Instruction Sets: Characteristics and Functions Addressing Modes

Chapters 10 and 11,
William Stallings
Computer Organization and Architecture
7th Edition

What is an Instruction Set?

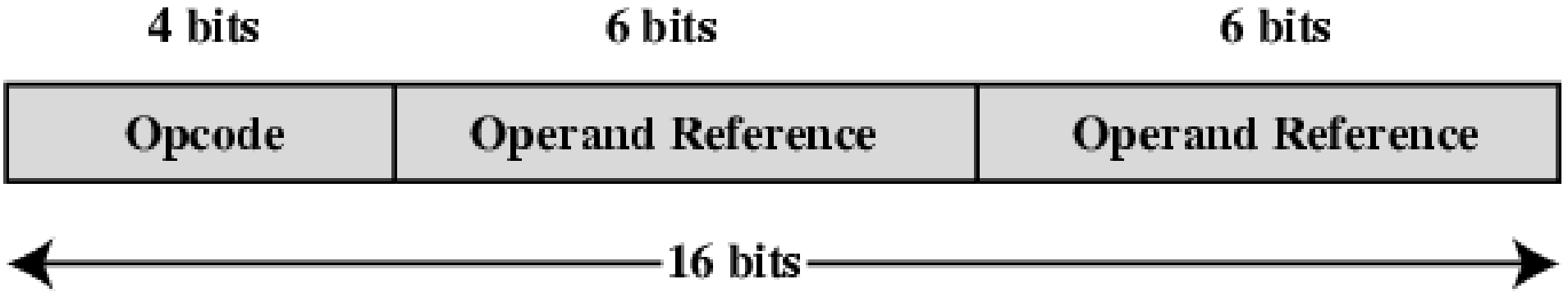
- The complete collection of instructions that are understood by a CPU
- Machine language: binary representation of operations and (addresses of) arguments
- Assembly language: mnemonic representation for humans, e.g.,

  OP A,B,C (meaning A <- OP(B,C))

