Today’s Topics

- Instruction opcodes and operands
- Instruction lengths
- Expanding opcodes
- Instruction formats for the sample architectures
- Types of operands
- Addressing modes
- Stack addressing
- Reverse Polish Notation
- Branch Addressing
- Addressing modes for the sample architectures

Instruction Formats

The computer is driven by the instructions that are stored in memory, and the format of those instructions is part of the Instruction Set Architecture. In very general terms, instructions contain the following two pieces of information:

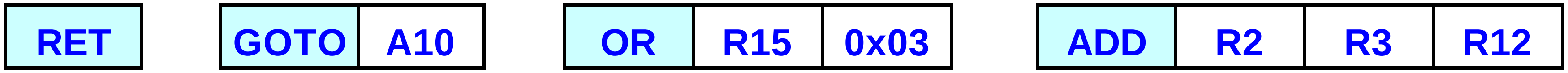
The Opcode

The opcode is the portion of the instruction that tells the CPU which instruction it is. Each different instruction that the ISA is capable of executing has uniquely recognizable pattern of bits. Every ISA instruction contains an opcode. Here are some of the opcodes we saw from the Java Virtual Machine:

Instruction	Hex Opcode	Binary Opcode
ADD	60	01100000
SUB	64	01100100
IRETURN	AC	10101100

The Operands

The operands (referred to as “addresses” in the textbook) tell the CPU what data is to be manipulated by the instruction. Operands can specify data in registers or in memory. ISA instructions can contain 1, 2, 3, or no operands:



return   goto A10   or hex 03 to reg 15   add r2 and r3, result to r12

(Note that the actual instructions contain nothing but “1” and “0” bits – for example the “ADD” opcode above might actually be “01100000”. But we’ve written the opcodes and operands with symbolic names here for clarity.

Instruction Formats

Instruction Lengths

Many ISAs have variable-length instructions. Each instruction starts with an opcode and then additional bit fields are added for each operand that is required:

RET

return

GOTO	A10

goto A10

OR	R15	0x03

or hex 03 to reg 15

ADD	R2	R3	R12

add r2 and r3, result to r12

Some ISAs allow prefix bytes which change the meaning of an instruction:

ILOAD	07

load var #0x7

WIDE	ILOAD	05	2C

load var #0x52C

Some ISAs (such as the UltraSPARC II) have fixed-length instructions. This makes it much easier and faster to fetch and decode instructions, but limits the variety of instruction formats and operand sizes available to the designer:

GOTO	A10

goto label A10

ADD	R2	R3	R12

add r2 and r3, result in r12

An ISA with fixed-length instructions may have instruction sizes that are the same as the size of the general-purpose registers, or the instructions may be longer or shorter.

Instruction Formats

Instruction Design

The opcode and operand portions of an ISA instruction have to be long enough to hold all the possible values needed by the instructions.

Opcode field

The size of the opcode field determines the number of different possible instructions the ISA can have. For example, an 8-bit opcode can have 2^8 = 256 different possible bit combinations, so it could support 256 different instruction types (ADD, SUB, etc.)

Operand fields

The size of the operand fields determine how many different data items can be identified. For example, if the operand field represents a memory address, then the size of the operand determines how much memory can be addressed.

One way to minimize the size of the operand field in an instruction is to have the operand be a register number. There are relatively few registers, so the operand can be fairly small (for example, 16 registers would require only 4 bits for the operand). The register could then, for example, hold a 32-memory address which allows for plenty of memory.

In general, designers try to keep the length of ISA instructions short so as to minimize the memory bandwidth required for instruction fetching. But if this is carried to extremes then it can make the instructions slow and difficult to decode.

Designers also try to leave some opcodes and instruction formats undefined so that there is room to add new capabilities in later generations of the architecture.

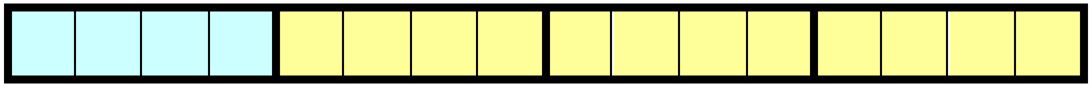
Instruction Formats

Expanding Opcodes

All Instruction Set Architectures require instructions with differing numbers of operands – some instructions might have no operands while others have 1, 2, or 3 operands. Handling differing numbers of operands is a challenge in an ISA that has a fixed instruction length.

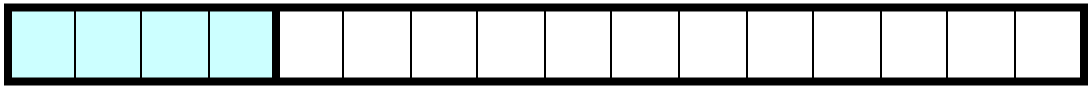
For example, consider an ISA that uses fixed-length 16-bit instructions and has 16 registers. The ISA requires various instructions with 0, 1, 2, and 3 operands.

An instruction for this ISA that has 3 operands would look like this:



opcode operand 1 operand 2 operand 3

…and an instruction that has no operands would look like this:



opcode unused bits

There are two problems here:

1. **The size of the opcode, and therefore the number of possible instruction types, is limited.**
2. **There is a lot of wasted space in 0-, 1-, and 2-operand instructions**

These problems can be expanded though a technique called an expanding opcode. And expanding opcode is one whose size is different for different types of instructions.

Instruction Formats

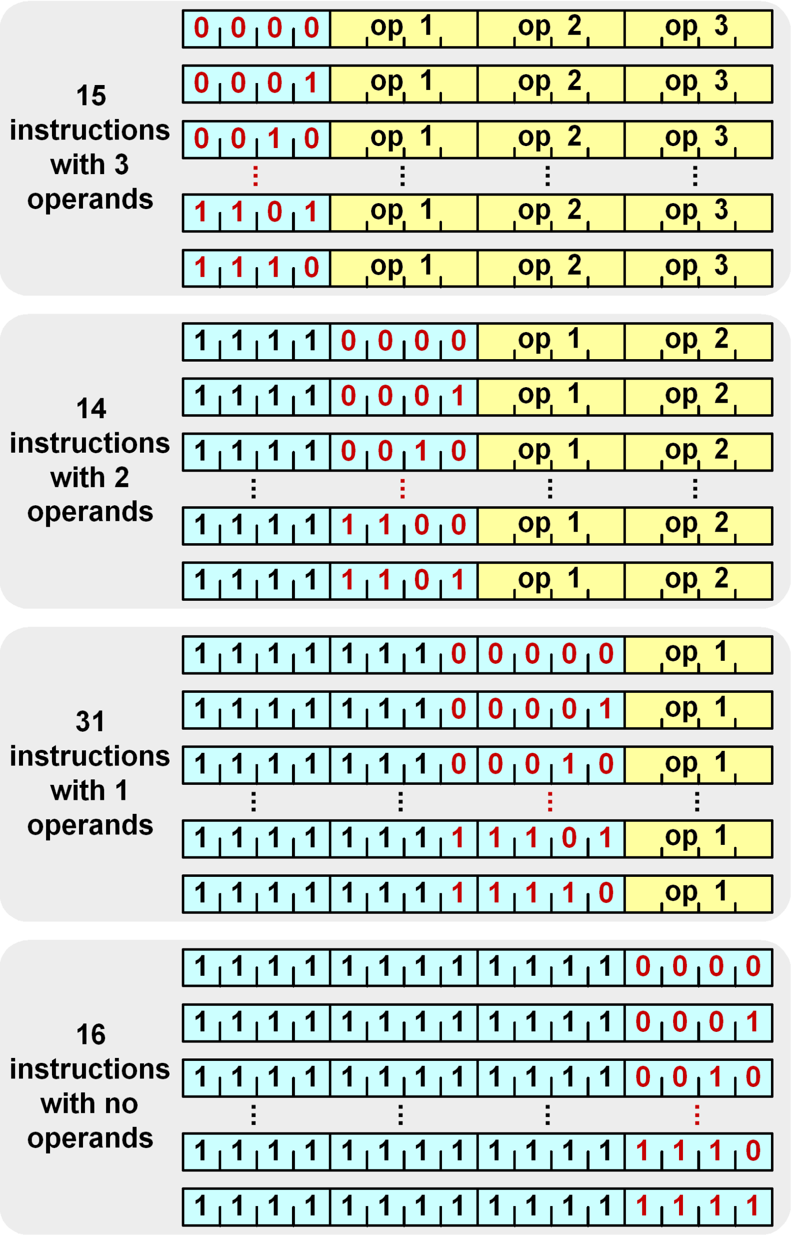
Expanding Opcodes

The example at right shows a series of opcodes for our fixed length 16-bit ISA. Through a clever arrangement of the bits the CPU is able to distinguish among the four different instruction formats by examining the first bits in the instruction:

- If any of the first 4 bits are zero, then it's a 3-operand instruction.
- Otherwise, if any of the first 7 bits are zero, then it's a 2-operand instructions.
- Otherwise, if any of the first 12 bits are zero, then it's a 1-operand instruction.
- Otherwise, it's a zero-operand instruction.

This example provides for 76 different instruction types and eliminates the wasted space we had before.

It's more complex to decode an instruction with expanding opcodes, but the savings in memory bandwidth because of the compact instruction size can more than make up for it.



15 instructions with 3 operands

0 0 0 0	op 1	op 2	op 3
0 0 0 1	op 1	op 2	op 3
0 0 1 0	op 1	op 2	op 3
⋮	⋮	⋮	⋮
1 1 0 1	op 1	op 2	op 3
1 1 1 0	op 1	op 2	op 3

14 instructions with 2 operands

1 1 1 1	0 0 0 0	op 1	op 2
1 1 1 1	0 0 0 1	op 1	op 2
1 1 1 1	0 0 1 0	op 1	op 2
⋮	⋮	⋮	⋮
1 1 1 1	1 1 0 0	op 1	op 2
1 1 1 1	1 1 0 1	op 1	op 2

31 instructions with 1 operands

1 1 1 1	1 1 1 0	0 0 0 0	op 1
1 1 1 1	1 1 1 0	0 0 0 1	op 1
1 1 1 1	1 1 1 0	0 0 1 0	op 1
⋮	⋮	⋮	⋮
1 1 1 1	1 1 1 1	1 1 0 1	op 1
1 1 1 1	1 1 1 1	1 1 1 0	op 1

16 instructions with no operands

1 1 1 1	1 1 1 1	1 1 1 1	0 0 0 0
1 1 1 1	1 1 1 1	1 1 1 1	0 0 0 1
1 1 1 1	1 1 1 1	1 1 1 1	0 0 1 0
⋮	⋮	⋮	⋮
1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 0
1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1

Exercise 1 - Expanding Opcodes

How many opcodes do each of the following instructions have?

Instruction 1: 1111 0111 1111 1101

Instruction 2: 1111 1110 1111 1101

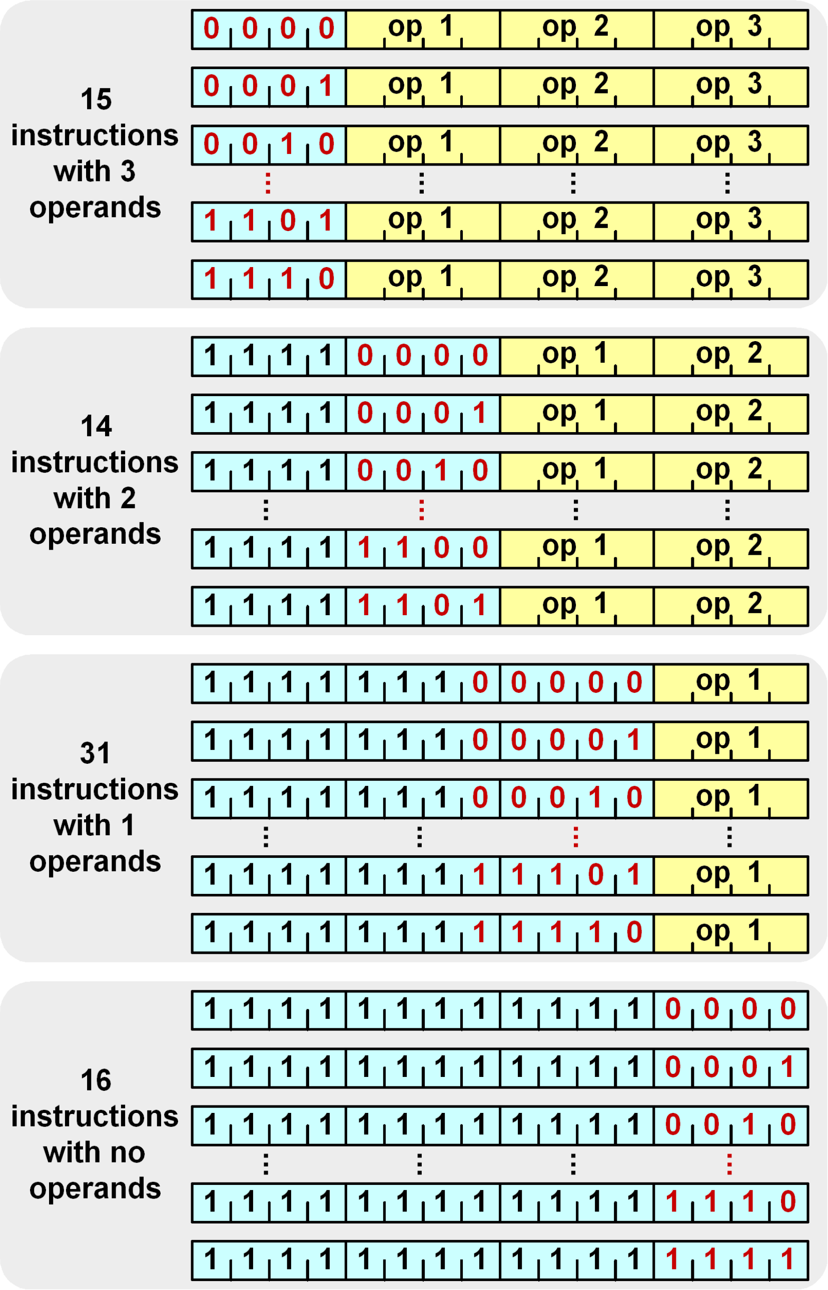
Instruction 3: 1011 1111 1111 1101

A – no operands

B – 1 operand

C – 2 operands

D – 3 operands



15 instructions with 3 operands

```
0 0 0 0 | op 1 | op 2 | op 3
0 0 0 1 | op 1 | op 2 | op 3
0 0 1 0 | op 1 | op 2 | op 3
   ⋮        ⋮      ⋮      ⋮
1 1 0 1 | op 1 | op 2 | op 3
1 1 1 0 | op 1 | op 2 | op 3
```

14 instructions with 2 operands

```
1 1 1 1 | 0 0 0 0 | op 1 | op 2
1 1 1 1 | 0 0 0 1 | op 1 | op 2
1 1 1 1 | 0 0 1 0 | op 1 | op 2
   ⋮         ⋮       ⋮      ⋮
1 1 1 1 | 1 1 0 0 | op 1 | op 2
1 1 1 1 | 1 1 0 1 | op 1 | op 2
```

31 instructions with 1 operands

```
1 1 1 1 | 1 1 1 0 | 0 0 0 0 | op 1
1 1 1 1 | 1 1 1 0 | 0 0 0 1 | op 1
1 1 1 1 | 1 1 1 0 | 0 0 1 0 | op 1
   ⋮         ⋮         ⋮        ⋮
1 1 1 1 | 1 1 1 1 | 1 1 0 1 | op 1
1 1 1 1 | 1 1 1 1 | 1 1 1 0 | op 1
```

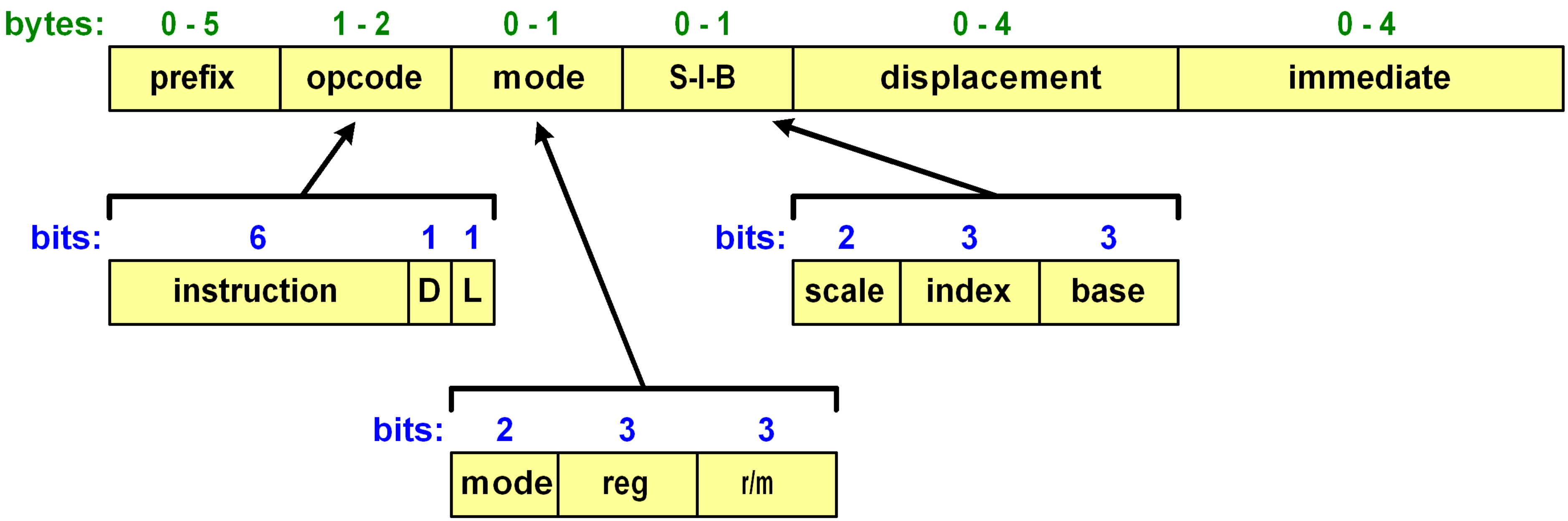
16 instructions with no operands

```
1 1 1 1 | 1 1 1 1 | 1 1 1 1 | 0 0 0 0
1 1 1 1 | 1 1 1 1 | 1 1 1 1 | 0 0 0 1
1 1 1 1 | 1 1 1 1 | 1 1 1 1 | 0 0 1 0
   ⋮         ⋮         ⋮         ⋮
1 1 1 1 | 1 1 1 1 | 1 1 1 1 | 1 1 1 0
1 1 1 1 | 1 1 1 1 | 1 1 1 1 | 1 1 1 1
```

Instruction Formats

Pentium-4 Instruction Formats

The Pentium-4 (and earlier Pentium and IA-32 compatible processors) has a very irregular and complex instruction format:



There can be up to 5 prefix bytes, two opcode bytes, mode and scale bytes, and 4-byte displacement and immediate values for a total possible 17 bytes of instruction!

