MODULE_3

INSTRUCTION SET ARCHITECTURE

general-purpose RISC architectures (MIPS, PowerPC, Precision Architecture, SPARC), four embedded RISC processors (ARM, Hitachi SH, MIPS 16, Thumb), and three older architectures (80x86, IBM 360/370, and VAX). Before we discuss

Classifying Instruction Set Architectures

- The type of **internal storage in a processor** is the most basic differentiation.
- The major choices are a **stack**, an **accumulator**, or a **set of registers**.
- Operands may be named explicitly or implicitly:
- The operands in a *stack architecture* are implicitly on the top of the stack.
- The operands in an *accumulator architecture* one operand is implicitly the accumulator.
- The *general-purpose register architectures* have only explicit operands—either registers or memory locations. The explicit operands may be accessed directly from memory or may need

  to be first loaded into temporary storage, depending

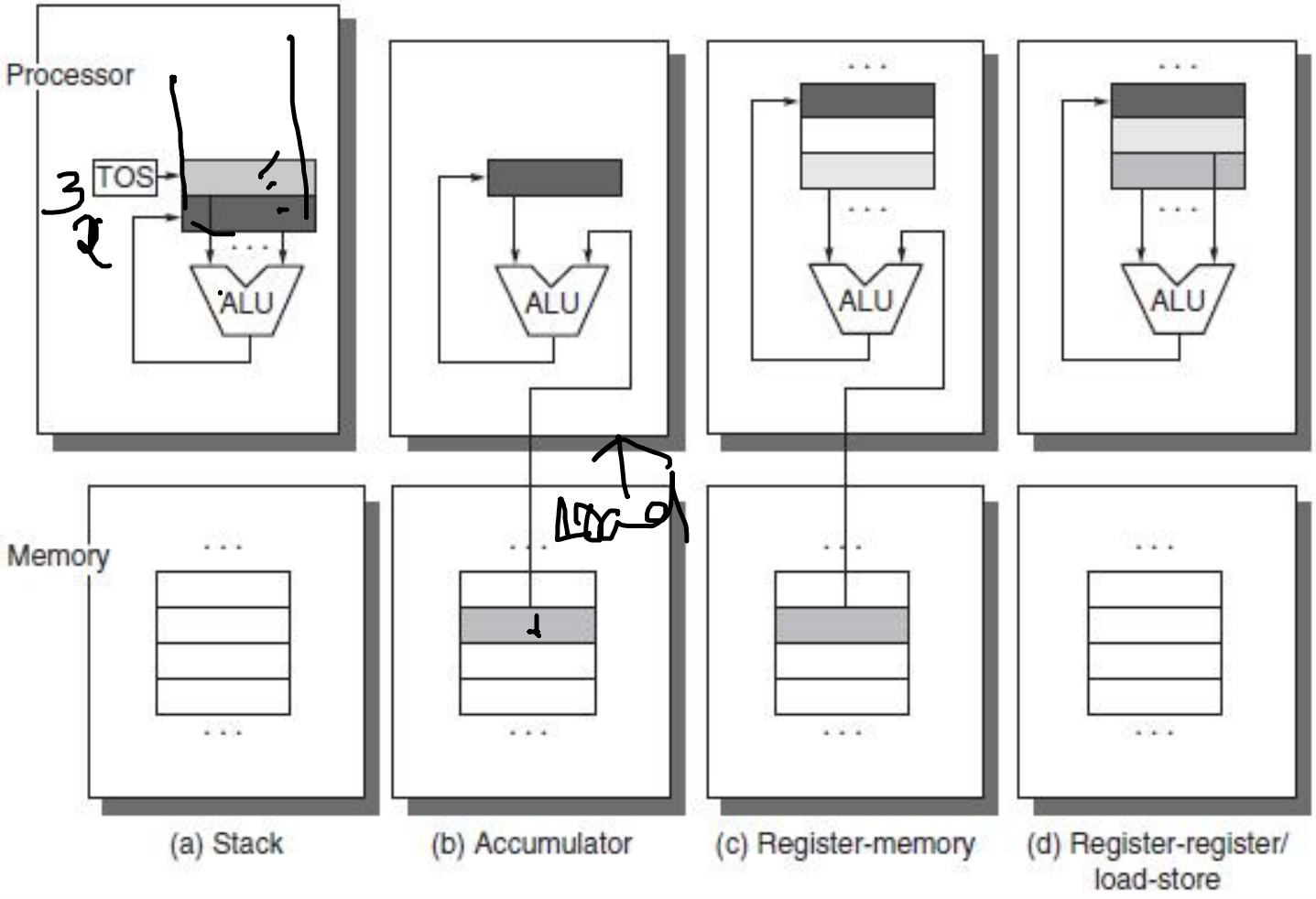