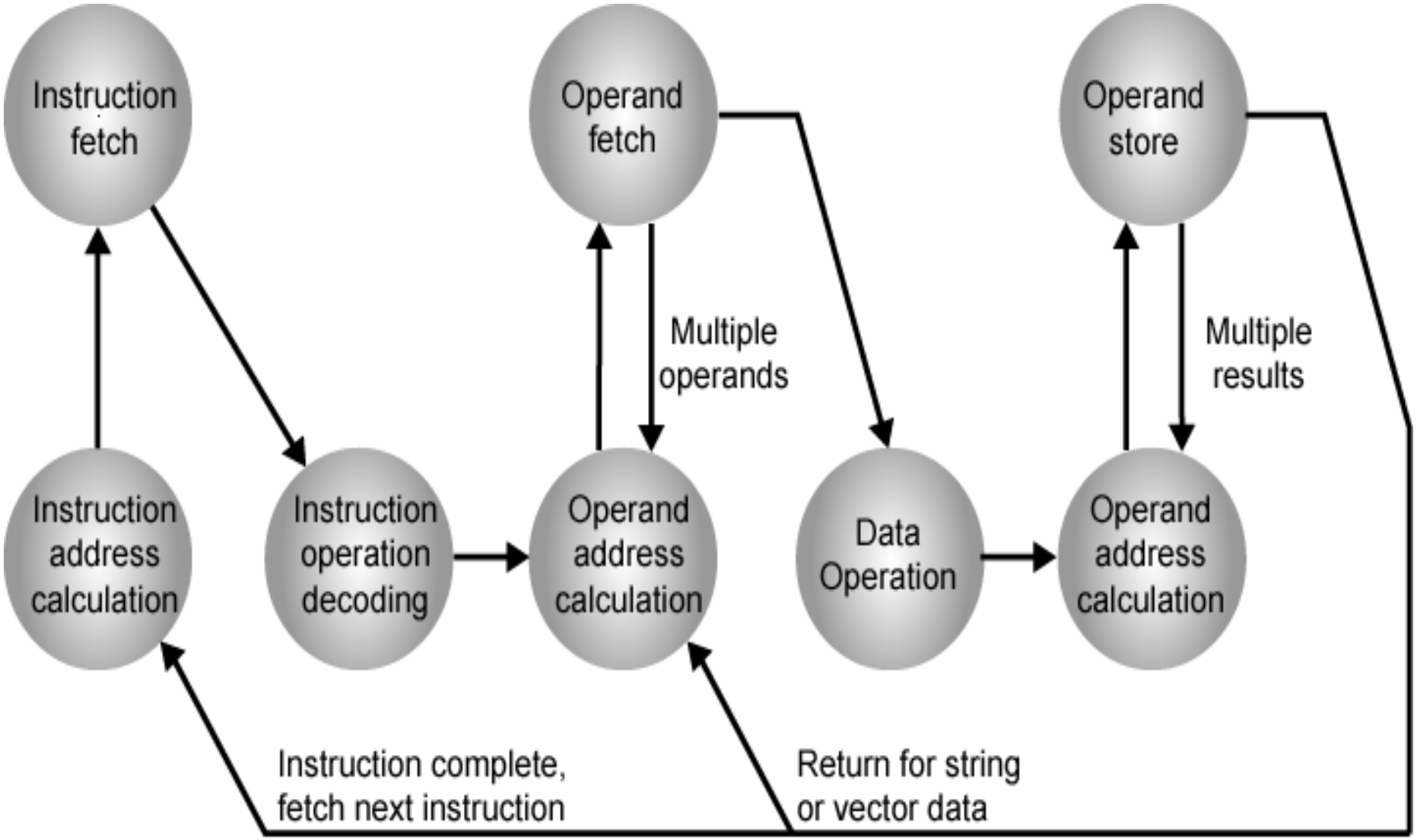
Elements of an Instruction

- Operation code (opcode)
  - Do this: ADD, SUB, MPY, DIV, LOAD, STOR
- Source operand reference
  - To this: (address of) argument of op, e.g. register, memory location
- Result operand reference
  - Put the result here (as above)
- Next instruction reference (often implicit)
  - When you have done that, do this: BR

Simple Instruction Format (using two addresses)



Instruction Cycle State Diagram



Design Decisions (1)

- Operation repertoire
  - How many ops?
  - What can they do?
  - How complex are they?
- Data types (length of words, integer representation)
- Instruction formats
  - Length of op code field
  - Length and number of addresses (e.g., implicit addressing)

Design Decisions (2)

- Registers
  - Number of CPU registers available
  - Which operations can be performed on which registers? General purpose and specific registers
- Addressing modes (see later)
- RISC v CISC

Instruction Types

- Data transfer: registers, main memory, stack or I/O
- Data processing: arithmetic, logical
- Control: systems control, transfer of control

Data Transfer

- Store, load, exchange, move, clear, set, push, pop
- Specifies: source and destination (memory, register, stack), amount of data
- May be different instructions for different (size, location) movements, e.g.,

  IBM S/390: L (32 bit word, R<-M), LH (halfword, R<-M), LR (word, R<-R), plus floating-point registers LER, LE, LDR, LD

  Or one instruction and different addresses, e.g. VAX: MOV

Input/Output

- May be specific instructions, e.g. INPUT, OUTPUT
- May be done using data movement instructions (memory mapped I/O)
- May be done by a separate controller (DMA): Start I/O, Test I/O

Arithmetic

- Add, Subtract, Multiply, Divide for signed integer (+ floating point and packed decimal) – may involve data movement
- May include
  - Absolute (|a|)
  - Increment (a++)
  - Decrement (a--)
  - Negate (-a)

Logical

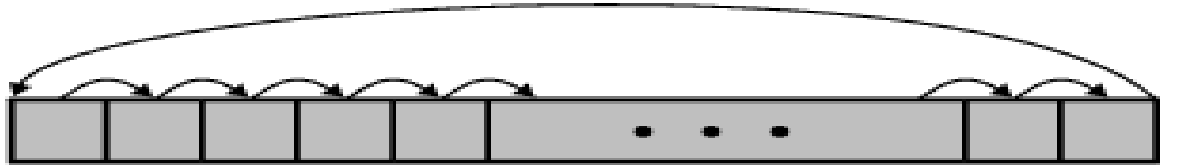
- Bitwise operations: AND, OR, NOT, XOR, TEST, CMP, SET
- Shifting and rotating functions, e.g.
  - logical right shift for unpacking: send 8-bit character from 16-bit word
  - arithmetic right shift: division and truncation for odd numbers
  - arithmetic left shift: multiplication without overflow