Only the opcode byte is mandatory, all other portions of the instruction are optional.

The general format is to have one register operand and one register or memory operand.

Instruction Formats

Pentium-4 Instruction Formats

Prefix Bytes

The IA-32 architecture has a plethora of possible prefix bytes which include the following functions:

- Repeat the instruction and decrement the ECX register until it is zero
- Use a different segment than the default one for the specified operand
- Use a different operand size than the default
- Lock memory accesses from other CPUs for the next instruction
- Branch hints

Opcode

Intel used up all of the available opcode combinations for a single byte, so they added a special “escape” opcode byte (0xFF) which directs the CPU to read and decode a second opcode byte.

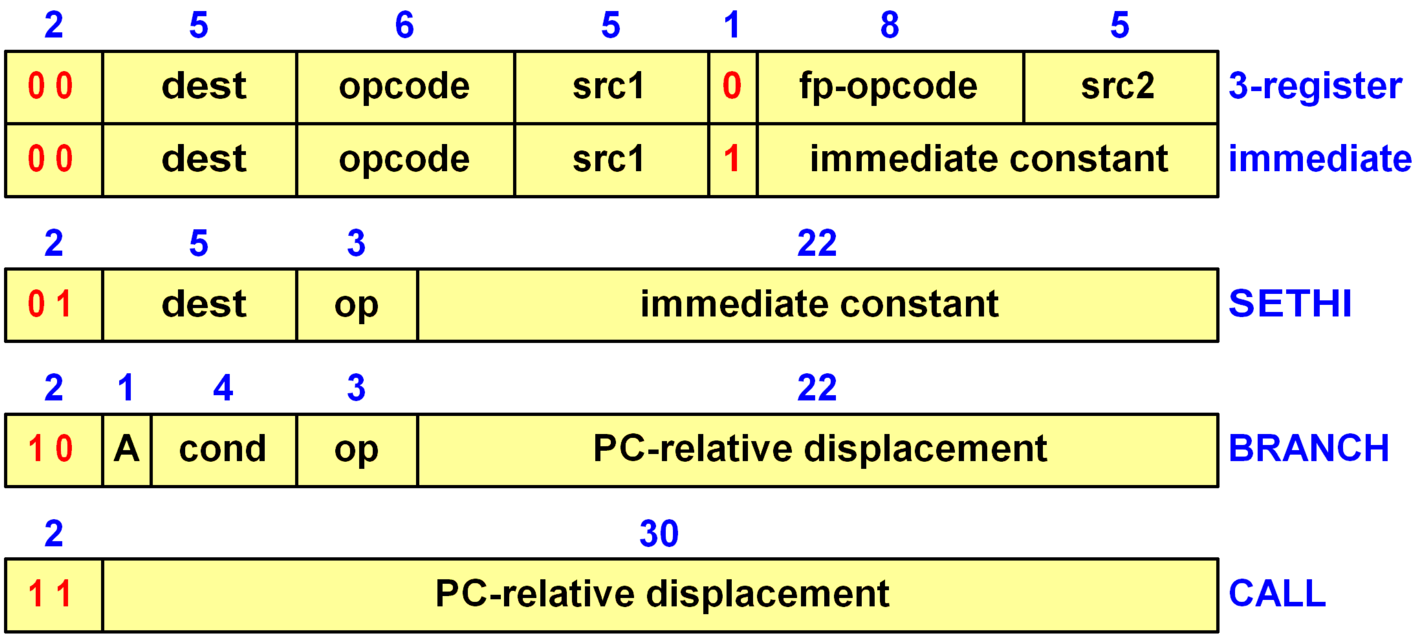
Some opcodes contain one particular bit to indicate the direction (“D”) of the instruction (to or from memory), and a second bit to indicate whether the instruction operates on one byte or one word of data (“L”, for “length”).

The other portions of the instruction relate to how the operands are interpreted. We’ll examine these in more detail shortly when we discuss addressing modes.

Instruction Formats

UltraSPARC III Instruction Formats

The UltraSPARC II uses a fixed-length 32-bit instruction format. The original architecture had a very simple set of instruction formats as befits it's RISC heritage:



2	5	6	5	1	8	5	
0 0	dest	opcode	src1	0	fp-opcode	src2	3-register
0 0	dest	opcode	src1	1	immediate constant		immediate

2	5	3	22	
0 1	dest	op	immediate constant	SETHI

2	1	4	3	22	
1 0	A	cond	op	PC-relative displacement	BRANCH

2	30	
1 1	PC-relative displacement	CALL

The first two bits indicate one of four general instruction formats:

- **00** One of two 3-operand formats whose 3rd-operand is either a register or an immediate value (a source data value stored right in the instruction)
- **01** A unique format used for the SETHI instruction (see below)
- **10** A BRANCH instruction, including compiler hints
- **11** A CALL instruction

Current UltraSPARC chips have many more formats which we won't bother discussing here.

Instruction Formats

UltraSPARC III Instruction Formats

The SETHI instruction

One of the challenges of the UltraSPARC designers was how to allow a program to load a 32-bit constant value into a register. The 32-bit constant value can't be stored as an immediate value in an instruction because the instructions are only 32 bits long, and there would be no room for the opcode telling the CPU what to do.

The designers created the SETHI instruction to solve this problem. The SETHI instruction sets the high-order 22 bits of a 32-bit register. By using the SETHI instruction along with a regular instruction that sets the low-order bits of a register, an entire 32-bit value can be moved into the register.

The CALL instruction

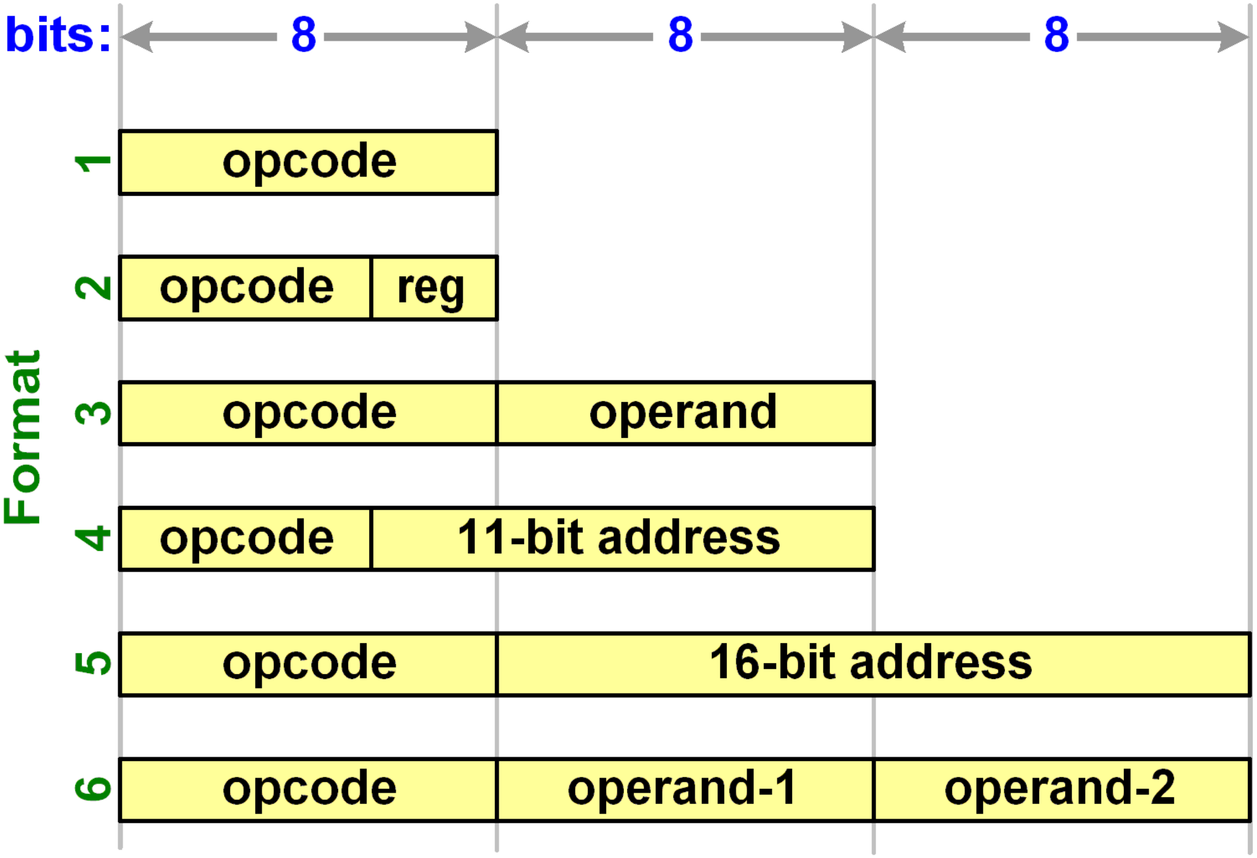
The CALL instruction uses only two opcode bits so that it can specify a 30-bit jump displacement. This gives the CALL instruction the ability to jump to one of 2^{30} addresses (1 billion addresses)

But since all UltraSPARC instructions are exactly 4 bytes long, the call displacement is actually shifted by 2 bits so that any instruction within a 4GB range can be accessed.

Instruction Formats

8051 Instruction Formats

The 8051 system uses variable-length instructions with 6 basic formats.



Format 1 is used by instructions that do not need an operand.

Format 2 is used by instructions that typically work on a register and the accumulator.

Format 3 is used with instructions that have an immediate value – for example a value to be loaded into a register.

Formats 4 and 5 are used for jump and call instructions. Format 4 is often used when there is no external memory attached to the system so all program addresses are no larger than 4195.

Format 6 is used for instructions that require two operands – for example, an instruction that moves an immediate value to an on-chip data memory location.

Addressing

How operands are specified is an important part of the design of an Instruction Set Architecture. Operands are usually the largest portion of an ISA instruction. Some of the ways to reduce the size of operands in an instruction include:

- **Use registers as operands. Since there are relatively few registers, it doesn't take many bits in the instruction to specify them. For example, a system with 16 registers only requires 4 bits to identify which register to use (because 2^4 = 16).**
- **Use implicit operands. An example of implicit operands is to go from three-operand format to two-operand format – the third operand is assumed to be one of the first two operands.**
- **Use stack addressing. An example of stack addressing is the Java Virtual Machine "IADD" instruction – there are no operands in the instruction itself because the operands are automatically assumed to be the top two words on the stack.**

Early computer architectures used a single "accumulator" register which allowed everything to be done with just one-operand instructions. But accumulator-based architectures require lots of memory accesses, and in modern CPUs that is too big a liability.

Addressing

5.4.1

Addressing Modes

The operands to ISA instructions must be able to handle data stored in different places in the computer system. In general, the data that an instruction will used may be located in one of three different places:

In a Register – the instruction must specify which register the data is located in.

In Memory – the instruction must specify the memory address where the data resides, or provide a means to find the address (for example, by identifying a register which holds the memory address)

In the Instruction itself – the instruction must have space to hold the data (this is known as "immediate mode")

So the operands of an ISA instruction may mean different things depending on which of these methods we're using. For example, consider the following instruction:

0001 0101	0000 0011

If the first byte is an opcode that means "PUSH" (put a value onto the stack), then the value that is actually put onto the stack might be:

- **The value that was in register R3, if the operand specifies a register number**
- **The value that was in the memory address specified by register R3, if the operand identifies a memory cell whose address is held in register R3, or**
- **The value "3", if the operand is an immediate mode value (theactual data itself)**

Addressing

Addressing Modes

How does the CPU know which addressing mode is being used by the operands in an instruction? There are two general methods:

- **Addressing modes may be specified by different opcodes. For example, there might be separate opcodes for “PUSH-Register”, “PUSH-Immediate”, and “PUSH-memory”.**
- **An additional “addressing mode” field is included in the instruction. In this method, our “PUSH” instruction from the last page might look like this:**

0001 0101	**mm**	**000011**

…where “mm” represents two bits that specify one of four possible addressing modes.

The latter method is quite common, especially in CISC architectures which have lots of memory addressing modes. When an instruction contains multiple operands it may be necessary to have multiple “addressing mode” fields in the instruction to tell the CPU how to use each operand.

On the next few pages we’ll describe how each of the common addressing modes works.

Addressing

Addressing Modes - Immediate

The simplest addressing mode is "Immediate" mode. An "Immediate" operand specifies the data to be used as part of the instruction itself:

Instruction:

0001 0101	**0000 0011**
(push)	(3)

In this example, the value that is actually pushed onto the stack is "3". Note that the size of the number being used is limited by the size of the operand field in the instruction. Some ISAs allow an immediate value as large as desired to be stored in a suffix field after the other fields in the instruction.

When an immediate operand is written using symbolic language (as in Assembly Language), it's identified by prefixing it with a special character such as "#" or "$":

```
PUSH    #3
```

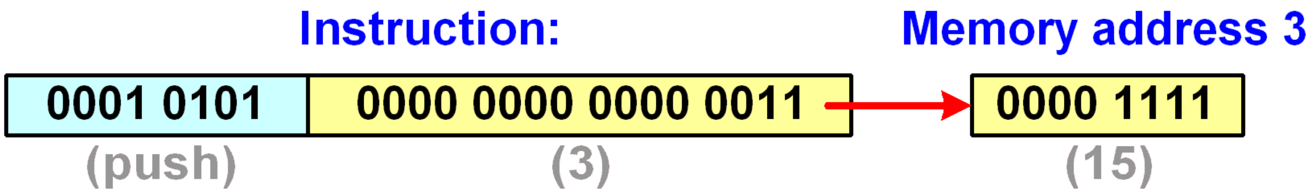
Note that "immediate" mode can only be used for constant "source" operands (operands that the data is coming from). You can never specify an immediate operand as the target of an operation. All the other addressing modes can be used with either "source" or "destination" operands.

Note – in the these addressing mode examples we're not going to show how the addressing mode is identified within the binary instruction – as discussed above it could either be via a unique opcode or via a special "mode" field (not shown) in the instruction.

Addressing

Addressing Modes - Direct

“Direct” addressing mode is used when the data to be used resides in memory and the address of the memory cell is included in the instruction:



Instruction:

0001 0101 | 0000 0000 0000 0011

(push) (3)

Memory address 3

0000 1111

(15)

In this example, the instruction contains a memory address of “3”, so the data stored at memory address “3” is used. That value is “15”, and so “15” is pushed onto the stack.

Since there are many memory addresses to choose from, the size of an operand that uses direct addressing mode must be large enough to specify any memory address. For example in a machine that has 4GB of memory a direct address must be 32 bits long (because 2^{32} = 4G).

When a direct address written using symbolic language (as in Assembly Language), no special is used:

PUSH 3

In practice, specific addresses are rarely coded in Assembly Language – usually the a symbolic variable name is used instead:

PUSH AGE

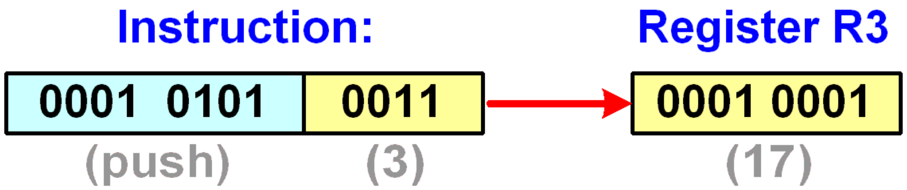
Direct addressing mode can be use for both “source” and “destination” operands.

Addressing

5.4.4

Addressing Modes - Register

“Register” addressing mode is used when the data to be used is in one of the registers and the register number is included as part of the instruction:



Instruction:

0001 0101 | 0011

(push) (3)

Register R3

0001 0001

(17)

In this example, the instruction contains a register number “3”, so the data stored in register “R3” is used. That value is “17”, and so “17” is pushed onto the stack.

Note that not all ISAs have numeric register names, but at the machine level registers are always specified by a bit pattern. So, for example, a bit pattern of “011” (decimal 3) in a Pentium instruction specifies the “ECX” register.

When a direct address written using symbolic language (as in Assembly Language), the register name is used:

PUSH R3

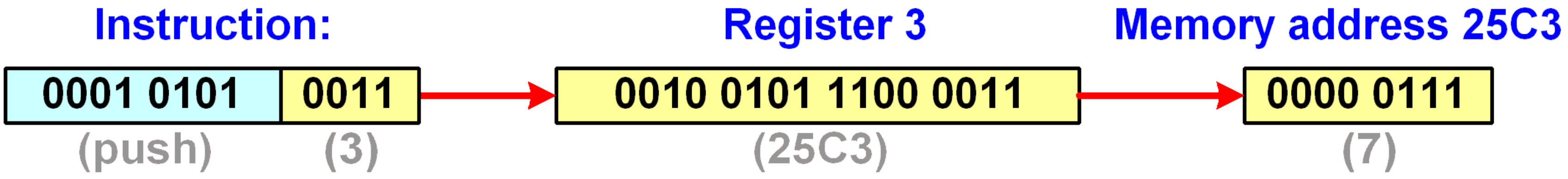
The names of CPU registers are “reserved words” in Assembly Language – it’s not possible to declare a variable that has the same name as a register. Therefore, there’s never any ambiguity between a register name and the name of a variable stored in memory.

Register addressing mode can be used for both “source” and “destination” operands.

Addressing

Addressing Modes – Register Indirect

“Register Indirect” addressing mode is used when the data is in memory and one of the registers holds a “pointer” (the memory address) to where the data is stored:



Instruction:

0001 0101 | 0011

(push) (3)

Register 3

0010 0101 1100 0011

(25C3)

Memory address 25C3

0000 0111

(7)

In this example, the instruction contains a register number “3”, so the address stored in register “R3” is used to find the data in memory. The R3 register contains “25C3”, so the data value stored at that memory address (with a value of “7”) is pushed onto the stack.

When a register indirect address written using symbolic language (as in Assembly Language), the register name is written in parenthesis:

PUSH (R3) or **PUSH [R3]**

Register indirect addressing mode can be used for both “source” and “destination” operands.

Since the memory address is in a register, and since the contents of the register can be changed while the program executes, this addressing mode is very useful when the program has a need to search through or perform the same processing on data stored in different memory locations, as shown in the example on the next page.

Addressing

Addressing Modes – Register Indirect

Here is a small Assembler Language program that shows how Register Indirect addressing mode works. This program determines the length of a null-terminated string in much the same way that the “C” Language “strlen” function does:

```
Text    DB      “Yes”, 0        !Definition of text string:  “Yes” followed by 0 byte

        MOV     ESI, #Text      !Put address of text string into ESI register
A10:
        CMP     [ESI], 0        !Is the byte at memory location pointed to by ESI = 0?
        JE      A20             !  Yes – exit the loop
        ADD     ESI, 1          !Else add 1 to the ESI register so it points to the next byte
        JMP     A10             !   …and go back to check it
A20:
        SUB     ESI, #Text      !Subtract the start address from the end address
```

This example is written in Pentium-II assembly language. In this assembler, the first operand of the instruction is the target of the operation, and the second operand is the source. Text following a “!” character is a comment.

