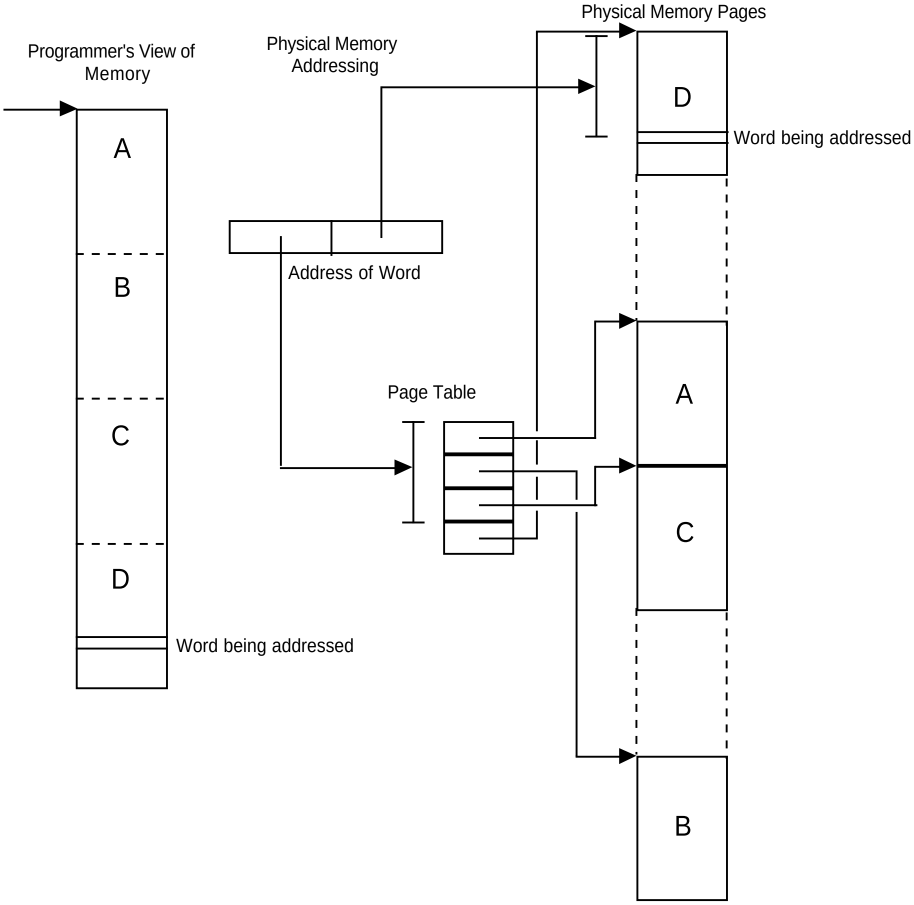


Figure 296: How address translation is done for virtual memory; The physical memory pages could be in main memory or on disk.

13.12 Processor's Relation to the Operating System

Very seldom is a processor accessed directly by the user. At a minimum, a set of software known as the "operating system" provides utility functions that would be too complex to code for the average user. These include:

Loading programs into memory from external storage (e.g. disk).

Communication with devices, interrupt-handling, etc.

A file system for program and data storage.

Virtual memory services, to give the user program the illusion that it has much more memory available than it really does.

Multiple-user coordination, so that the processor resource can be kept in constant use if there is sufficient demand.

The close connection between processors and operating systems demands that operating systems, as well as other software, must be kept in mind when processors are designed. For this reason, it is unreasonable to consider designing a modern processor without a thorough knowledge of the kind of operating system and languages that are anticipated being run on it.

Exercises

1 ••• Modify the design of the ISC to include an interrupt handling facility. Show all additional registers and define the control sequences for when an interrupt occurs.

2 ••• Design a channel processor for the ISC. Show how the ISC would initiate channel operations and how the channel would interact with the interrupt mechanism.

3 ••• Design a paging mechanism for the ISC.

4 ••• A feature of most modern processors is *memory protection.* This can be implemented using a pair of registers in the processor that hold the lower and upper limit of addresses having contents modifiable by the currently-running program. Modify the ISC design to include memory protection registers. Provide an instruction for setting these limit registers under program control.

13.13 Chapter Review

Define the following terms:

assembly language
complex-instruction set computer
direct memory access (DMA)
directives (assembler)
effective address
handshaking
instruction decoding
interpreter
interrupt
linear addressing principle
memory address register
memory data register
recursion
reduced-instruction set computer
stack
strobe
switch statement

trap
virtual memory
wait state

13.14 Further Reading

V. Carl Hamacher, Zvonko G. Vranesic, and Safwat G. Zaky, *Computer Organization*, Third Edition, McGraw-Hill, New York, 1990.

John L. Hennessy and David A. Patterson, *Computer architecture – A quantitative approach*. Morgan Kauffman Publishers, Inc., 1990.

Robert M. Keller. *Look-ahead processors*. ACM Computing Surveys, **7**, 4, 177-195 (December 1975).