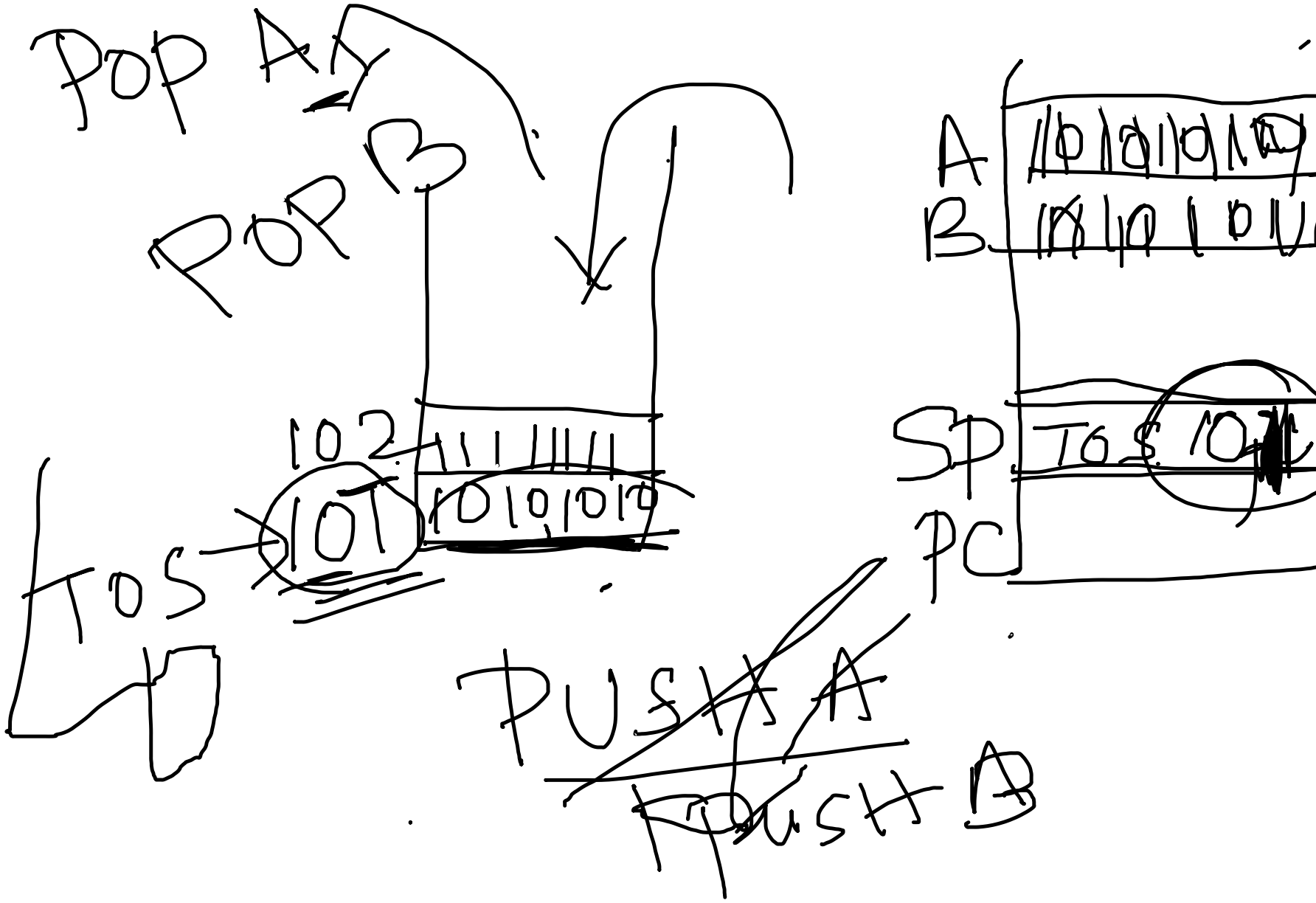


Figure A.1 Operand locations for four instruction set architecture classes. The arrows indicate whether the operand is an input or the result of the arithmetic-logical unit (ALU) operation, or both an input and result. Lighter shades indicate inputs, and the dark shade indicates the result. In (a), a Top Of Stack register (TOS) points to the top input operand, which is combined with the operand below. The first operand is removed from the stack, the result takes the place of the second operand, and TOS is updated to point to the result. All operands are implicit. In (b), the Accumulator is both an implicit input operand and a result. In (c), one input operand is a register, one is in memory, and the result goes to a register. All operands are registers in (d) and, like the stack architecture, can be transferred to memory only via separate instructions: push or pop for (a) and load or store for (d).