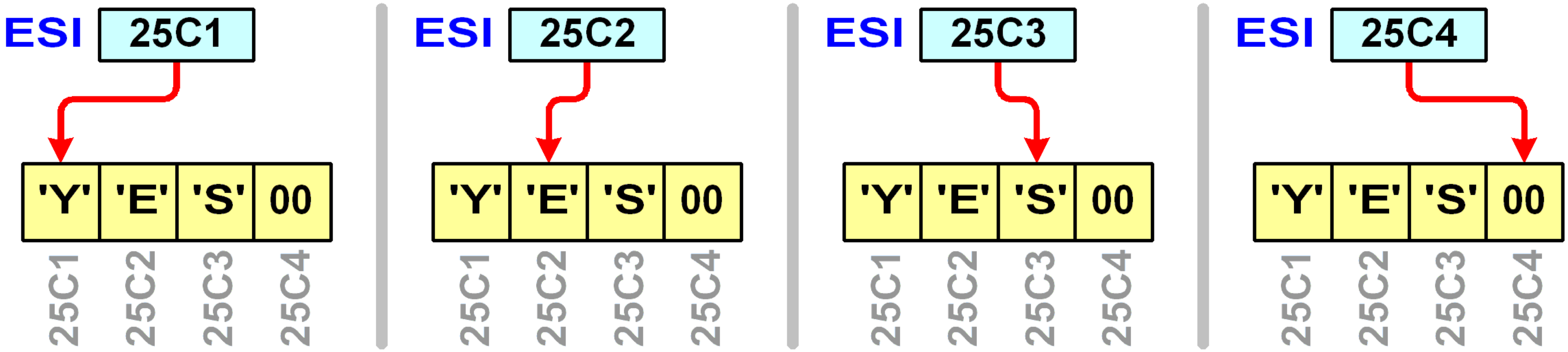
The statements in green are the main body of the loop, and the use of Register Indirect addressing mode is highlighted in red.

The ESI register is initialized with the address of the first byte of the “Yes” string, then each time the loop executes, the address in ESI is incremented by one. So each time the loop executes, the ESI register points to a different character in the string and the “CMP” (compare) statement checks a different value.

Addressing

Addressing Modes – Register Indirect

This diagram shows an example of how the ESI register changes to point to a different character in the string on each pass through the loop is executed:



(The memory addresses shown here are sample addresses – if the string was located at different addresses than these the program would still work correctly)

On the last iteration, the ESI register points to the “0” byte at the end of the string and the “CMP” instruction finds a match. The program then jumps to the last instruction, which subtracts the starting address of the string (25C1) from the address of the byte that contains “0” (25C4) and the result is 3, the length of the string.

Addressing

Addressing Modes – Register Indirect

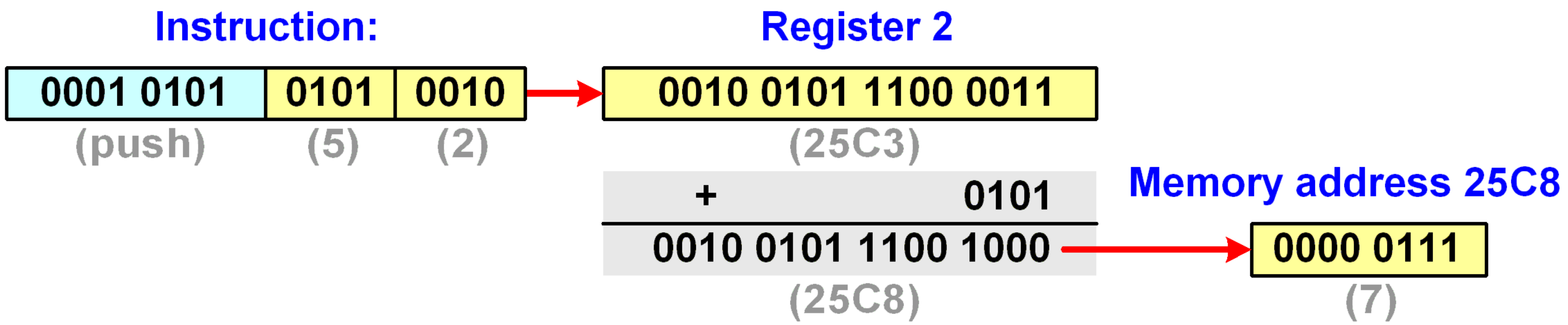
On a machine without Register Indirect addressing mode, the only way to scan memory would be to actually change the address field of an instruction that uses direct addressing mode. Changing an instruction as the program executes is known as "self-modifying code", and it's considered to be a Very Bad Thing for a number of reasons:

- **It's very difficult to debug a program that has modified itself, because the instructions you are trying to debug keep changing.**
- **A single copy of the instructions cannot be shared among multiple processes.**
- **Changing an instruction that is about to be executed may have no effect if it has already been loaded into the pipeline.**
- **Most CPUs with split L1 caches have no mechanism to write I-cache changes back to memory (the designers omit this circuitry to save space because they assume that code will not modify itself)**

Addressing

Addressing Modes – Indexed

“Indexed” addressing mode is used when the data is in memory and one of the registers holds a “pointer” (the memory address) to the area where the data is stored, but the register doesn’t point directly to the data iteself:



Instruction:

0001 0101 (push) | 0101 (5) | 0010 (2)

Register 2

0010 0101 1100 0011 (25C3)

+ 0101

0010 0101 1100 1000 (25C8)

Memory address 25C8

0000 0111 (7)

In this example, the instruction contains a register number “2”, so the address stored in register “R2” (25C3) is used as a “base address”. A second field in the instruction (called the “offset”) (with a value of 5) is added to the base address, and the resulting number (25C8) is then used to find the data in memory. Memory address 25C8 contains the value 7, and so 7 is pushed onto the stack. Note that the register is not changed by the address computation.

When an indexed address is written using symbolic language (as in Assembly Language), the register name is written in parenthesis adjacent to the offset value:

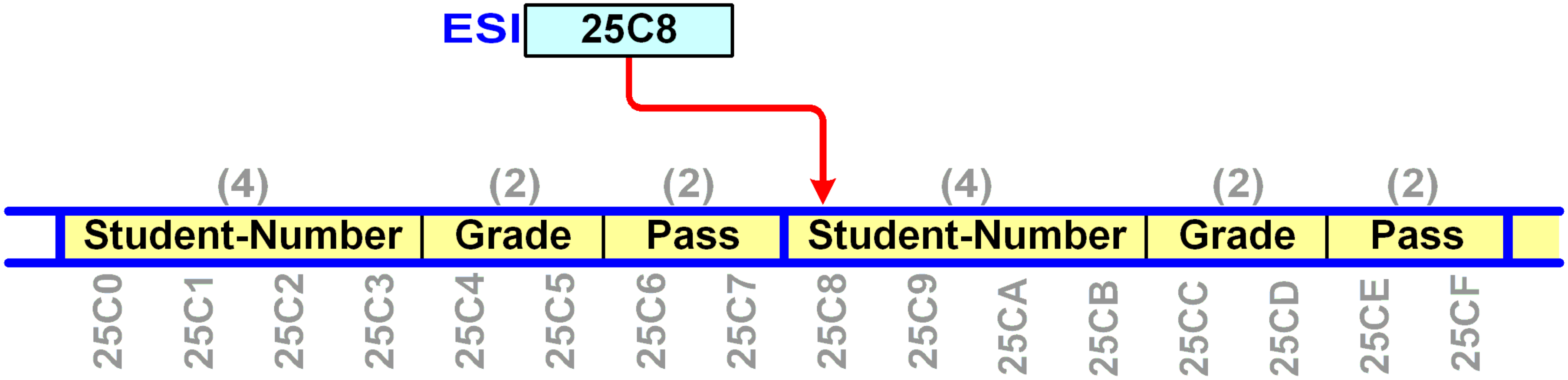
PUSH 5(R2) or **PUSH 5[R2]**

Indexed addressing mode can be used for both “source” and “destination” operands.

Addressing

Addressing Modes – Indexed

Indexed addressing is useful when processing a series of structured data items. Imagine a loop which points the ESI register to each of several student records stored in memory:



An assembler language program can access data that is offset from the ESI address by using indexed addressing mode:

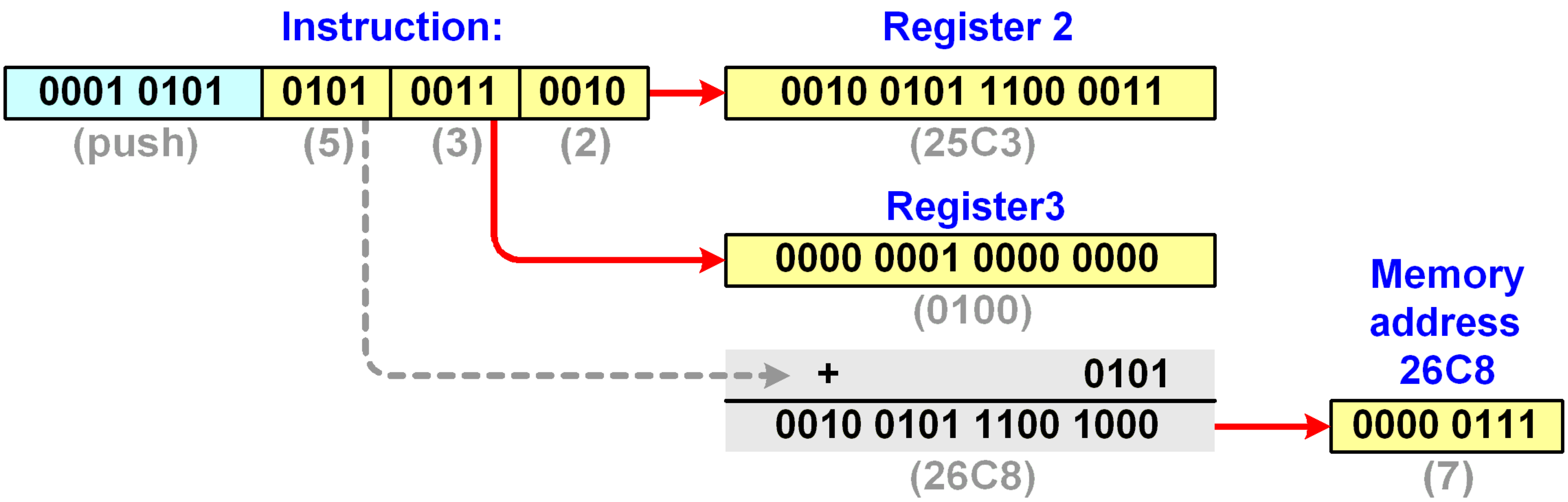
```
        CMP   4[ESI], #50   !Is the grade >= 50?
        JLT   A20           !  No – go around
        MOV   6[ESI], "P"   !Else set “Pass” field to “P”
A20:
```

Note that in practice a real assembler program would define symbolic names for the offsets for the “Grade” and “Pass” fields in the record and use a notation like “Grade[ESI]” rather than “4[ESI]”.

Addressing

Addressing Modes – Based-Indexed

"Based-Indexed" addressing mode uses two registers and an offset:



In this example, the instruction specifies register numbers "2" and "3" and an offset of "5", so the register values (25C3 and 0100) are added to the offset, and resulting number (26C8) is then used to find the data in memory. Memory address 26C8 contains the value 7, and so 7 is pushed onto the stack. Note that the registers are not changed by the address computation.

When a based-indexed address is written in symbolic language (as in Assembly Language), the register names is written with a "+" in parenthesis adjacent to the offset value:

PUSH 5(R2+R3) or **PUSH 5[R2+R3]**

Based-indexed mode can be used for both "source" and "destination" operands.

Exercise 2 – Addressing Modes

Assume that the registers and memory contain the data values shown at the right.

Show what value is pushed onto the stack by each of the following instructions:

Registers:

R0	155
R1	-2
R2	159
R3	100

Memory:

154	377
155	16
156	1053
157	42
158	2478
159	213

1. PUSH (R0)
2. PUSH #155
3. PUSH R2
4. PUSH 158
5. PUSH 2(R0)
6. PUSH 58(R1+R3)

A 16

B 42

C 155

D 159

E 1053

F 2478

Branch Addressing

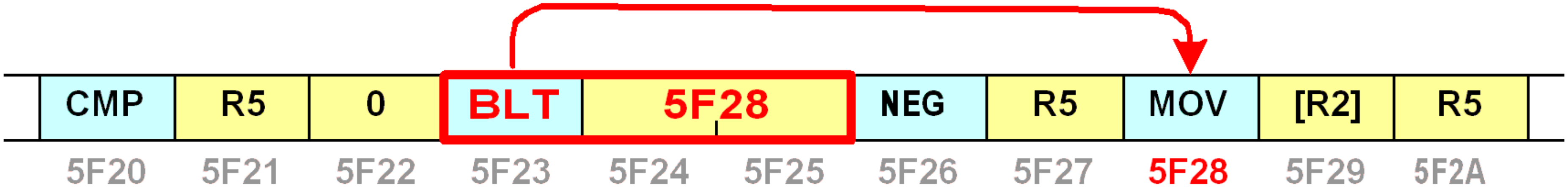
Unlike most other ISA instructions, the operands of branch instructions don’t specify data to be used, but rather the address of another ISA instruction to transfer control to.

There’s no reason why any of the previous addressing we’ve discussed won’t work with branch instructions. But most branch instructions use either direct addressing mode or another mode we’ll discuss shortly called “PC-relative” addressing.

Let’s look again at direct addressing mode, which is often called “absolute addressing”. With this mode, the memory address of the branch target instruction is actually coded as part of the instruction. For example, the following sequence of assembly-language instructions…

```
        CMP     R5,#0      ! Compare R5 to zero
        BLT     SKIP10     !   Branch if Less Than (zero)
        NEG     R5         !  Negate R5 (switches it from – to +)
SKIP10:
        MOV     [R2],R5    !Move result to memory at [R2] address
```

…would turn into something like the following set of ISA instructions stored in memory:



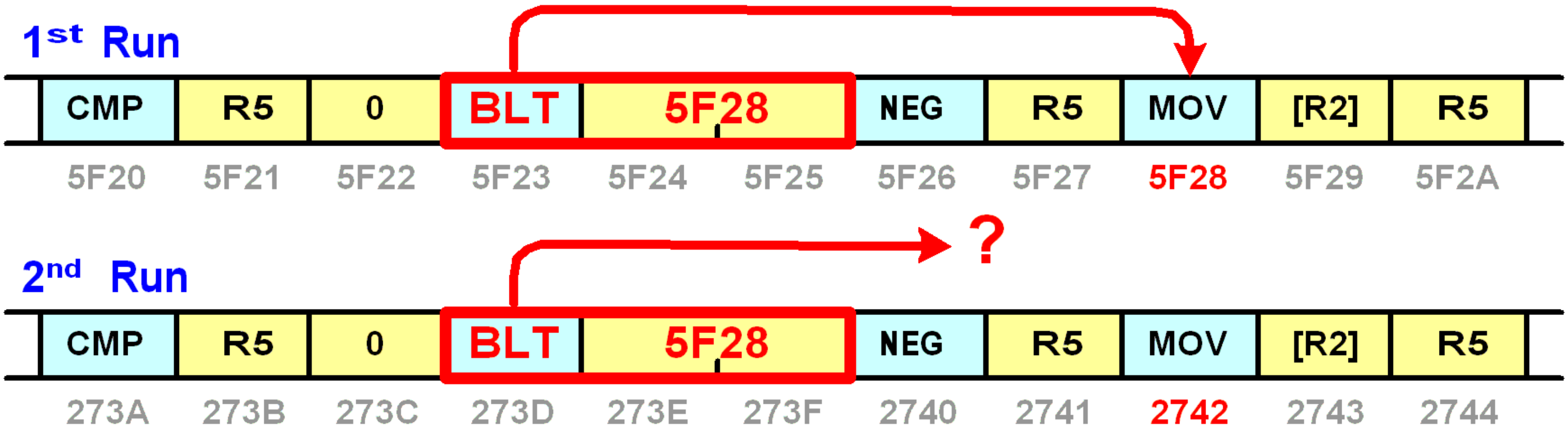
CMP	R5	0	BLT	5F28		NEG	R5	MOV	[R2]	R5
5F20	5F21	5F22	5F23	5F24	5F25	5F26	5F27	5F28	5F29	5F2A

…assuming that the instruction labeled “SKIP10” is loaded into memory at address 5F28

Branch Addressing

Absolute Addressing

Using absolute addressing for branch instructions has a disadvantage: if the program is loaded into different memory locations on different runs then the address in the branch instruction is no longer valid:



1st Run

CMP	R5	0	BLT	5F28		NEG	R5	MOV	[R2]	R5
5F20	5F21	5F22	5F23	5F24	5F25	5F26	5F27	5F28	5F29	5F2A

2nd Run

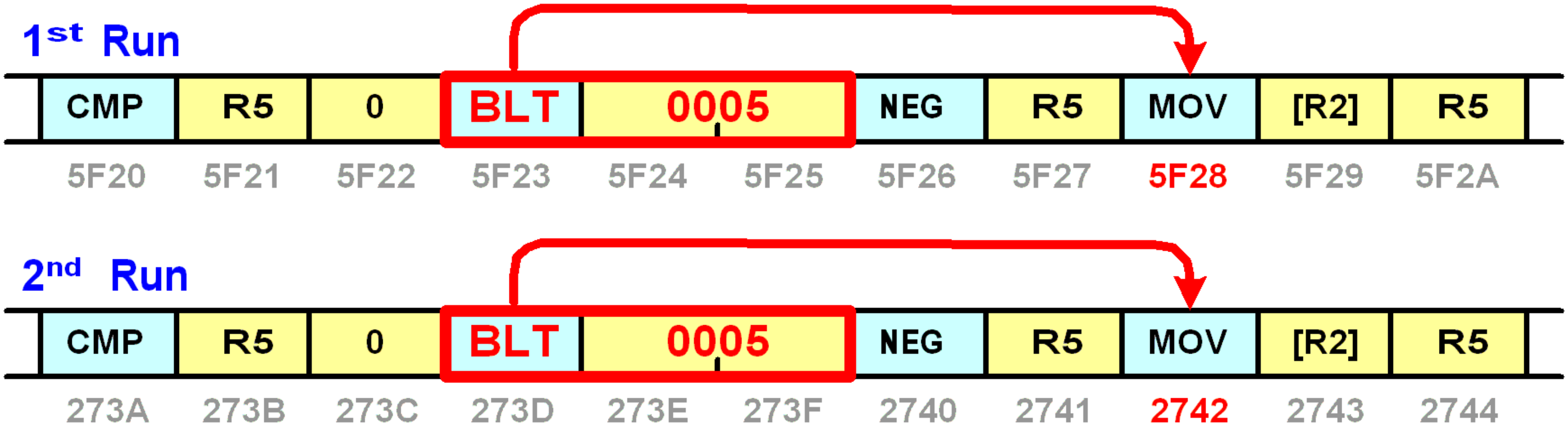
CMP	R5	0	BLT	5F28		NEG	R5	MOV	[R2]	R5
273A	273B	273C	273D	273E	273F	2740	2741	2742	2743	2744

With absolute addressing, the program loader (the piece of software that reads a program from the disk and puts it into memory) has to identify all of the absolute addresses in the program and apply a “fix-up” to them.