(a) Logical right shift

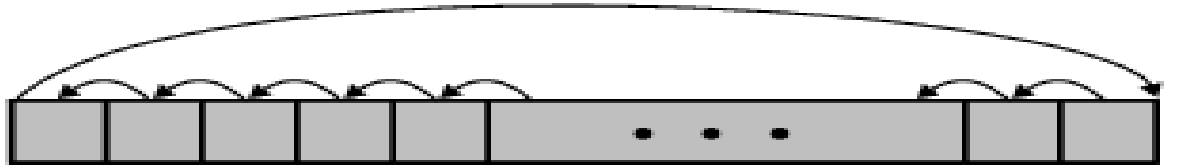
(b) Logical left shift

(c) Arithmetic right shift

(d) Arithmetic left shift



(e) Right rotate



(f) Left rotate

Systems Control

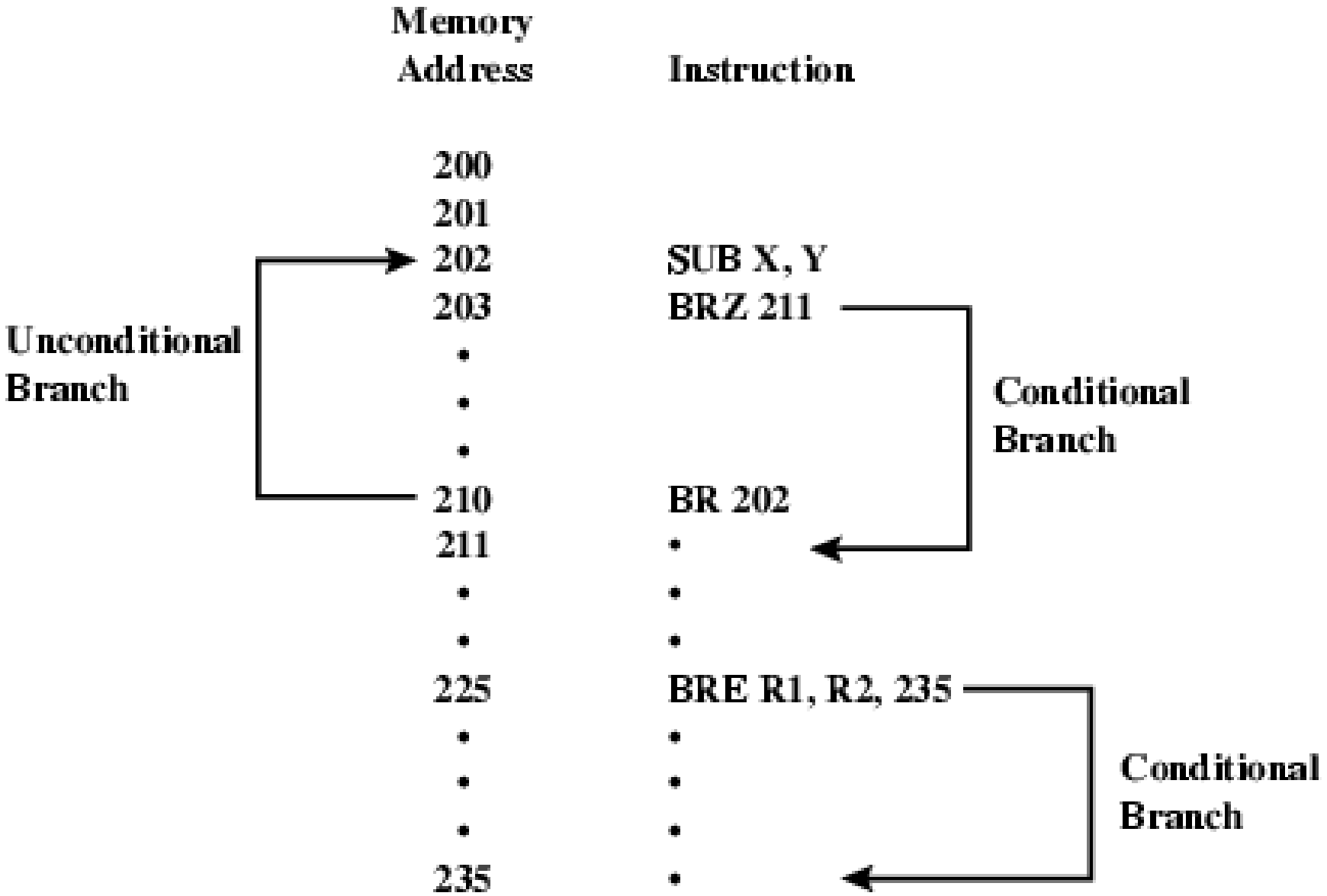
- Privileged instructions: accessing control registers or process table
- CPU needs to be in specific state
  - Ring 0 on 80386+
  - Kernel mode
- For operating systems use