POP A
POP B
A
B
SP
PC
TOS
PUSH A
PUSH B

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

Figure A.2 The code sequence for C = A + B for four classes of instruction sets. Note that the Add instruction has implicit operands for stack and accumulator architectures and explicit operands for register architectures. It is assumed that A, B, and C all belong in memory and that the values of A and B cannot be destroyed. Figure A.1 shows the Add operation for each class of architecture.

- As the figures show, there are really two classes of register computers. One class can access memory as part of any instruction, called *register-memory* architecture.
- and the other can access memory only with load and store instructions called *load-store* architecture.
- Although most early computers used stack or accumulator-style architectures, virtually every new architecture designed after 1980 uses a load-store register architecture.

- The major reasons for the emergence of general-purpose register (GPR) computers are twofold. First, registers—like other forms of storage internal to the processor—are faster than memory.
- Second, registers are more efficient for a compiler to use than other forms of internal storage.
- On a stack computer the hardware must evaluate the expression in only one order, since operands are hidden on the stack, and it may have to load an operand multiple times.
- More importantly, registers can be used to hold variables. When variables are allocated to registers, the memory traffic reduces, the program speeds up (since registers are faster than memory), and the code density improves (since a register can be named with fewer bits than can a memory location).

- Two major instruction set characteristics divide GPR architectures. Both characteristics concern the nature of operands for a typical arithmetic or logical instruction (ALU instruction).
- The first concerns whether an ALU instruction has two or three operands. In the three-operand format, the instruction contains one result operand and two source operands.
- In the two-operand format, one of the operands is both a source and a result for the operation.
- The second distinction among GPR architectures concerns how many of the operands may be memory addresses in ALU instructions.

Memory Addressing

- Independent of whether the architecture is load-store or allows any operand to be a memory reference, it must define how memory addresses are interpreted and how they are specified.

- **Addressing Modes**
- addressing modes—how architectures specify the address of an object they will access.
- Addressing modes specify constants and registers in addition to locations in memory.
- When a memory location is used, the actual

  memory address specified by the

Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`Regs[R4] ← Regs[R4] + Regs[R3]`	When a value is in a register.
Immediate	`Add R4,#3`	`Regs[R4] ← Regs[R4] + 3`	For constants.
Displacement	`Add R4,100(R1)`	`Regs[R4] ← Regs[R4] + Mem[100 + Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	`Add R4,(R1)`	`Regs[R4] ← Regs[R4] + Mem[Regs[R1]]`	Accessing using a pointer or a computed address.
Indexed	`Add R3,(R1 + R2)`	`Regs[R3] ← Regs[R3] + Mem[Regs[R1] + Regs[R2]]`	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	`Add R1,(1001)`	`Regs[R1] ← Regs[R1] + Mem[1001]`	Sometimes useful for accessing static data; address constant ma need to be large.
Memory indirect	`Add R1,@(R3)`	`Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]`	If R3 is the address of a pointer then mode yields $*p$.
Autoincrement	`Add R1,(R2)+`	`Regs[R1] ← Regs[R1] + Mem[Regs[R2]]` `Regs[R2] ← Regs[R2] + d`	Useful for stepping through arra within a loop. R2 points to start array; each reference incremen R2 by size of an element, d.
Autodecrement	`Add R1, –(R2)`	`Regs[R2] ← Regs[R2] – d` `Regs[R1] ← Regs[R1] + Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implem a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1] ← Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] * d]`	Used to index arrays. May be applied to any indexed addressi mode in some computers.

Figure A.6 Selection of addressing modes with examples, meaning, and usage. In autoincrement/-decrem

Encoding an Instruction Set

- Instructions are encoded into a binary representation for execution by the processor.
- The operation is typically specified in one field, called the *opcode*. As we shall see, the important decision is how to encode the addressing modes with the operations.
- This decision depends on the range of addressing modes and the degree of independence between opcodes and modes.
- Some older computers have one to five operands with 10 addressing modes for each operand (see Figure A.6).
- For such a large number of combinations, typically a separate *address specifier* is needed for each operand: The address specifier tells what addressing mode is used to access the operand.
- At the other extreme are load-store computers with only one memory operand and only one or two addressing modes; obviously, in this case, the addressing mode can be encoded as part of the opcode.