Branch Addressing

Relative Addressing

Relative addressing, where the branch instruction contains the distance between the branch instruction and the branch target, eliminates the problem.



1st Run

CMP	R5	0	BLT	0005		NEG	R5	MOV	[R2]	R5
5F20	5F21	5F22	5F23	5F24	5F25	5F26	5F27	5F28	5F29	5F2A

2nd Run

CMP	R5	0	BLT	0005		NEG	R5	MOV	[R2]	R5
273A	273B	273C	273D	273E	273F	2740	2741	2742	2743	2744

If the instruction needs to branch to an earlier instruction, it can use a negative distance.

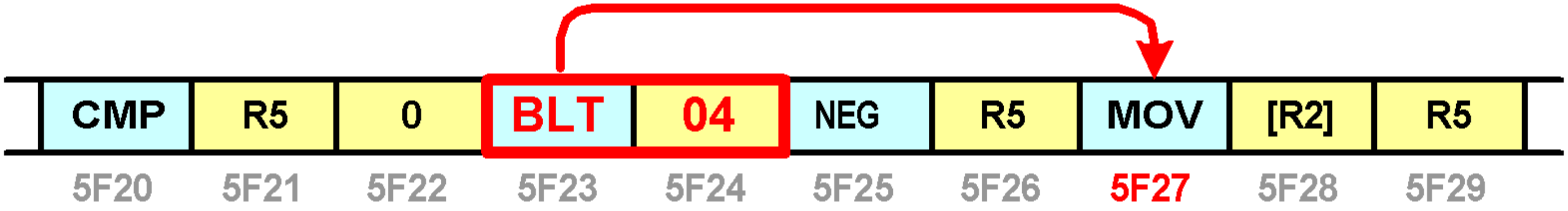
No matter where the program is loaded in memory, the distance from the branch instruction to the target instruction is always 5 bytes – so the instruction never has to be changed. In fact, the same instruction can be used for two different processes in a virtual memory system that have different memory addresses assigned to them.

These kinds of instructions that don't depend on an absolute memory address are known as "position independent code" (PIC).

Branch Addressing

Relative Addressing

Another advantage of relative addressing is that the branch instruction can usually be shorter, because most branch instructions jump to a target instruction that's located nearby. In our branch example, we only had to jump 5 bytes ahead in the program. Since "5" will fit into a single byte, we could use a relative branch instruction that has only an 8-bit offset. This actually reduces the program size by a byte and makes the jump target only 4 bytes away:



CMP	R5	0	BLT	04	NEG	R5	MOV	[R2]	R5
5F20	5F21	5F22	5F23	5F24	5F25	5F26	5F27	5F28	5F29

Making the instruction smaller speeds up the system because it doesn't need to fetch as many bytes from memory.

ISAs that use short offsets for branch instructions need to also provide a long version of the instruction for use in those cases where the branch target is more than about +/-128 bytes away. For example, the Pentium-II uses short offsets for all of the conditional branch instructions, but it has both a short-offset and a long-offset version of the unconditional branch instruction.

Note that "relative" branch addressing is really "Indexed" addressing mode using the PC as the base register.

Exercise 3 - Relative Addressing

Relative addressing has the following advantages:

A – a relative address can always refer to any location in memory

B – relative addresses stored as part of program instructions don't need to be changed if the program is moved to a different memory location

C – relative addresses can often be stored in fewer bytes than absolute addresses

D – A and B above

E – A and C above

F – B and C above

G – all of the above

Addressing Modes and Instruction Formats

As mentioned earlier, there needs to be a way for the CPU to tell which addressing mode is being used. This can be done by having different opcodes for different modes, or by including some bits in the instruction which identify the addressing mode.

The choice of how to identify addressing modes is significant because different addressing modes require different information to be included in the instruction:

Immediate

Needs to include room for the immediate data itself. The size of the immediate data field determines how big the immediate data can be. For example, a 6-bit immediate data field can accommodate unsigned data from 0 – 63 or signed data from -32 to +31.

Direct

Needs to include room for a complete memory address. The size of the memory address field needs to be large enough to accommodate the any address. For example, a system that stores 4GB of data needs to allow for a 32-bit address (because 2^{32} = 4G).

Register and Register Indirect

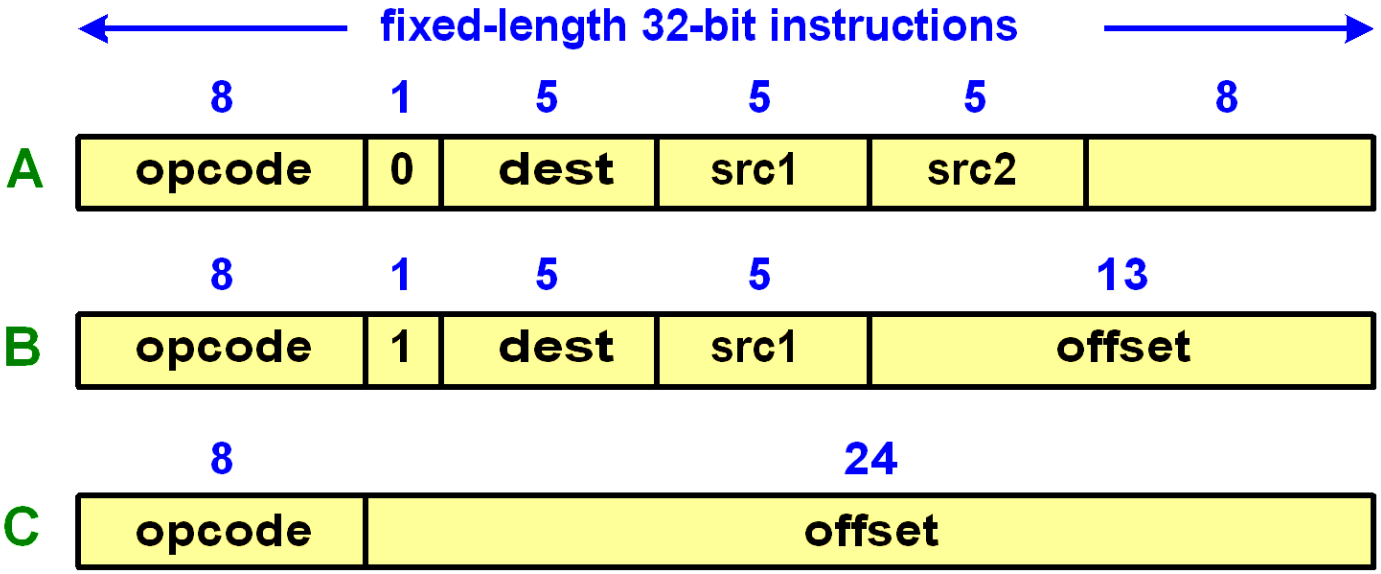
Needs to include room for a register number. The number of registers determine the size required to hold the register number. For example, a system with 16 registers requires a 4-bit register number (because 2^4 = 16).

Indexed and Based+Indexed

Needs to include room for one (Indexed) or two (Based+Indexed) register numbers, along with an offset value. The size of the register number fields depends on the number of registers that the CPU has (as described above), while the size of the offset field determines the maximum offset that can be used. For example, a 10-bit offset field means that offsets could range from -512 to +511 (a total range of 2^{10} = 1024).

Addressing Modes and Instruction Formats

Here’s an example of an ISA design that has three instruction formats that support different addressing modes:



fixed-length 32-bit instructions

A: opcode (8) | 0 (1) | dest (5) | src1 (5) | src2 (5) | (8)

B: opcode (8) | 1 (1) | dest (5) | src1 (5) | offset (13)

C: opcode (8) | offset (24)

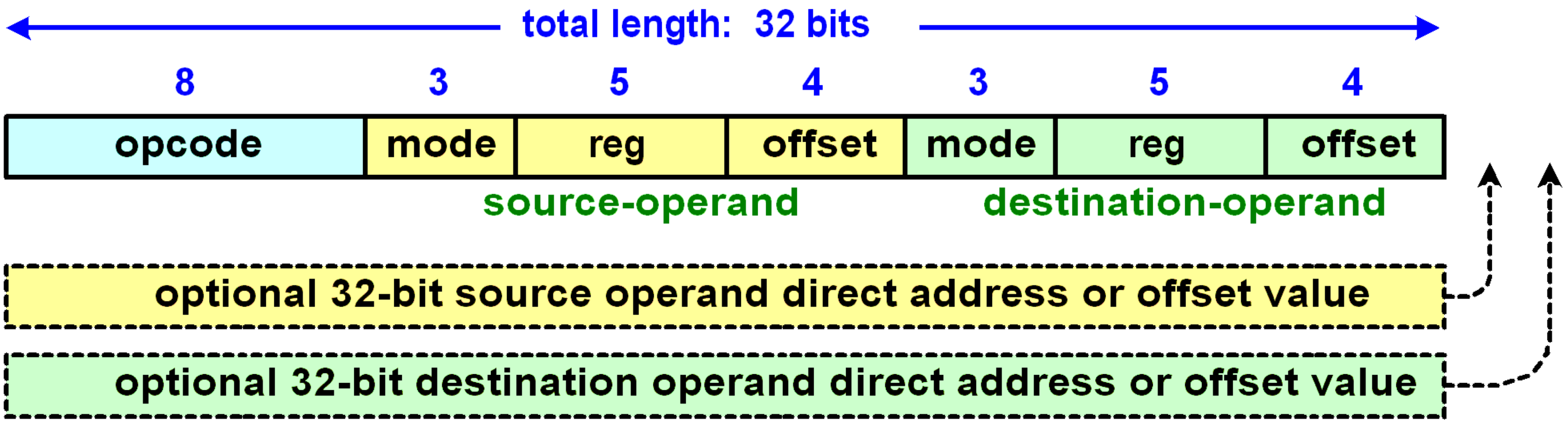
Format “A” provides a 3-operand format that can accommodate register to register instructions. The unused 8-bit field could be used for additional instructions, for example, if a specific opcode is used for floating-point operations then it could contain the floating-point opcode.

Format “B” provides a 2-operand format that can accommodate all addressing modes with a 13-bit offset or immediate value.

Format “C” can be used for instructions that require a single longer offset, such as a CALL or direct memory address for moves to or from memory.

Addressing Modes and Instruction Formats

Here’s a different example of an ISA design that has supports two operands and any addressing format for either operand:



total length: 32 bits

8	3	5	4	3	5	4
opcode	mode	reg	offset	mode	reg	offset

source-operand

destination-operand

optional 32-bit source operand direct address or offset value

optional 32-bit destination operand direct address or offset value

This is the instruction format that was used in the highly successful Digital Equipment PDP-11 and VAX minicomputers. The “mode” field in the source and destination operands chooses one of 8 addressing modes for it’s respective operand. Immediate or indexed offsets can be included in the basic instruction format if the values are small enough. For larger values, one or two optional words can be appended to the instruction to contain a full 32-bits.

This design allows any instruction to make two memory references, a practice that has fallen out of favour in modern systems because of the slow speed of memory compared to the CPU. But the ISA design is very clean – any operand can use any addressing mode. This makes it easy for a compiler to generate ISA instructions and is referred to as “orthogonal addressing”.

Addressing Mode Summary

The following table summarizes the addressing modes supported by the sample architectures:

Mode	Pentium 4	UltraSPARC III	8051
Accumulator			✓
Immediate	✓	✓	✓
Direct	✓		✓
Register	✓	✓	✓
Register Indirect	✓		✓
Indexed	✓	✓	
Based-Indexed	✓	✓	

Points to note include:

- The number of addressing modes isn't usually a factor in the effectiveness of an ISA since most code is now generated by compilers.
- A clean ISA has a few addressing modes that are universally used and can specify all of the registers, including special-purpose registers such as SP and PC
- Immediate, direct, register, and indexed addressing modes are enough for most uses.

Instruction Types

There are a number of general types of instructions that can be executed by most typical Instruction Set Architectures:

Data Movement Instructions

Instructions which move data between registers and/or memory

Dyadic Operations

Instructions such as “ADD” which operate on two values

Monadic Operations

Instructions such as “SHIFT” which operate on one value

Comparisons and Conditional Branches

Instructions which allow a program to choose a course of action

Procedure Calls and Returns

Instructions which transfer control to or from a subroutine or procedure

Loop Control

Instructions which control the repeated execution of a series of statements

Input/Output Instructions

Instructions which move data between the CPU and external I/O devices.

Data Movement Instructions

5.5.1

Data movement instructions should really be called "data copy" instructions because they make a copy of data in a new location. For example, the instruction:

```
MOV     R0,  R1       !Move R0 to R1
```

…overwrites the R1 register with the value of R0, and leaves the R0 register unchanged.

There are two general reasons to use a data movement instruction:

- **To make a copy of data (as in the high-level language statement "A = B;")**
- **To stage data for use with another instruction.**

The later case is required in an architecture where not all operations are possible on all registers or on data stored in any memory location. Some examples of the need to stage data include:

- **In the Java Virtual Machine, two numbers must be put onto the stack before they can be added together using the IADD instruction. The data movement instruction which does this is "ILOAD".**
- **In the Ultra SPARC, a number in memory moved into a register using the LOAD instruction before it can be used in an operation such as "ADD".**

Data Movement Instructions

In order to execute a data movement instruction, the CPU must know three things:

- **Which memory address or register the data is coming from (the "source")**
- **Which memory address or register the data is going to (the "destination").**
- **How many bytes or words are being moved (the "length")**

There are three ways the an instruction can specify the length of the data to be moved:

- **By including the length as an operand in the instruction or in the addressing mode. For example, the addressing mode of the Pentium II can specify a move to or from the 32-bit EAX register, the 16-it AX register, or the 8-bit AL register.**
- **By using unique opcodes for different move lengths. For example, the Motorola 68000 used symbolic opcodes such as "MOVE.L" (move long) or "MOVE.B" (move byte).**
- **By putting a special flag (for example, a byte containing zero) at the end of the data to be moved.**

Dyadic Operations

Dyadic operations include the familiar arithmetic operations such as Add, Subtract, Multiply and Divide. All computers support integer addition and subtraction, and most modern systems also support integer multiplication and subtraction, as well as their equivalent floating point operations. Boolean logical operations such as AND and OR are also dyadic operations.

Dyadic instructions can use 2- or 3-operand format. Examples:

AND R0, R1

This instruction ANDs the values in R0 and R1 and stores the result in R1:

Before:

R0: 0 0 1 0 1 1 0 1

R1: 1 0 1 1 0 1 1 0

After:

R1: 0 0 1 0 0 1 0 0

AND R0, R1, R2

This instruction ANDs the values in R0 and R1 and stores the result in R2:

Before:

R0: 0 0 1 0 1 1 0 1

R1: 1 0 1 1 0 1 1 0

After:

R2: 0 0 1 0 0 1 0 0

Having three operands for dyadic operations requires more space in the instruction to identify the operands, but it’s also more flexible and reduces the need for separate data movement instructions. Most modern RISC architectures use 3-operand instructions.

Monadic Operations

Monadic operations have only one input and one output value. Many Instruction Set Architectures have monadic versions of common dyadic operations. Here are some examples:

CLEAR (CLR) – Set a word to zero

Setting a word to zero can be done using a "MOV" instruction with an immediate "0" value as the source operation. But many ISAs have a one-operand "CLEAR" instruction to do the same thing.

INCREMENT (INC) – Add one to a value

Adding one to a value can be done using an "ADD" instruction with an immediate "1" value as one of the source operands. But many ISAs have a one-operand instruction to do the same thing.

NEGATE (NEG) – Change the sign of an integer value

Changing the sign of a value can be done by subtracting the value from 0 (for example, 0 – 10 = –10). But many ISAs have a special one-operand instruction to do this. (Remember, signed integers are stored in two's-complement notation, so changing the sign consists of reversing all of the bits and adding one).

Exercise 4 - Monadic Operations

Monadic one-operation instructions such as "Clear", "Increment" and "Negate" can be executed using two-operation instructions such as "Move", "Add" and Subtract". Indicate whether the following advantages and disadvantages of monadic instructions are true or not:

1. **Monadic instructions are shorter, therefore using them makes your program slightly smaller and faster to execute**
2. **Each monadic instruction requires it's own opcode, so an ISA that includes them ends up with fewer opcodes available for other instructions.**

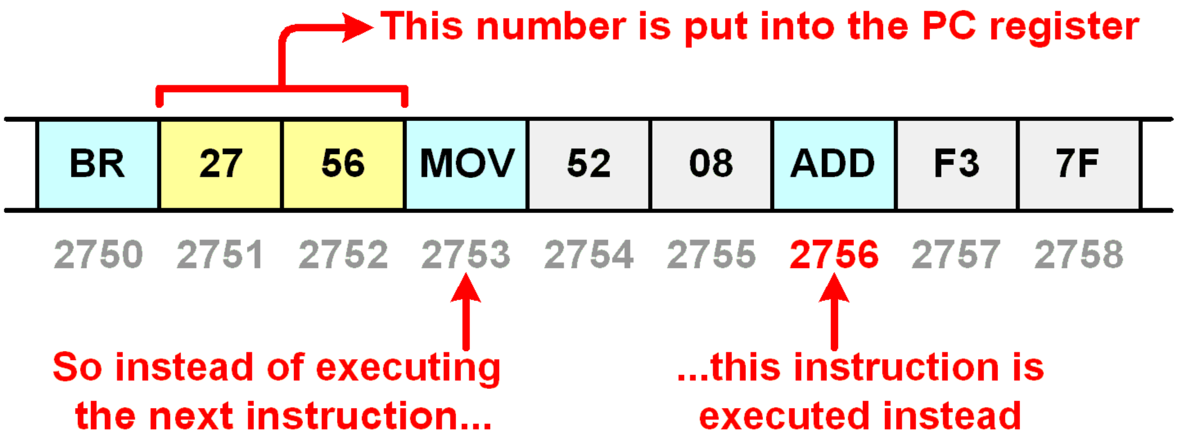
A – TRUE

B – FALSE

Comparisons and Conditional Branches

Conditional Branch instructions are a core feature of every Instruction Set Architecture. They allow the program to make decisions and take different actions depending on the result.