Transfer of Control

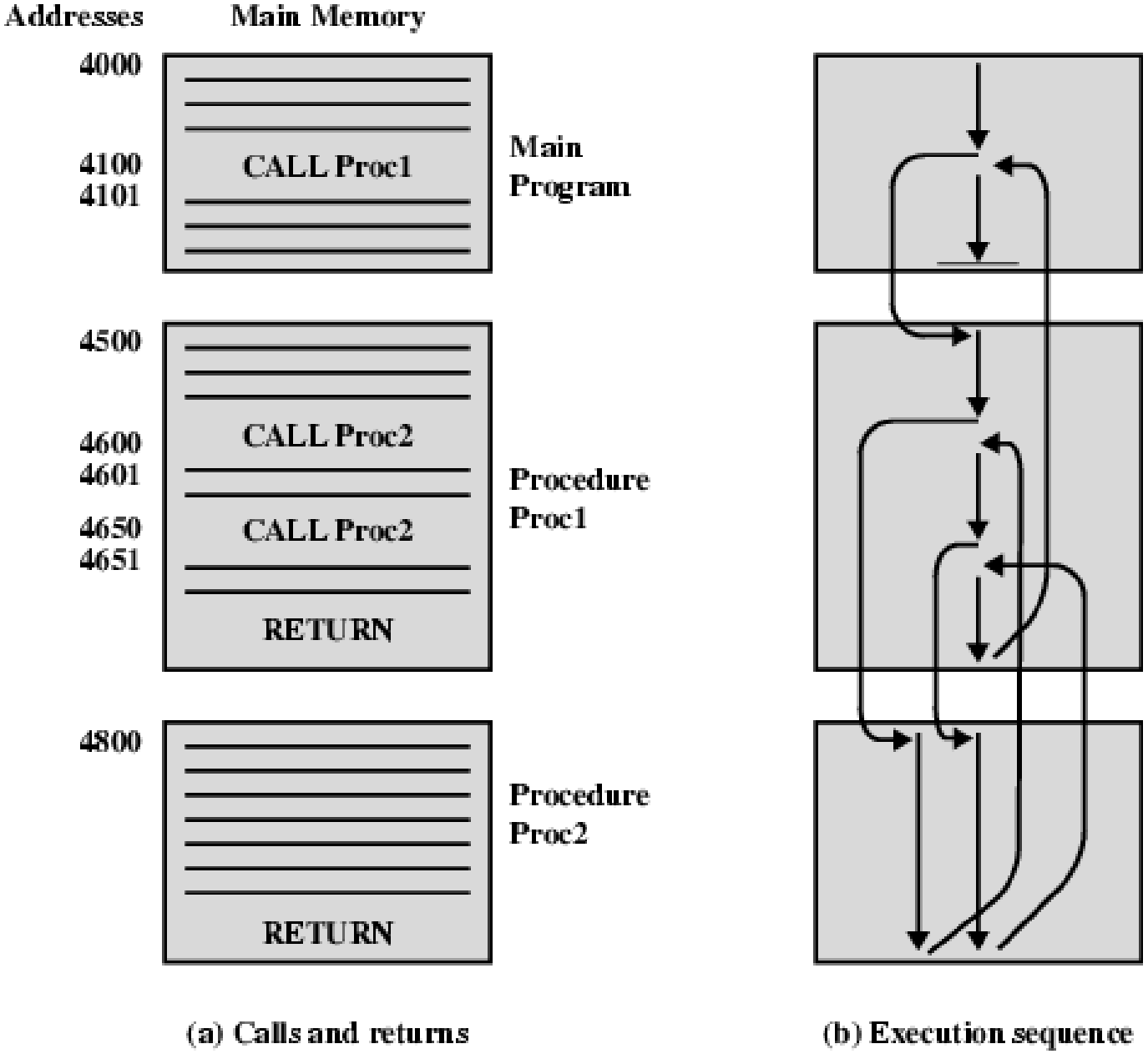
- Skip, e.g., increment and skip if zero:

  ISZ Reg1, cf. jumping out from loop
- Branch instructions: BRZ X (branch to X if result is zero), BRP X (positive), BRN X (negative), BRE X,R1,R2 (equal)
- Procedure (economy and modularity): call and return