- When encoding the instructions, the number of registers and the number of addressing modes both have a significant impact on the size of instructions, as the register field and addressing mode field may appear many times in a single instruction.
- for most instructions many more bits are consumed in encoding addressing modes and register fields than in specifying the opcode.
- The architect must balance several competing forces when encoding the instruction set:
- 1. The desire to have as many registers and addressing modes as possible.
- 2. The impact of the size of the register and addressing mode fields on the average instruction size and hence on the average program size.
- 3. A desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation. As a minimum, the architect wants

- instructions to be in multiples of bytes, rather than an arbitrary bit length.
- Many desktop and server architects have chosen to use a fixed-length instruction to gain implementation benefits while sacrificing average code size.
- Figure A.18 shows three popular choices for encoding the instruction set. The first we call *variable*, since it allows virtually all addressing modes to be with all operations. This style is best when there are many addressing modes and operations.
- The second choice we call *fixed*, since it combines the operation and the addressing mode into the opcode. Often fixed encoding will have only a single size for all instructions; it works best when there are few addressing modes and operations.
- The trade-off between variable encoding and fixed encoding is size of programs versus ease of decoding in the processor. Variable tries to use as few bits as possible to represent the program, but individual instructions can vary widely in both size and the amount of work to be performed.
- Let’s look at an 80x86 instruction to see an example of the

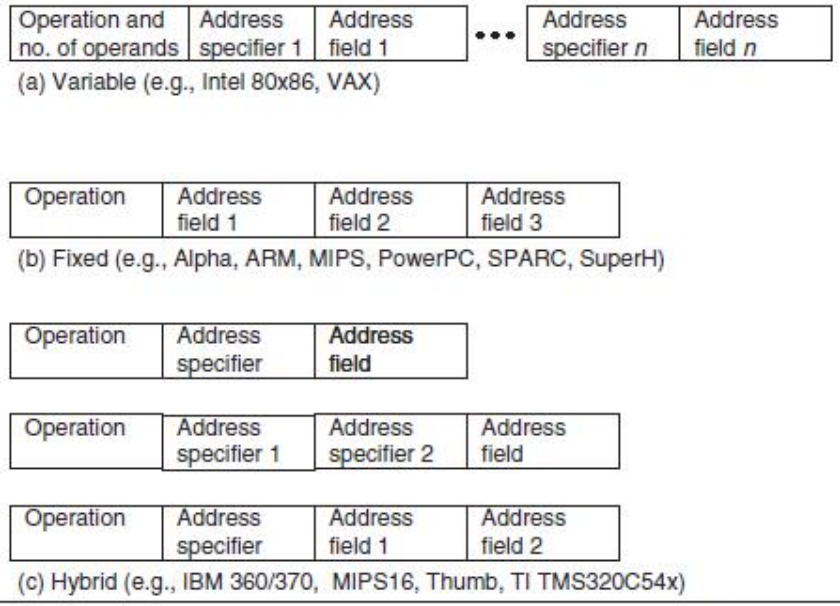


Figure A.18 Three basic variations in instruction encoding: variable length, fixed length, and hybrid. The variable format can support any number of operands, with each address specifier determining the addressing mode and the length of the specifier for that operand. It generally enables the smallest code representation, since unused fields need not be included. The fixed format always has the same number of operands, with the addressing modes (if options exist) specified as part of the opcode. It generally results in the largest code size. Although the fields tend not to vary in their location, they will be used for different purposes by different instructions. The hybrid approach has multiple formats specified by the opcode, adding one or two fields to specify the addressing mode and one or two fields to specify the operand address.

- The name add means a 32-bit integer add instruction with two operands, and this opcode takes 1 byte. An 80x86 address specifier is 1 or 2 bytes, specifying the source/destination register (EAX) and the addressing mode (displacement in this case) and base register (EBX) for the second operand. This combination takes 1 byte to specify the operands. When in 32-bit mode (see Appendix K), the size of the address field is either 1 byte or 4 bytes. Since 1000 is bigger than 28, the total length of the instruction is 1 + 1 + 4 = 6 bytes
- The length of 80x86 instructions varies between 1 and 17 bytes. 80x86 programs are generally smaller than the RISC architectures, which use fixed formats
- Given these two poles of instruction set design of variable and fixed, the third alternative immediately springs to mind: Reduce the variability in size and work of the variable architecture but provide multiple instruction lengths to reduce code size. This *hybrid* approach is the third encoding alternative,

- **Reduced Code Size in RISCs**
- As RISC computers started being used in embedded applications, the 32-bit fixed format became a liability since cost and hence smaller code are important.
- In response, several manufacturers offered a new hybrid version of their RISC instruction sets, with both 16-bit and 32-bit instructions. The narrow instructions support fewer operations, smaller address and immediate fields, fewer registers, and the two-address format rather than the classic three-address format of RISC computers.
- the ARM Thumb and MIPS MIPS16, which both claim a code size reduction of up to 40%.
- In contrast to these instruction set extensions, IBM simply compresses its standard instruction set and then adds hardware to decompress instructions as they are fetched from memory on an instruction cache miss. Thus, the instruction cache contains full 32-bit instructions, but compressed code is kept in main memory, ROMs, and the disk.
- The advantage of MIPS16 and Thumb is that instruction caches act as if they are about 25% larger, while IBM's CodePack means that compilers need not be changed to handle different instruction sets and instruction decoding can remain simple.