A normal (unconditional) Branch instruction replaces the Program Counter with a new value and thereby causes a different instruction to be executed instead of the one following the Branch.



This number is put into the PC register

BR	27	56	MOV	52	08	ADD	F3	7F
2750	2751	2752	2753	2754	2755	2756	2757	2758

So instead of executing the next instruction...

...this instruction is executed instead

(This example shows absolute branch addressing, but relative addressing can also be used)

Conditional Branches work similarly, but whether the Program Counter is replaced or not depends on the some condition in the machine. Conditions that can be tested include:

- **Whether a number is zero**
- **Whether a number greater than zero**
- **Whether a number is less than zero**
- **Whether the last operation produced a signed overflow**
- **Whether the last operation produced an unsigned overflow**

Comparisons and Conditional Branches

In general there are two forms of conditional branch instructions:

- Instructions which test a condition and perform the branch as part of the same ISA instruction. For example:

```
BZ     R0, A10       ! Branch to label A10 if register R0 = 0
```

- Instructions which test a bit in the flags register and branch or not depending on it's value. These instructions rely on the flags register having been set by some previous instruction. For example:

```
ADD    R0, R1, R2    ! Add R0 and R1 and store result in R2
JZ     A10           ! Jump to label A10 if result was zero
```

In this example, the ADD instruction sets the "zero" flag in the flags register to TRUE if the result of the addition was zero. The "JZ" (Jump if Zero) instruction jumps if the "zero" flag is TRUE.

This method is more common because it's more flexible – you can use conditional branch instructions along with arithmetic, Boolean, shift, or any other instruction that sets the flags.

Note – some ISAs use "BZ" (Branch if Zero) as a mnemonic name and some use "JZ" (Jump if Zero"). The same applies to "JNZ" (Jump Not Zero), "BGE" (Branch Greater or Equal), and so on. Whether the mnemonic names use "Bxx" or "Jxx" has nothing to do with whether the instructions use the flags register or not.

Comparisons and Conditional Branches

If you are writing a program for an ISA whose conditional branch instructions depend on the flags in the status register, it's important to understand which instructions affect those flags. For example, in most ISAs the "MOV" instruction does not change any flags. This means that in the following instruction sequence:

```
ADD     R0, R1, R2      ! Add R0 and R1 and store the result in R2
MOV     R0, NETPAY      ! Store value in memory variable
JZ      A10             ! Jump if gross pay is zero
```

The "JZ" instruction is testing whether or not the result of the "ADD" instruction was zero, and not whether the "MOV" instruction was zero.

Many ISAs contain a special "Compare" instruction (usually with the mnemonic name "CMP") whose only purpose is to compare two numbers and set the flags. Here's an example of it's use:

```
CMP     AGE, #65        ! Is age greater than 65?
JG      A10             ! Jump if gross pay is zero
```

The "Compare" instruction actually subtracts one value from another, but it doesn't save the result anywhere – it's only purpose is to set the flags for a subsequent conditional branch instruction to test.

Exercise 5 - Conditional Branch

Indicate whether the following statements are true regarding instructions which test and condition and branch in a single operation, such as:

```
BZ      R0, A10       ! Branch to label A10 if register R0 = 0
```

1. **Combined test & branch statements require less program memory and execute more quickly.**

2. **Different opcodes are required for “forward” jumps in a program versus “backward” jumps.**

A – TRUE

B – FALSE

Comparisons and Conditional Branches

When comparing numbers, it's important to keep in mind the difference between signed and unsigned numbers. For example, consider this number:

1 0 1 1 0 0 1 0 (hex B3)

If this is being used as an unsigned number, it has a value of decimal 178.

If it's being used as a signed number, then it has a value of – 5 .

If you compare this number to zero:

```
MOV     #0xB3,  R0   ! Put binary "10110010" into R0
CMP     R0, 0        ! Compare it to zero
```

Is the number greater than zero, or less than zero?

We need to have two different ways to compare then number, one if we're treating it as "signed" and one if we're treating it as "unsigned".

Comparisons and Conditional Branches

We saw that the flags register contains a number different bits that show the status of the last operation:

N (Negative) – set to TRUE when the last result was less than zero

Z (Zero) – set to TRUE when the last result was zero

V (oVerflow) – set to TRUE when the last result had a signed overflow

C (Carry) – set to TRUE when the last result had an unsigned overflow

The key to treating comparisons as signed or unsigned lies in the fact that there are two different flags register bits, "V" and "C", for signed and unsigned overflows. When you compare two numbers, both these bits get set. Once the bits are set, you can use different conditional branch instructions to check the bits and branch based on signed or unsigned values:

Unsigned Branches	Signed Branches
JA – Jump Above	JG – Jump Greater Than
JAE – Jump Above or Equal	JGE – Jump Greater Than or Equal
JB – Jump Below	JL – Jump Less Than
JBE – Jump Below or Equal	JLE – Jump Less Than or Equal

(Note – different ISAs may use slightly different mnemonic names than this)

Comparisons and Conditional Branches

5.5.4

So if we go back to our example number:

1 0 1 1 0 0 1 0 (hex B3)

Then we can make an unsigned comparison as follows:

```
MOV     #0xB3,  R0   ! Put binary “10110010” into R0
CMP     R0, 0        ! Compare it to zero
JA      A10          ! Jump if greater than zero (as an unsigned number)
```

The “JA” instruction does jump, because as an unsigned value this number is greater than zero.

Using the same number, we can make a signed comparison as follows:

```
MOV     #0xB3,  R0   ! Put binary “10110010” into R0
CMP     R0, 0        ! Compare it to zero
JG      A10          ! Jump if greater than zero (as a signed number)
```

The “JG” instruction does not jump, because as an signed value this number is less than zero.

Exercise 6 - Signed Comparisons

Indicate whether the following instructions will branch or not branch:

```
1:    MOV    #-4,R4         !Put a decimal 4 into R4
      MOV    #5,R5          !Put a decimal 5 into R5
      CMP    R4,R5          !Compare R4 to R5
      JA     A10            ! Jump if R4 > R5 (as an unsigned value)
```

```
2:    MOV    #4,R4          !Put a decimal 4 into R4
      MOV    #-5,R5         !Put a decimal -5 into R5
      CMP    R4,R5          !Compare R4 to R5
      JG     A10            ! Jump if R4 > R5 (as a signed value)
```

A – No Branch

B – Branch

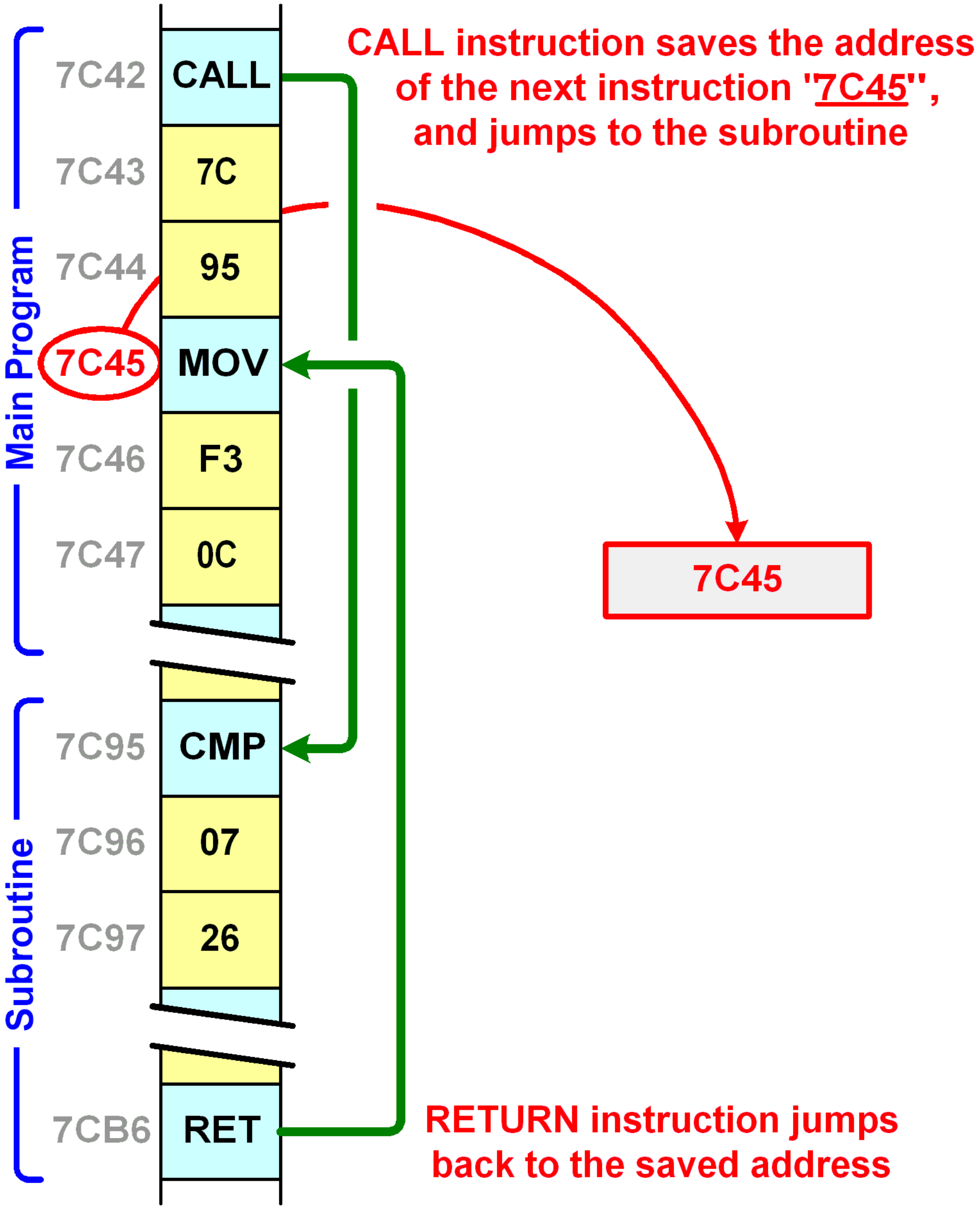
Procedure Calls and Returns

Procedure (subroutine) calls and returns are very similar to unconditional jump instructions – they replace the value in the PC register and cause different sequences of instructions to be executed. (The example shows the CALL statement using absolute addressing, but it can also use relative addressing)

But unlike unconditional jump instructions, the procedure call needs to "save" the address of the instruction that follows it so that the corresponding return instruction can branch back to it.

The address has to be saved because the subroutine can be called from many different places in the program, and the same return instruction has to be able to jump back to the instruction following whichever call it came from.

The big question is: where does the return address get saved?



CALL instruction saves the address of the next instruction '7C45', and jumps to the subroutine

RETURN instruction jumps back to the saved address

Procedure Calls and Returns

The return address could potentially be stored in a number places:

In memory

Storing the return address in memory is problematic because procedures can be nested – one subroutine can call another subroutine down to any number of levels. There needs to be a way to ensure that different memory locations are used for each nested CALL, so that the nested RETURNs can find their way back again. In practice it's a lot of extra work to do this, and this method is no longer used by modern ISAs.

In a register

Storing the return address in a register is also troublesome because you need to find another place to put it as soon as you CALL a nested subroutine. So this is really not much different than storing it in memory.

On the stack

The stack is an ideal place to restore return addresses because it's a Last-In, First-Out (LIFO) list. As each nested subroutine is called, a new return address is pushed onto the top of the stack. As each subroutine returns, it pulls the return address back again from the top of the stack.

The stack allows virtually unlimited nesting of subroutines (limited only by the amount of space available on the stack), and it also permits the use of recursion (which we'll talk about in the next session).

Almost all modern Instruction Set Architectures use the stack to store the return address.

Procedure Calls and Returns

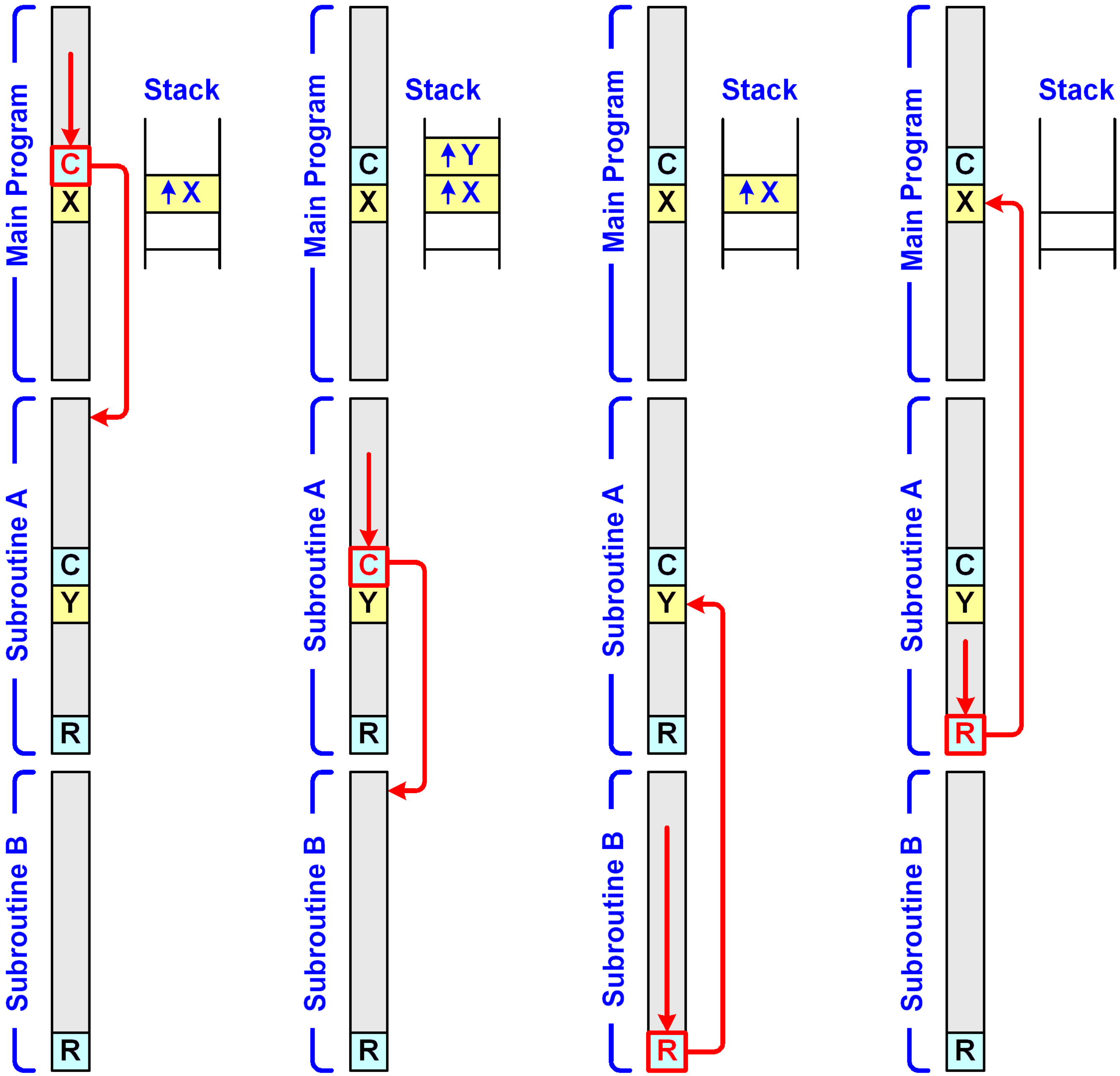
This diagram shows how the return addresses are stored on the stack during nested procedure calls.

1. Main program calls Subroutine A and stores the address of X on the stack.

2. Subroutine A calls Subroutine B and stores the address of Y on the stack.

3. Subroutine B removes the address of Y from the stack and returns to it.

4. Subroutine A removes the address of X from the stack and returns to it.



Loop Control

Most Instruction Set Architectures don’t have specific instructions to control loops – they simply use the same conditional branch instructions that are used for other purposes.   But the instructions have to be arranged so that they produce the correct results.

Consider the following high-level language statement:

```
for ( i = 0;   i < 10;   i++)   { statements }
```

This loop will cause “statements” to be executed 10 times.   The first time they are executed the “i” variable will contain “0”, then next time “i” will contain “1”, and so on until the final execution will be done with “i” containing “9”.

The compiler for a typical ISA would translate these statements into something like this:

```
            MOV     #0,  R0       ! Put loop count into R0  (use R0 to hold “i”)
    A10:
              :
            “statements” go here
              :
            ADD     #1, R0        ! Increment “i” by 1
            CMP     R0, #10       ! Is “i” less than 10?
            JL      A10           !   Yes – go back to top of the loop
```

This method of loop control is known as “test-at-the-bottom” type looping.

Loop Control

Based on what we've seen, you might think that the following language statement…

```
for ( i = 0;   i < n;   i++)   { statements }
```

…might be translated into something like this:

```
        MOV     #0,  R0      ! Put loop count into R0  (use R0 to hold "i")
A10:
          :
        "statements" go here
          :
        ADD     #1, R0       ! Increment "i" by 1
        CMP     R0, n        ! Is "i" less than "n"?
        JL      A10          !   Yes – go back to top of the loop
```

But this doesn't work correctly!

The problem is that we don't know what "n" contains. What happens if "n" contains zero? If so, then "statements" in the body of the loop shouldn't be executed at all. But in the ISA-level code above, the "statements" will always be executed at least once.

Loop Control

Because this high-level language statement…

```
for ( i = 0;   i < n;   i++)   { statements }
```

…may result in "statements" never being executed, we have to use a different method of loop control by putting the test at the top of the loop:

```
          MOV      #0,  R0        ! Put loop count into R0  (use R0 to hold "i")
A10:
          CMP      R0, n          ! Is "i" greater than or equal to "n"?
          JGE      A20            !    Yes – skip to end of loop
            :
          "statements" go here
            :
          ADD      #1, R0         ! Increment "i" by 1
          JMP      A10            !    Yes – go back to top of the loop
A20:
           :
```

This "test-at-the-top" method of loop control is somewhat less efficient because there are two jump statements that are executed for each loop iteration.   So it's best to use "test-at-the-bottom" method whenever you know that the loop will always be executed at least once.

Exercise 7 - Loop Control

What type of loop control would a compiler use when converting the following high level language statements into ISA instructions?

1: `for ( i = 100;   i > 0;   i--)   { statements }`

2: `for ( i = startval;   i < 100;   i++)   { statements }`

A – Test at the top

B – Test at the bottom

Input / Output Instructions

The input / output statements for most Instruction Set Architectures are relatively simple. There are typically one or two variants of an “INPUT” instruction and one or two variants of an “OUTPUT” instruction.   The instructions are typically two-operand instructions that specify an I/O port number and the location where the data is coming from (for OUTPUT instructions) or going to (for INPUT instructions).