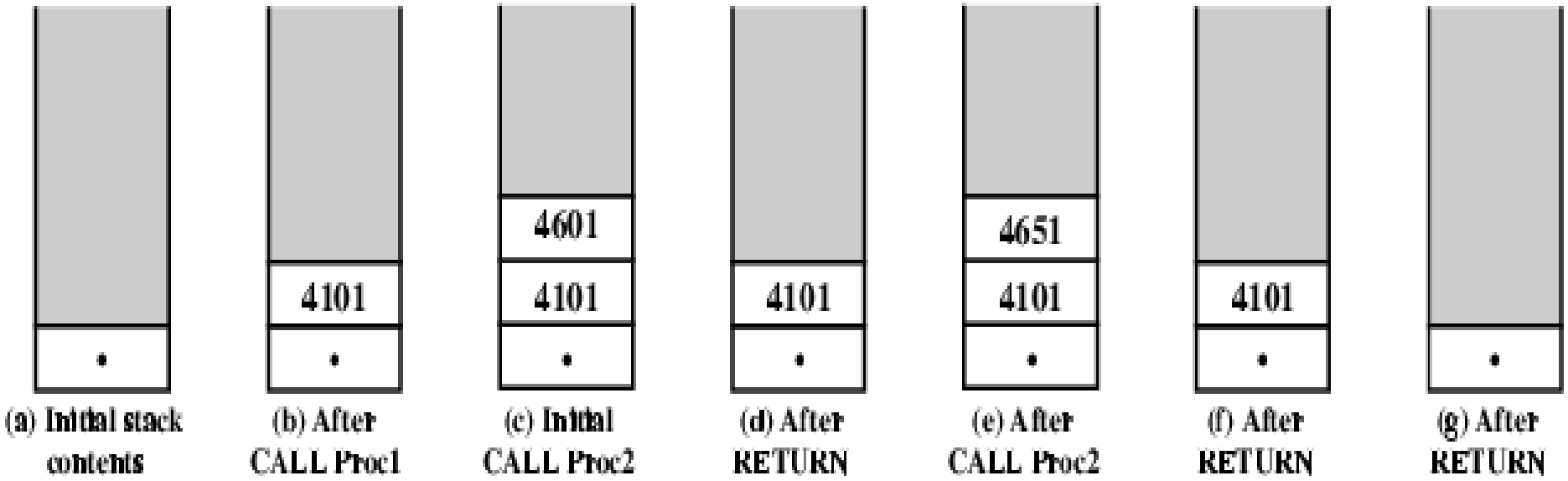
Branch Instruction



Nested Procedure Calls



Use of Stack:
Saving the return address for reentrant procedures



Types of Operand

- Addresses: immediate, direct, indirect, stack
- Numbers: integer or fixed point (binary, twos complement), floating point (sign, significand, exponent), (packed) decimal (246 = 0000 0010 0100 0110)
- Characters: ASCII (128 printable and control characters + bit for error detection)
- Logical Data: bits or flags, e.g., Boolean 0 and 1

Pentium Data Types

- Addressing is by 8 bit unit
- General data types: 8 bit Byte, 16 bit word, 32 bit double word, 64 bit quad word
- Integer: signed binary using twos complement representation
- (Un)packed decimal
- Near pointer: offset in segment
- Bit field
- Strings
- Floating point

Instruction Formats

- Layout of bits in an instruction
- Includes opcode
- Includes (implicit or explicit) operand(s)
- Usually more than one instruction format in an instruction set

Instruction Length

- Affected by and affects:
  - Memory size
  - Memory organization - addressing
  - Bus structure, e.g., width
  - CPU complexity
  - CPU speed
- Trade off between powerful instruction repertoire and saving space

Allocation of Bits

- Number of addressing modes: implicit or additional bits specifying it
- Number of operands
- Register (faster, limited size and number, 32) versus memory
- Number of register sets, e.g., data and address (shorter addresses)
- Address range
- Address granularity (e.g., by byte)

Number of Addresses

- More addresses
  - More complex (powerful?) instructions
  - More registers - inter-register operations are quicker
  - Less instructions per program
- Fewer addresses
  - Less complex (powerful?) instructions
  - More instructions per program, e.g. data movement
  - Faster fetch/execution of instructions
- Example: Y=(A-B):[(C+(DxE)]

3 addresses

Operation Result, Operand 1, Operand 2

- Not common
- Needs very long words to hold everything

SUB Y,A,B	Y <- A-B
MPY T,D,E	T <- DxE
ADD T,T,C	T <- T+C
DIV Y,Y,T	Y <- Y:T

2 addresses

One address doubles as operand and result

- Reduces length of instruction
- Requires some extra work: temporary storage

MOVE Y,A	Y <- A
SUB Y,B	Y <- Y-B
MOVE T,D	T <- D
MPY T,E	T <- TxE
ADD T,C	T <- T+C
DIV Y,T	Y <- Y:T

1 address

Implicit second address, usually a register (accumulator, AC)