Here are a couple of examples:

```
IN       #0x37C,  R0      ! Read data from I/O port 37C (hex) and store in R0

OUT      [R5],  #0x208    ! Send data from memory location whose address
                          !    is in the R5 register to I/O port 208 (hex)
```

Most of the complexity in doing I/O lies in the organization of the I/O devices themselves. Each different port number refers to a particular I/O device, or to some particular function within an I/O device.

It’s very common for each I/O device (a mouse port, for example) to have several I/O ports associated with it, each one controlling a different function or passing different data.

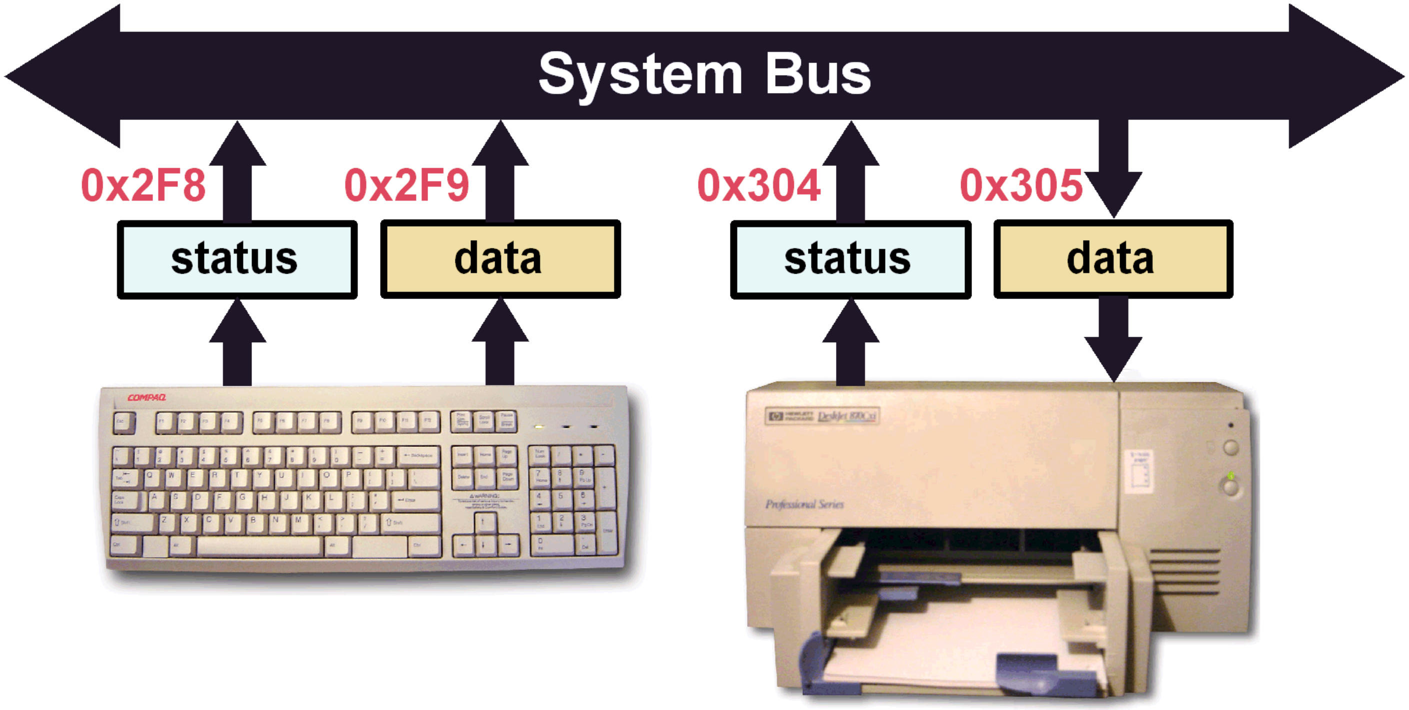
Like the Instruction Set Architecture itself, the types of I/O devices and the numbers and usage of the I/O ports for them are usually very different from one type of system to another, and often even within the same type of system (as an example, consider all the different video cards available for Intel computers).   Because of this, the instructions that directly interact with an I/O device are usually contained a “device driver”, a modular component that can easily be loaded into the operating system when it detects the presence of a certain type of I/O device.

Input / Output Instructions

I/O devices are much slower than the CPU, so when you write or read to or from an I/O device you have to wait for it to be ready to accept or deliver data. There are different techniques used to do this.

To illustrate these techniques, we're going to use an example of a program which reads data from the keyboard and sends it to a printer (in effect, using the computer like a glorified typewriter).

Both the keyboard and the printer have two I/O ports – a "data" port which is used to actually input or output a character, and a "status" port which the program can read to see if the device is ready to accept or deliver another byte of data.



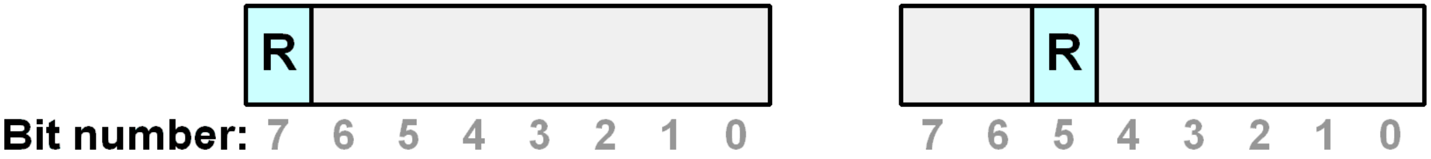
Each port has a unique I/O address. Notice that, while the keyboard data port is an input port, and the printer data port is an output port, both device's status ports are input ports.

Input / Output Instructions

The status ports for the keyboard and printer contain individual bits that indicate whether or not the device is ready for a data transfer. The layout of the status information looks like this:

Keyboard Status Byte (I/O port 0x2F8)

Printer Status Byte (I/O port 0x304)



The "R" bits are "Ready" bits for each device:

Keyboard – the "R" bit means that the user has pressed a key on the keyboard and the character is ready to be read from the data port.

If the program tries to read data from the data port before the "R" bit has been turned on, it will receive garbage.

Printer – the "R" bit means that the printer has finished printing the previous byte and is ready to accept a new byte from the program.

If the program tries to write data to the data port before the "R" bit has been turned on, the data will be lost.

The other bits for each status port are either not used or are used for other functions that are not relevant to this example.

Input / Output Instructions

In order to copy data from the keyboard to the printer, we need to follow the process shown here.

The blue portion waits for a byte of data to become available at the keyboard. It's a loop in which we read the status byte and check it's "Ready" bit to see if the byte is available. If not, we go back and read the status byte again, repeatedly, until the "Ready" bit show that a byte is available to be read.

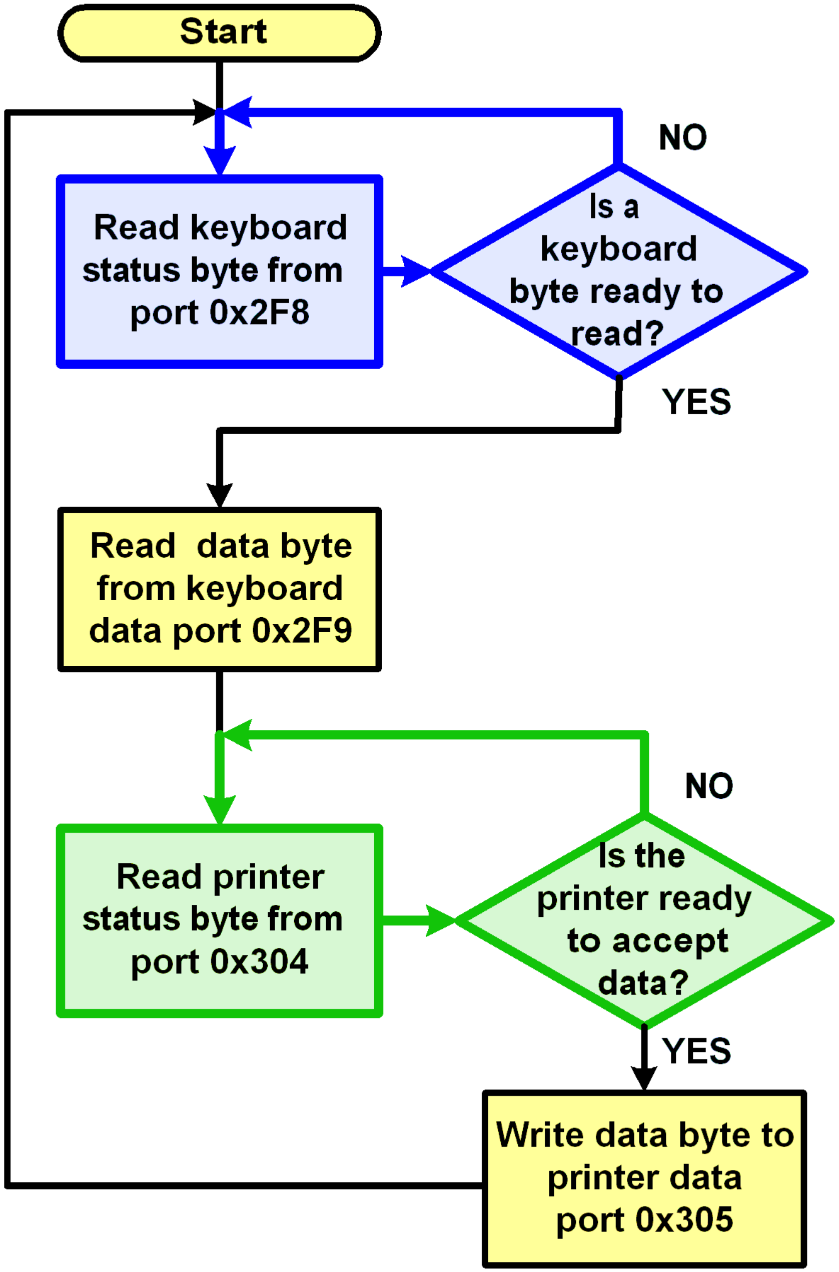
Once a byte is available, we read it.

The green portion waits for the printer to be ready to accept data. Like the keyboard wait loop, this portion repeatedly reads the printer status byte and checks it's "Ready" bit, repeating the process until the "Ready" bit shows the printer is ready to accept data.

Once the printer is ready, we write the byte we read from the keyboard to it.

Then the process is repeated for the next byte.

(Note that this simple process doesn't show any way to exit – once this program is started it will keep running until the power is turned off)


Start
Read keyboard status byte from port 0x2F8
Is a keyboard byte ready to read?
NO
YES
Read data byte from keyboard data port 0x2F9
Read printer status byte from port 0x304
Is the printer ready to accept data?
NO
YES
Write data byte to printer data port 0x305

Input / Output Instructions

Here’s an example of ISA instructions which copy data from the keyboard to the printer:

```
A10:
        IN      #0x2F8,  R0     !  Read keyboard status byte into R0
        TEST    #0x80,  R0      !  Is the “ready” bit (bit 7) on?
        JZ      A10             !     No – go back and read status byte again

        IN      #0x2F9, R1      !  Read keyboard data into R1

A20:
        IN      #0x304,  R0     !  Read printer status byte into R0
        TEST    #0x20,  R0      !  Is the “ready” bit  (bit 5) on?
        JZ      A20             !     No – go back and read status byte again

        OUT     R1, #0x305      !  Write keyboard data in R1 to printer port

        JMP     A10             !  Go back and process the next byte
```

Note that this an endless loop.   If we want to provide a way to exit the loop, we can watch for a “^Z” (end-of-file) byte from the keyboard by adding these instructions before label “A20”:

```
        CMP     R0, #26         !  Is the byte we just read a “^Z” character?
        JE      A90_EXIT        !    Yes – exit the loop
```

Input / Output Instructions

Our sample program to copy data from the keyboard to the printer uses CPU instructions to copy every bytes of data from one I/O device to another. This type of I/O is known as "programmed I/O" because the I/O is directly driven by program instructions.

The sample program also uses loops to repeatedly test the I/O devices' status bytes until they show the devices are ready. This technique is called "busy waiting" in the textbook, but a more common term in the industry is "polling".

Polling is very wasteful of the CPU, because it can't do anything else while it's executing the statements to repeatedly read and check the status byte.

For a simple embedded system such as a VCR or a microwave oven, this isn't a problem because the CPU doesn't have anything else to do anyway. But for a general-purpose multitasking system, we want to allow the CPU to execute instructions from other programs while one program is waiting for an I/O device to be ready.

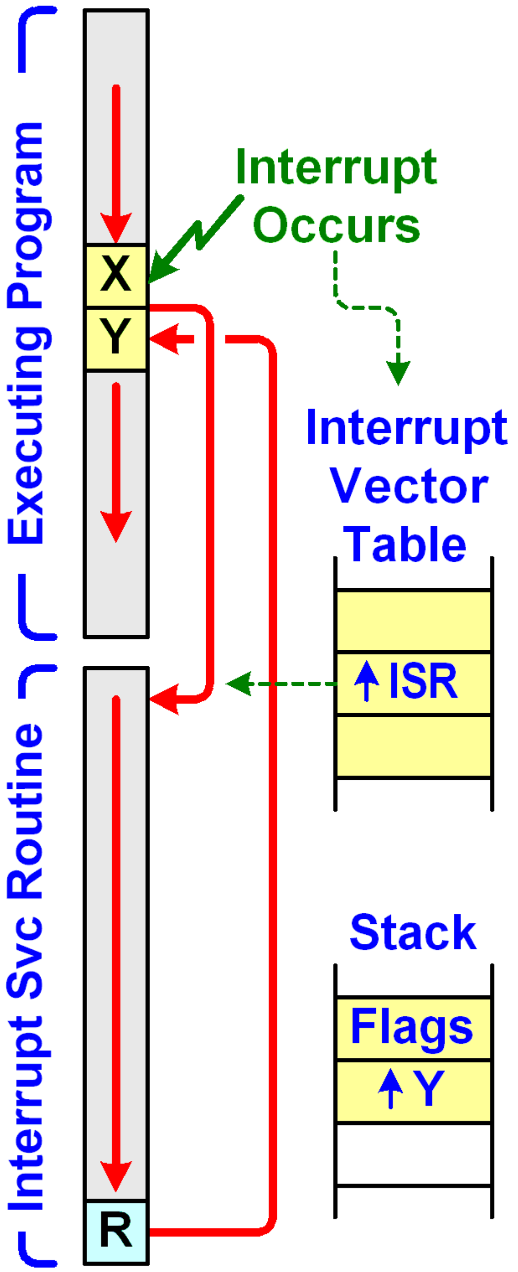
Another way to wait for devices to become ready is to use what's known as "interrupt-driven" I/O. In interrupt-driven I/O we configure the I/O device to interrupt the CPU when it is ready to accept or deliver data. The CPU can be doing other work while it's waiting for the I/O device, and the I/O device will inform the CPU when it's ready.

Input / Output Instructions

What is an interrupt? It's a signal sent by an I/O device to the CPU, and it's also the sequence of events that the CPU performs when it receives such a signal.

The diagram at the right shows what happens when an interrupt signal is received by the CPU:

- An interrupt occurs while the CPU is executing instruction X
- The CPU saves the current state of the executing program on the stack along with the address of the next instruction.
- The CPU determines what type of interrupt has occurred. It looks up the equivalent entry in an Interrupt Vector Table to tell it which Interrupt Service Routine (ISR) it is supposed to call to handle this interrupt.
- The CPU jumps to the first instruction in the Interrupt Service Routine.
- The interrupt service routine executes the instructions requires to handle the interrupt.
- When the interrupt service routine is finished, it executes a special "Return From Interrupt" instruction to restore the original state of the machine from the stack and return to the interrupted program.



Input / Output Instructions

There are many different types of devices that can interrupt the CPU:

- **I/O devices like disk drives or USB controllers**
- **System devices like timers**
- **Error checking circuitry like memory controllers**

The Interrupt Vector table contains an entry for each different type of interrupt. Each entry contains the address of the first instruction in the Interrupt Service Routine that handles that type of interrupt.

When an interrupt occurs, the process is very much like a CALL instruction, but with these differences:

- **The transfer of control is caused by the external interrupt rather than by executing a particular instruction.**
- **The target of the control transfer is given by the Interrupt Vector Table entry rather than by the instruction itself.**
- **Additional state information is saved on the stack so that it's possible to return to the interrupted program with absolutely no change in it's execution context (this is referred to as "transparency").**
- **A special "Return from Interrupt" instruction is used to return to the interrupted program and restore the saved state information.**

The operating system loads device drivers which contain the Interrupt Service Routine code, and it fills in the Interrupt Vector Table with the correct addresses for each driver.

Input / Output Instructions

With interrupt-driven I/O, there’s no need to have a loop which waits for the I/O device to become ready. Instead, we simply need to create a pair of Interrupt Service Routines to perform the required I/O statements. The interrupt service routines for the keyboard and the printer might look something like this:

```
Keyboard-ISR:
    IN      #0x2F9, R0       !  Read keyboard data into R0
    MOV     R0, KB-Buffer    !  Save the keystroke data in a memory variable
    IRET                     !  Return to interrupted program

Printer-ISR:
    MOV     KB-Buffer-R0     !  Move saved keyboard data from memory to R0
    OUT     R0, #0x305       !  Write data to printer.
    IRET                     !  Return to interrupted program
```

There’s no need to test the “ready” bit for the I/O devices because the statements in the interrupt service routines are only ever called when the I/O devices are ready for the I/O operation. And the CPU can be doing other work while it’s waiting for the I/O devices.

Note **– this is an overly simplified example, because:**

- **It uses the R0 register without saving it’s previous value. This will cause problems if the interrupted program was using the R0 register for something.**
- **It doesn’t take into account the fact that a new keyboard byte may not have been read yet when the printer ISR is called. To work properly, some more status information would have to be stored in memory and acted upon by the both routines.**

Input / Output Instructions

So far, our examples have used "Programmed I/O" to perform our input and output. This isn't too much of a problem for slow-speed devices like keyboards, but for high speed devices like disk drives it uses up a lot of CPU power because there are several instructions required to process each byte and there are potentially millions of bytes per second being executed.

An alternative to "Programmed I/O" is called "Direct Memory Access", or "DMA". DMA uses a special I/O controller device in the system to move data between memory and an I/O device without the CPU having to execute instructions for each byte.

The DMA device has several I/O ports which the CPU can write out control information to. The control information includes:

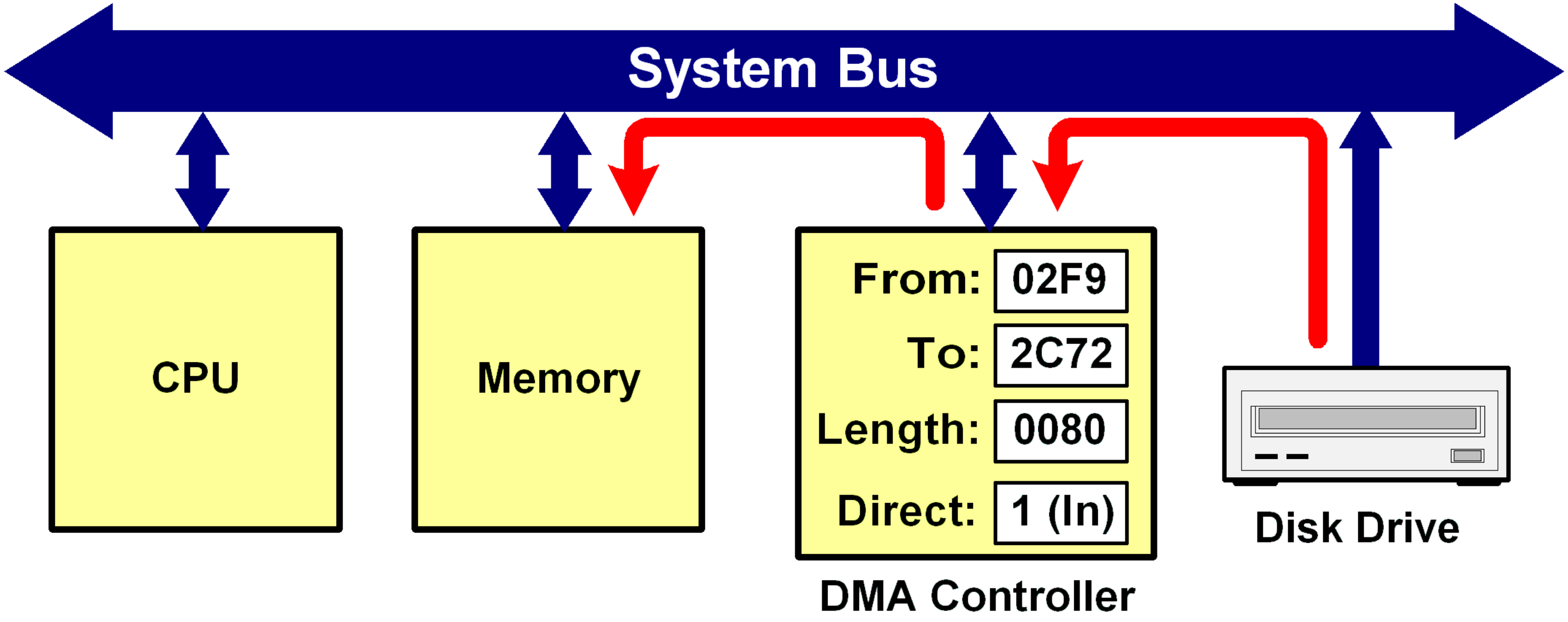
- **Which I/O port the data is being transferred to or from.**
- **Which memory address the data is being transferred to or from**
- **How many bytes are being transferred.**
- **Which direction the data is going in (memory → I/O device or vice versa)**

Once the information has been loaded into the DMA controller, it will request access to the system bus (using the bus's "Bus Arbitration" circuitry as we discussed some time back) and then request bus transfers between itself and the I/O device and memory.

This process is shown in the diagram on the next page.

Input / Output Instructions

Once it's been programmed with the control information, the DMA controller performs it's own transfers between the I/O device and memory:

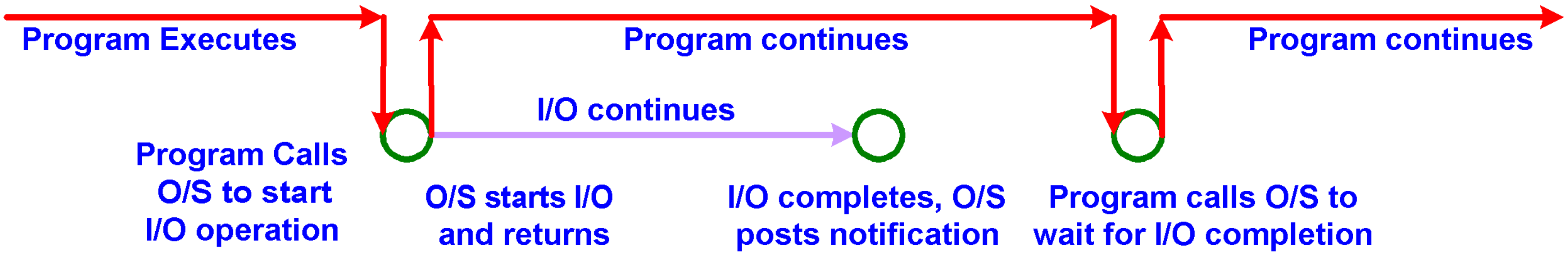


In order to transfer the information, the DMA Controller requests ownership of the bus using the bus's arbitration circuitry. This effectively "steals" bus cycles from the CPU, and is often known as "cycle stealing". But the impact of the bus cycles stolen by the DMA circuit is much smaller than the CPU having to execute the I/O instructions to transfer the data itself.

High-Level Programming and Asynchronous I/O

Normally I/O in high level programs is done synchronously. The program issues an I/O statement or call, which pauses the program until the I/O is completed, then execution resumes at the next statement.

Operating systems such as Windows also support asynchronous I/O. The program calls the appropriate O/S routine to request an I/O operation, and the routine returns back again without waiting for the I/O to complete. The program can then do other things at the same time that the I/O is progressing. When the program needs to confirm that the I/O is complete it can call an O/S routine to query the status of the I/O or to wait for it’s completion.


Program Executes
Program continues
Program continues
I/O continues
Program Calls O/S to start I/O operation
O/S starts I/O and returns
I/O completes, O/S posts notification
Program calls O/S to wait for I/O completion

Asynchronous I/O can speed up programs that are able to do work while waiting for I/O operations to complete. For example, a program copying data from one disk to another can write the first set of data at the same time it reads the second set, and continue to read and write subsequent sets of data at the same time. Asynchronous I/O in this case could cut the execution time in half.

Exercise 8 – I/O

What type of I/O is being described in each of these cases:

1. **A program processes one set of data at the same time that the operating system is reading the next set of data from disk.**
2. **The firmware in a cell phone is constantly checking to see if any keys are being pressed**
3. **The program executes an “output” instruction for each byte of data sent to a printer**

A – Polling

B – Programmed I/O

C – Interrupt-Driven I/O

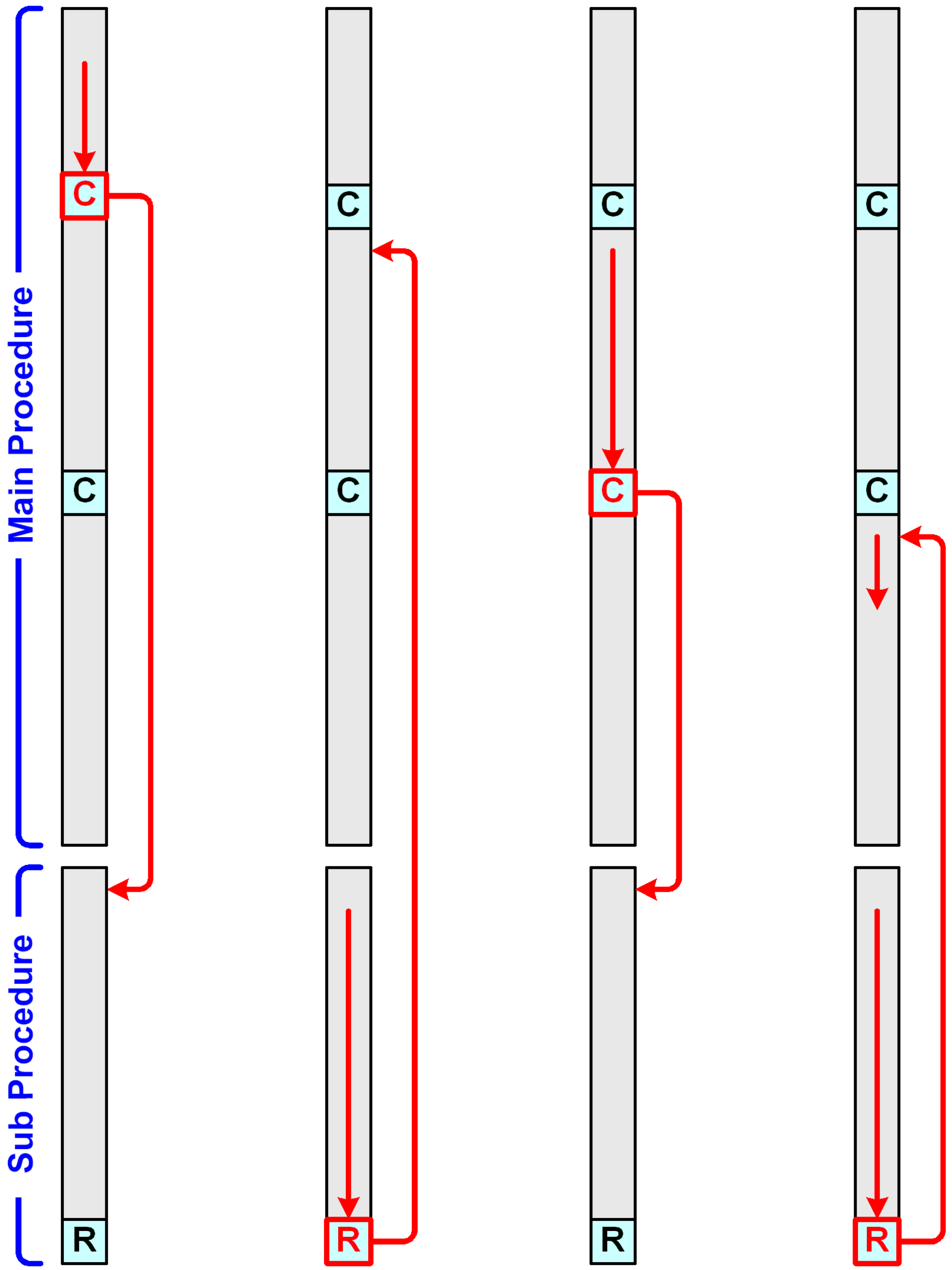
D – Asynchronous I/O

Procedures

The highest level of structure in a program is the procedure. A procedure is a collection of instructions that performs some specific function and which can be executed from any point in the overall program.

The diagram at right shows how a procedure can be called from different points in the program. The procedure executes the same instructions, but returns to a different place depending on where it was called from.

From the point of view of the main program, the procedure "CALL" statement performs all of the functions done by the procedure, as if the "CALL" statement itself did the work. The fact that a procedure is called, and the exact workings of the procedure, is irrelevant to the main program.



Procedures

Parameter Passing

5.6.2

When one procedure calls another procedure, it usually passes some data which the called procedure will process. The passed data is usually called "parameters" for the called procedure. For example, in the high-level language statement:

```
CALL    CLEAR(  Table, 5 );
```

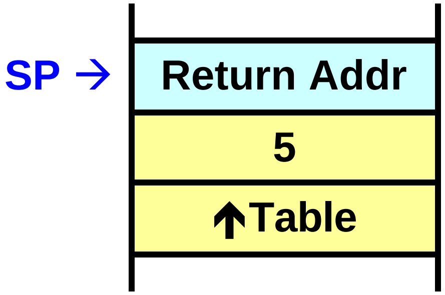
The variable Table and the constant value 5 are parameters that the CLEAR routine will use.

At the ISA level, parameters are usually passed to a called routine by pushing either their value or their address onto the stack. So the above high-level statement might be translated into ISA-level instructions such as:

```
PUSH    #Table        ! Push address of "Table" onto the stack
PUSH    #5            ! Push the constant "5" onto the stack
CALL    CLEAR         ! Call the "CLEAR" routine
```

When control reaches the first statement of the "CLEAR" routine, the stack will have the values shown on the right:

- The two parameters that were pushed onto the stack by the calling routine
- The return address that is pushed onto the stack by the CALL instruction.



Procedures

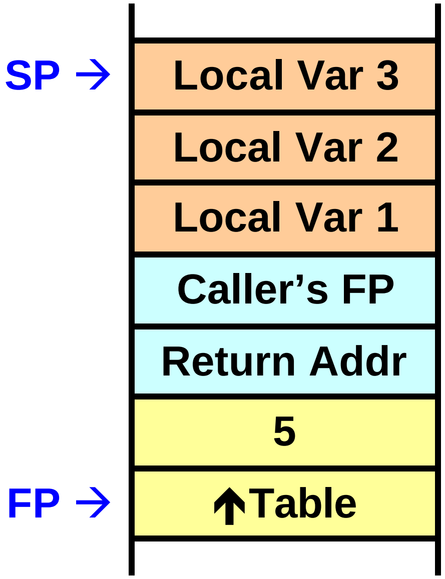
The Stack Frame

You may recall that the local variables for a procedure are also stored on the stack. When the called routine starts executing, the first thing it has to do is to create a “stack frame” to hold the local variables. There are two things that need to be done to create the stack frame:

1. The stack pointer register needs to be incremented to leave room for the procedure’s local variables (and those variables need to be initialized if necessary)
2. The “Frame Pointer” register needs to be changed to points to the parameters and local variables to give the procedure an easy way to find it’s data. But before changing the FP register, the routine must save it on the stack so that it can be restored before returning to the program that called it.

Once the stack frame has been created, the stack will look something like what’s shown at the right. In addition to the original parameters and return address, the following changes have been made:

- The FP register that was being used by the caller before the CALL instruction has been saved on the stack.
- The SP register has been incremented to allow space for the routine’s local variables (3 variables in this example).



SP →

Local Var 3

Local Var 2

Local Var 1

Caller’s FP

Return Addr

5

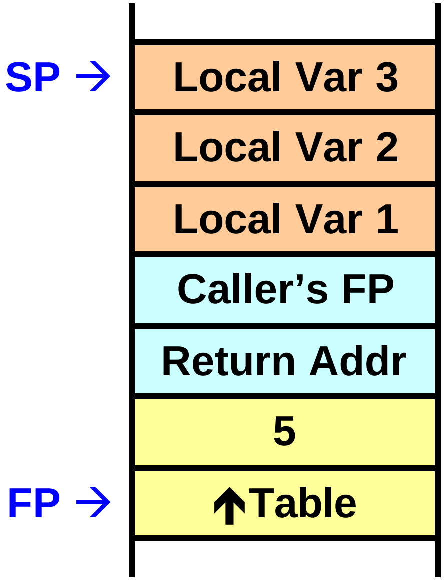
FP →

↑Table

Procedures

The Stack Frame

When the called procedure finishes and is about to return to the calling program, it has to get rid of the stack frame and return everything to the state it was in when the CALL instruction was issued. This means that it must:



- Subtract the number of local variables from the stack pointer register so that it points back to the saved FP value on the stack.
- POP the saved FP value back into the FP register so that it has the same value it did before the CALL instruction.
- Use a RETURN instruction to pop the return address off the stack and into the PC register. This will cause the program to resume execution at the instruction following the CALL.

The parameters are still on the stack. In some ISA's the calling procedure has to remove them from the stack, and in others a special version of the RETURN instruction can be used which removes them from the stack at the same time that the PC is reset to the instruction following the CALL.

The instructions to manage the stack at the entry and exit points of the called procedure are called the prologue and epilogue code. Since they're executed at the beginning and end of every call, it's important that they be as efficient as possible. Many ISAs have special instructions which help to do this work.

Procedures

Nested Stack Frames

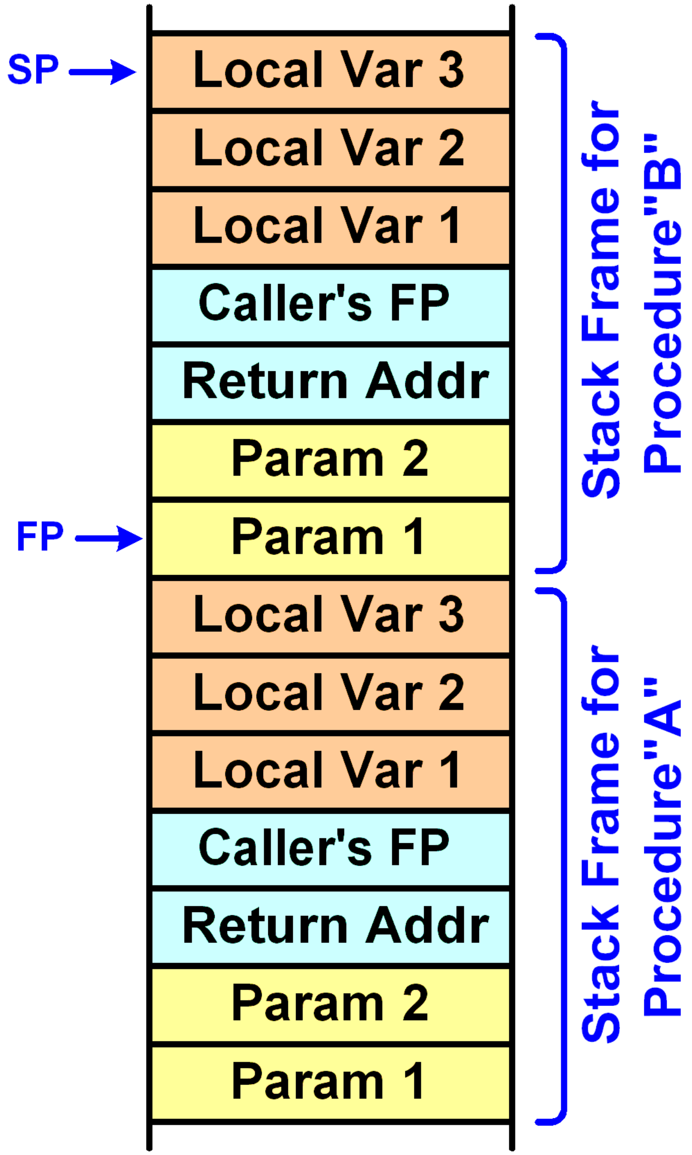
When one procedure calls another procedure, a new stack frame is created. While the called procedure executes, two frames exist on the stack – one which holds the variables and parameters for the original calling procedure, and one which holds the parameters and variables for the called procedure.

The diagram at right shows what the stack holds when a main procedure, “Procedure A”, calls a sub procedure, “Procedure B”.

As long as each procedure is active, the stack contains space for the procedure’s local variables and parameters.

When the procedure exits, the stack space is released and can be reused for the parameters and variables required for a different procedure.

This is why a procedure’s local variables are said to exist only for the life of the procedure itself.