LOAD D	AC <- D
MPY E	AC <- ACxE
ADD C	AC <- AC+C
STOR Y	Y <- AC
LOAD A	AC <- A
SUB B	AC <- AC-B
DIV Y	AC <- AC:Y
STOR Y	Y <- AC

0 (zero) addresses

All addresses implicit, e.g. ADD

- Uses a stack, e.g. pop a, pop b, add
- c = a + b

Addressing Modes

- Immediate
- Direct
- Indirect
- Register
- Register Indirect
- Displacement (Indexed)
- Stack

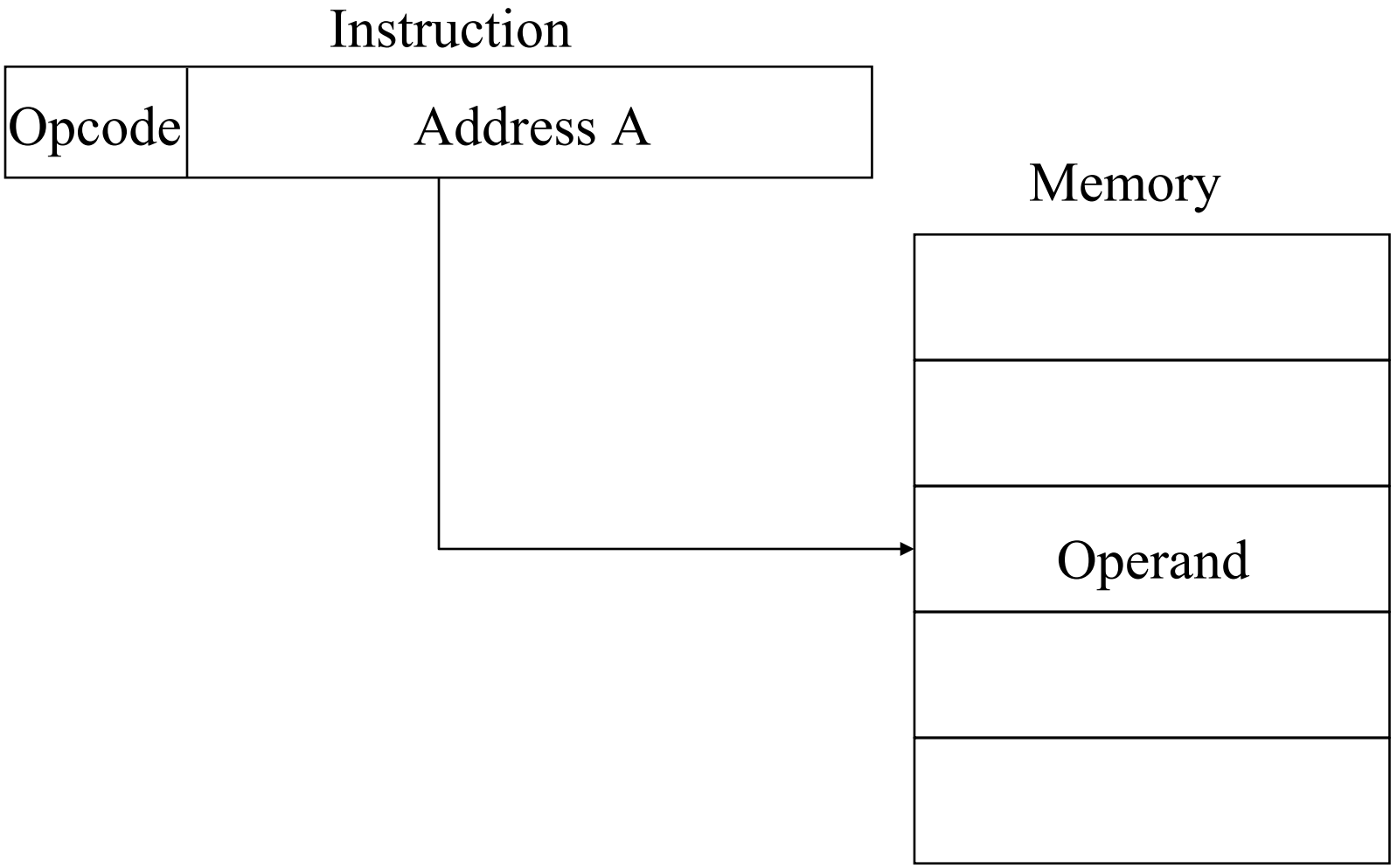
Immediate Addressing

- Operand is part of instruction
- Operand = address field
- e.g., ADD 5 or ADD #5
  - Add 5 to contents of accumulator
  - 5 is operand
- No memory reference to fetch data
- Fast
- Limited range