Procedures

Recursive Procedure Calls

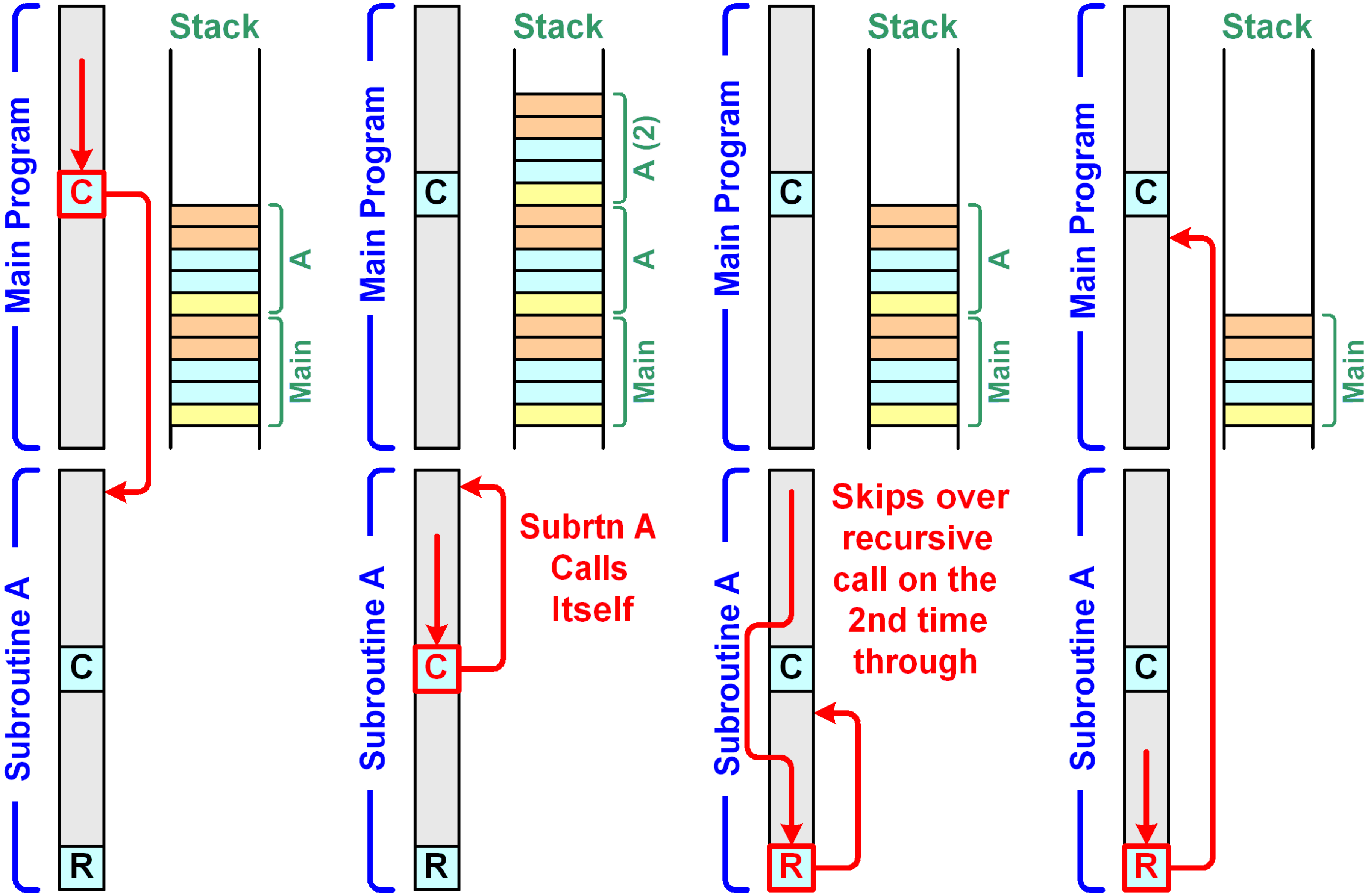
It is possible for a procedure to contain a CALL statement that calls itself – this is known as a Recursive Procedure. Recursive procedures are handy for solving certain types of problems (although there is always a way to solve them non-recursively as well). Here are some points about recursive procedures:

- Each time the procedure calls itself, another stack frame is created to hold a new set of parameters and local variables. So different call levels of the procedure can have different values for it's parameters and variables.
- Like a loop, a recursive procedures must have some conditional test (i.e., an IF statement) that controls whether it continues to call itself or not. If there is no such test, the procedure will call itself over an over again like an endless loop.
- Because a new stack frame is allocated for each call, the number of calls is limited by the amount to stack space available. If calls are nested too deeply the stack will overflow and cause the program to crash. In most cases there is enough room on the stack to hold tens of thousands of nested calls, so this is rarely a problem (unless the "exit test" doesn't work and the procedure calls itself endlessly). But you should be careful about creating recursive procedures that require huge amounts of local variable space (large arrays, for example).

The next page show the flow of control through a recursive procedure and how the stack frames are allocated and released on the calls and exits.

Procedures

Recursive Procedure Calls



Subrtn A Calls Itself

Skips over recursive call on the 2nd time through

Procedures

Recursive Procedure Calls

If you have difficulty understanding a recursive procedure, you can simply think of it as a second procedure that you wrote that is identical to the first one. Here's a simple example of a recursive call:

```
main ()                     recursive ( n ) )   [n = 2]     recursive ( n ) )   [n = 1]
{                           {                               {
        recursive( 2 );             print  n;                       print  n;
}                                   if  (n > 1)                     if  (n > 1)
                                            recursive ( n-1);               recursive ( n-1);
                            }                               }
```

In this example, we have a "main" procedure and recursive procedure called "recursive". There's actually only one "recursive", but we've shown two identical copies of it to clarify how control flows through it:

- **"main" calls the middle "recursive" with a parameter of 2.**
- **the middle "recursive" prints the parameter, so a "2" appears at the output.**
- **the parameter is > 1, so the middle "recursive" calls itself with a parameter of 1.**
- **the right "recursive" prints the parameter, so a "1" appears at the output.**
- **the parameter is not > 1, so the right "recursive" simply returns to the middle "recursive"**
- **the middle "recursive" returns to the "main"**
- **"main" exits.**

The result is that the program prints "2 1" as output.

Procedures

Recursive Procedure Calls

Remember that there's really just one copy of a "recursive" procedure's instructions, but multiple copies of it's data.   Let's see what happens if we put the print statement after the exit test:

```
main ()
{
        recursive( 2 );
}
```

```
recursive ( n )    [n = 2]
{
        if  (n > 1)
                recursive ( n-1);
        print  n;
}
```

```
recursive ( n )    [n = 1]
{
        if  (n > 1)
                recursive ( n-1);
        print  n;
}
```

- **"main" calls the middle "recursive" with a parameter of 2.**
- **the parameter is > 1, so the middle "recursive" calls itself with a parameter of 1.**
- **the parameter is not > 1, so the right "recursive" does not call itself again**
- **the right "recursive" prints it's parameter with a value of 1**
- **the right "recursive" returns to the middle recursive**
- **the middle "recursive" prints it's parameter with a value of 2**
- **the middle "recursive" returns to the "main" procedure**
- **the "main" procedure exits**

The result is that the program prints "1 2" as output, the opposite of the first example.

Exercise 9 - Recursive Procedure

What is wrong with the following recursive procedure:

```
recursive ( n )
{
        print  n;
        if  (n > 1)  recursive ( n-1);
                else recursive ( n+1);
}
```

A – **There's no way for the procedure to stop calling itself**

B – **There's no check for the case where n = 0**

C – **The print statement should follow the conditional test**

D – **There are two recursive calls in the same procedure**

Traps and Exceptions

A trap (known on some systems as an “exception”) is similar to an interrupt, but it is caused directly by an instruction in the program that is executing rather than as a result of receiving a signal from an external device. Traps are events that are synchronous to program execution, while interrupts are asynchronous.

Like an interrupt, a trap essentially performs a hardware procedure call by using the type of exception to look up a “service routine” address in a table.

Traps are caused by important program events but generally uncommon that need to be handled somehow. Examples of program events that can cause traps include:

- Division by zero and sometimes other arithmetic errors such as overflow
- Memory access violation (i.e., attempting to write to read-only memory)
- Attempt to execute an illegal opcode
- Stack overflow
- Attempt to access an unaligned memory address (in ISAs that prohibit this)
- Attempt to access a virtual memory address that is not resident in main memory

Avoiding Traps and Exceptions

It's important to understand the rules that your language uses when it evaluates expressions. Consider the following statement:

```
if  (   (x != 0)  &&  (y/x > 500)   )  {  …do something… }
```

In this statement, it's very important that the (x != 0) part be executed and checked first, and if it's false that the (y/x > 500) part not be executed.   If (y/x > 500) is executed first, or if it's executed even when (x != 0) is false, then your program could crash with a divide-by-zero exception.

If the language you're using could potentially cause this problem, then you need to make the evaluation order explicit, as in:

```
if  (x != 0)
{
      if (y/x > 500)  {  …do something… }
}
```

Traps and Exceptions

A trap is <u>not necessarily</u> an indication of an error in the program. Here are a couple of examples of traps that could occur as a normal part of program execution:

Page Fault

Operating systems that use virtual memory keep some parts of the program on disk rather than loading the entire program into memory. The O/S sets flags in the memory management hardware to indicate which pages are on disk.

If the program attempts to access a memory address that is on disk, it triggers a “page fault” trap. This causes the O/S virtual memory manager to run, load the missing page into memory, reset the memory management hardware to indicate the page is in memory, then return to the program to re-execute the same instruction that caused the trap.

As a result, the program continues executing with the data now loaded into memory. This process is completely transparent to the program (except for performance).

Illegal Opcode

A “family” of computers with high-end and low-end models may not include all of the ISA instructions in the low-end models that are implemented on the high-end models. If a program tries to execute a missing instruction on a low-end model, it causes an “illegal opcode” trap.

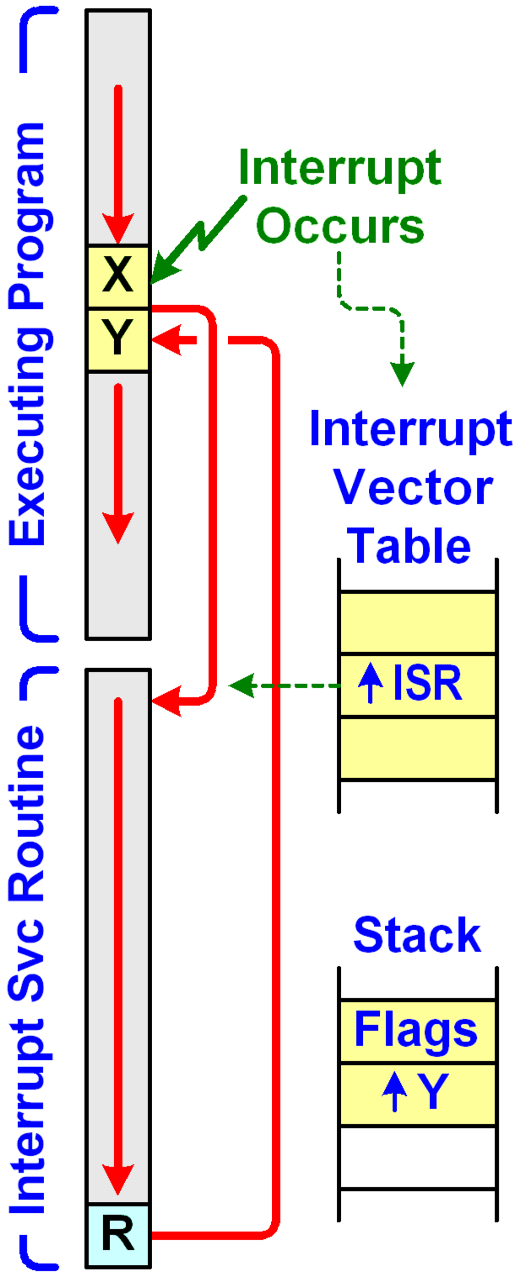
This trap can be passed to a software interpreter which then performs the same work as the high-end hardware would have. So programs can run properly on all models of the family as if they all have the same capabilities.

Interrupts

As we discussed earlier, an Interrupt is an event that's initiated by a device external to the CPU. Unlike traps, which are repeatable (if you run the same instruction under the same conditions you'll generate the same trap), interrupts are much harder to reproduce because exactly <u>when</u> the interrupt occurs is <u>asynchronous</u> to the execution of the program (i.e., it doesn't depend on the instructions being executed).

When the interrupt occurs, the CPU identifies which device caused the interrupt and uses that information to determine which Interrupt Service Routine (ISR) will handle the interrupt. (This is also sometimes called an "Interrupt Handler".) The addresses of the ISRs to handle all the different devices are contained in an Interrupt Vector Table located at some special memory location. Once the ISR address is found, the system performs the equivalent of a CALL instruction to branch to the ISR, where the procedure executes instructions to deal with the interrupt.

When the ISR has finished it works, it issues a "return from interrupt" instruction to return back to the program that was executing when the interrupt occurred. That program carries on with the next instruction, blissfully unaware of what just happened.



Interrupts

Nested Interrupts

Many different devices on the system can interrupt the CPU, so it's quite possible that an interrupt can occur while the Interrupt Service Routine for a different device is already executing. In fact, it's even possible for an interrupt to occur while the Interrupt Service Routine for the same device is already executing.

Interrupts occurring while an Interrupt Service Routine is executing are known as "nested interrupts".

Writing the Interrupt Service Routine so that it can start executing again while it's in the middle of a previous iteration of execution can be quite complex. A simple solution to the problem is to disable interrupts while an Interrupt Service Routine is executing.

Most ISAs have a means to disable interrupts from external I/O device, often by setting a particular bit in the status register. When interrupts are disabled, the CPU does not recognize interrupt signals it receives until they status register is reset to re-enable interrupts.

In many ISAs, the action of accepting and interrupt and calling the Interrupt Service Routine automatically disables further interrupts until the "Return From Interrupt" instruction is executed.

Many CPUs have a special "Non-Maskable Interrupt" (NMI) signal for important interrupts that should not be ignored. This signal is usually used for critical problems such as memory errors that cannot be ignored.

It's important to minimize the time that interrupts are disabled to eliminate the possibility of losing data that needs to be delivered quickly.

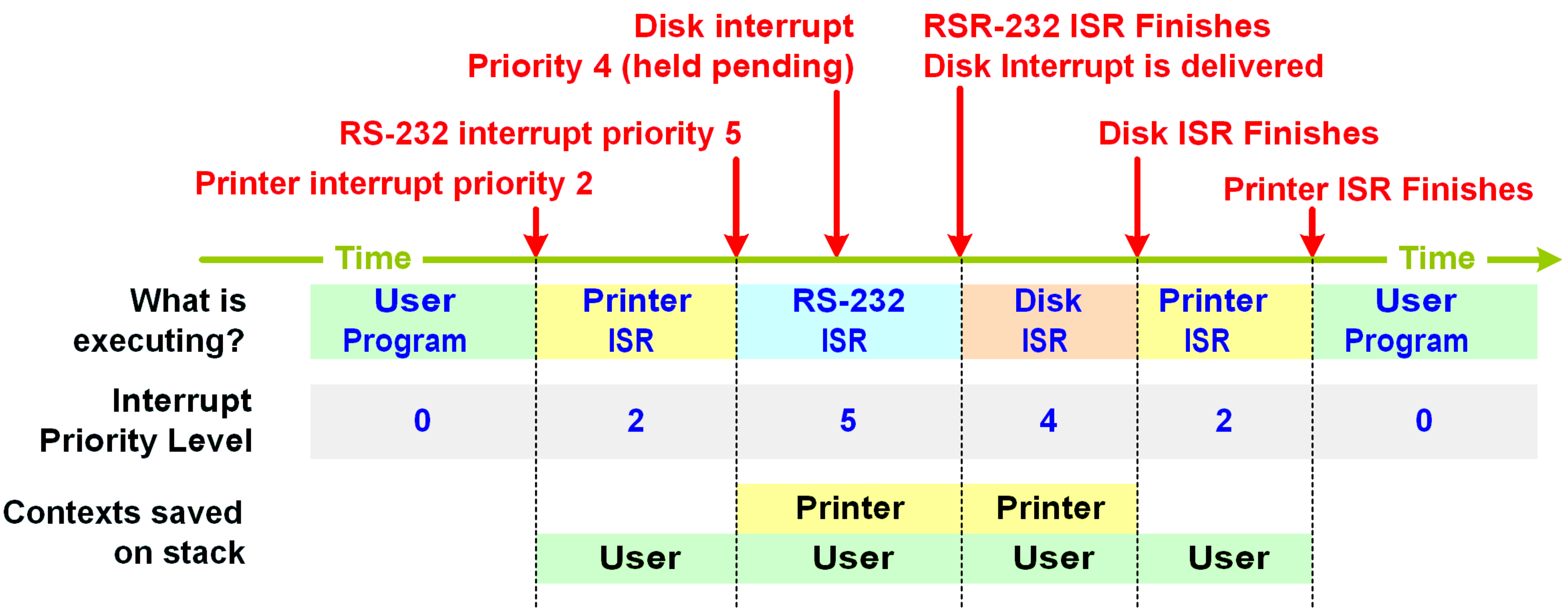
Interrupts

Interrupt Priority Levels

A better solution for dealing with nested interrupts is to have different "Interrupt Priority Levels" (IPLs). Different devices are given different priorities, and a new interrupt from a device at one priority level will only be accepted if the CPU is not executing an interrupt of the same or higher priority level. The current IPL is typically held in the status word.

In a multilevel scheme such as this, the more important interrupt signals and the devices that are more time-sensitive are given a higher priority level than slower devices.

The diagram below shows the processing sequence that occurs in a multi-level interrupt system when several nested interrupts are received:



Printer interrupt priority 2
RS-232 interrupt priority 5
Disk interrupt
Priority 4 (held pending)
RSR-232 ISR Finishes
Disk Interrupt is delivered
Disk ISR Finishes
Printer ISR Finishes
Time
Time

What is executing?	User Program	Printer ISR	RS-232 ISR	Disk ISR	Printer ISR	User Program
Interrupt Priority Level	0	2	5	4	2	0
Contexts saved on stack			Printer	Printer		
		User	User	User	User	

Exercise 10 – Traps and Interrupts

In what way are Traps and Interrupts <u>similar</u>:

A – They are both caused by actions occurring asynchronously to the execution of the program

B – They are both caused by actions which occur synchronously as a result of program execution

C – They both cause a hardware procedure call to a service routine

D – They both indicate an error has occurred

Key Concepts

- Parts of an instruction
- How expanding opcodes work
- Addressing modes
    - ➢ Register
    - ➢ Immediate
    - ➢ Direct
    - ➢ Register Indirect
    - ➢ Indexed and Based Indexed
- Absolute vs. Relative branch addressing
- Loop Control
- Polling vs. Interrupt-driven I/O
- Procedure calls and parameter passing

What's Next

- **Look at Review Questions for this week**

- **Study for the Final exam next week, which includes, primarily:**
  - **Lectures since the mid-term exam**
  - **WebCT modules 5 thru 8**

(Note – marked quizzes and assignments will be handed back before the start of the exam)