Direct Addressing

- Address field contains address of operand
- Effective address (EA) = address field (A)
- e.g., ADD A
  - Add contents of cell A to accumulator
  - Look in memory at address A for operand
- Single memory reference to access data
- No additional calculations needed to work out effective address
- Limited address space (length of address field)

Direct Addressing Diagram



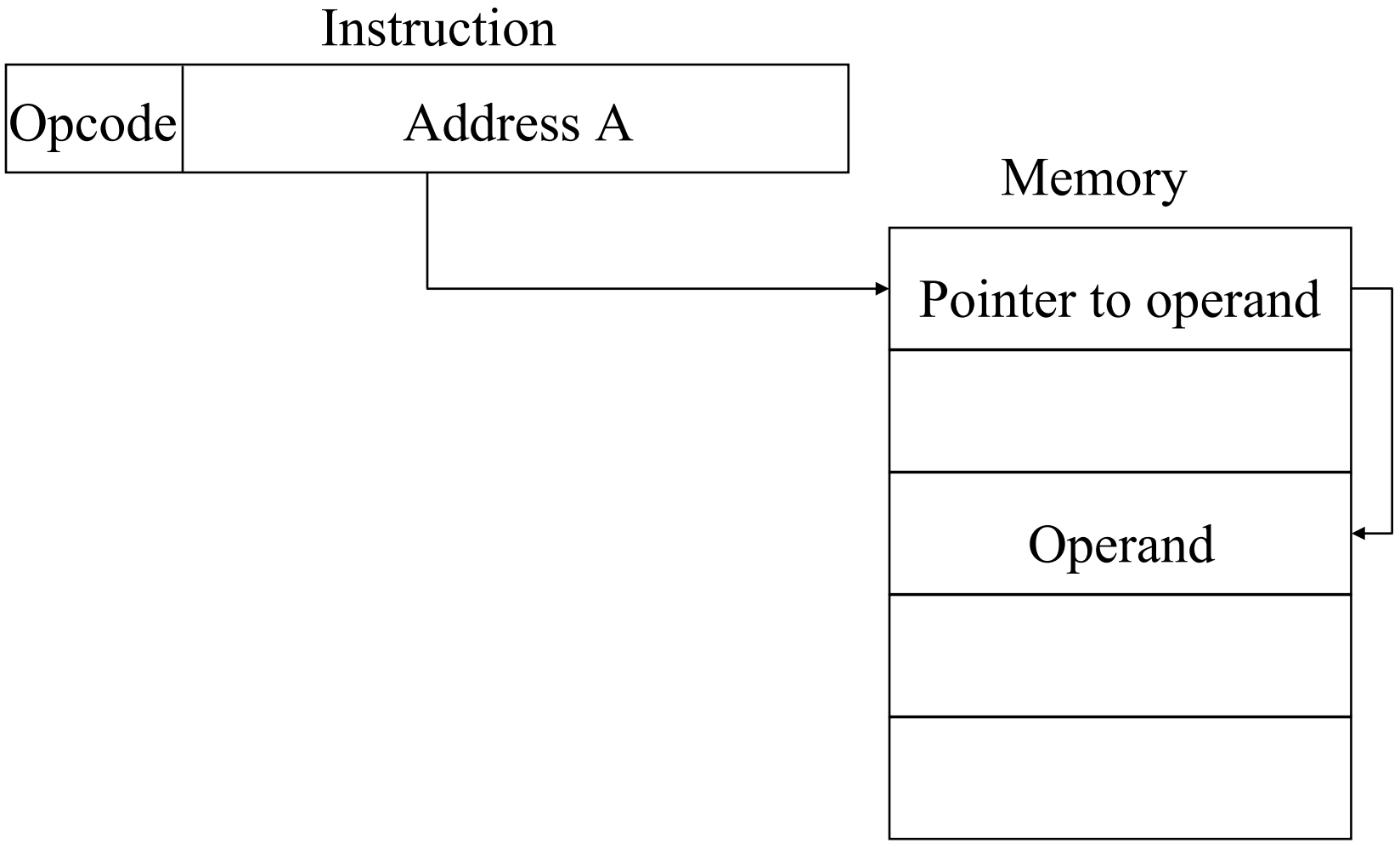
Indirect Addressing (1)

- Memory cell pointed to by address field contains the address of (pointer to) the operand
- EA = (A)
  - Look in A, find address (A) and look there for operand
- E.g. ADD (A)
  - Add contents of cell pointed to by contents of A to accumulator

Indirect Addressing (2)

- Large address space
- 2^n where n = word length
- May be nested, multilevel, cascaded
  - e.g. EA = (((A)))
- Multiple memory accesses to find operand
- Hence slower

Indirect Addressing Diagram



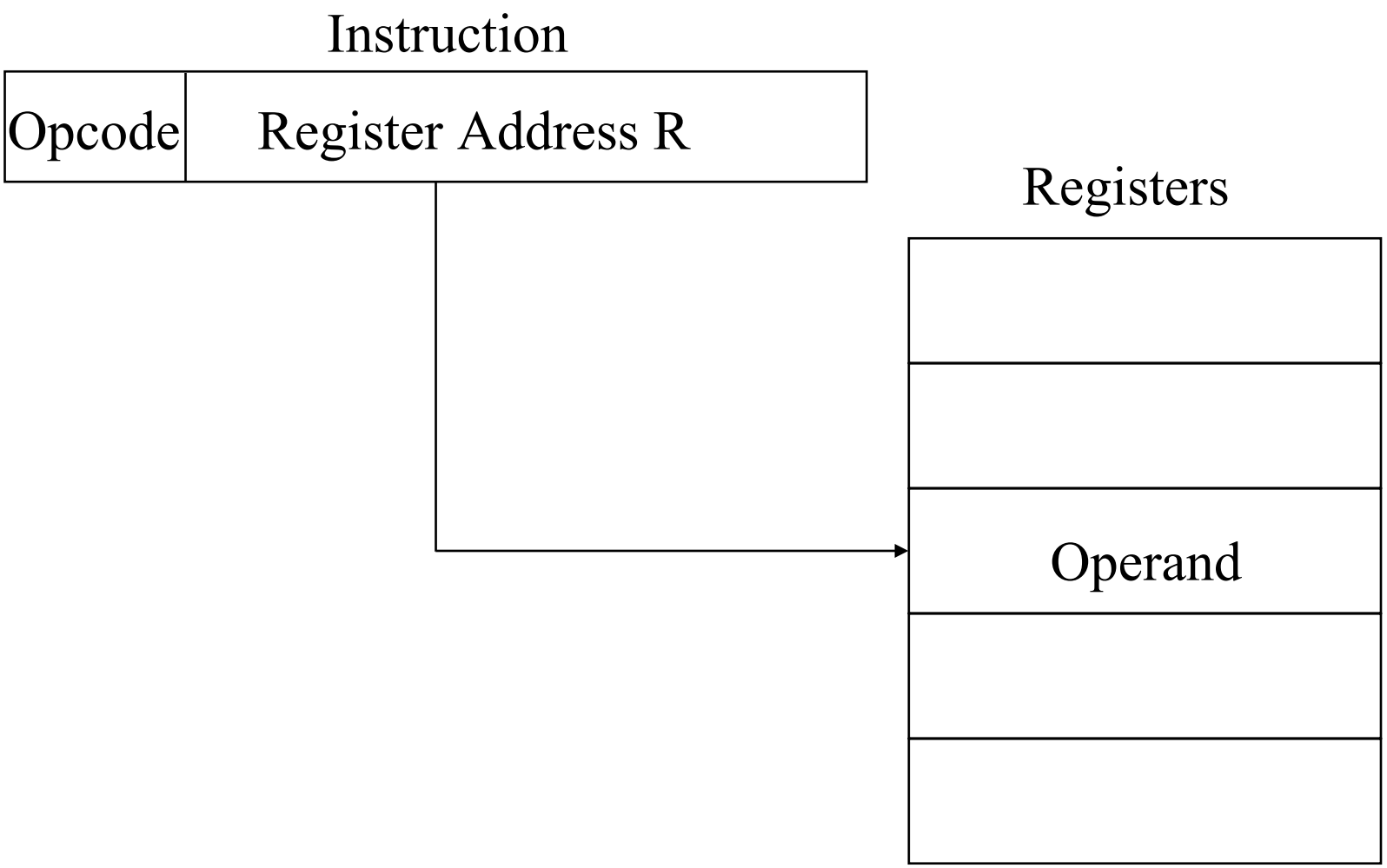
Register Addressing (1)

- Operand is held in register named in address field
- EA = R
- Limited number of registers
- Very small address field needed
  - Shorter instructions
  - Faster instruction fetch

Register Addressing (2)

- No memory access
- Very fast execution
- Very limited address space
- Multiple registers helps performance
  - Requires good assembly programming or compiler writing – see register renaming
- cf. direct addressing

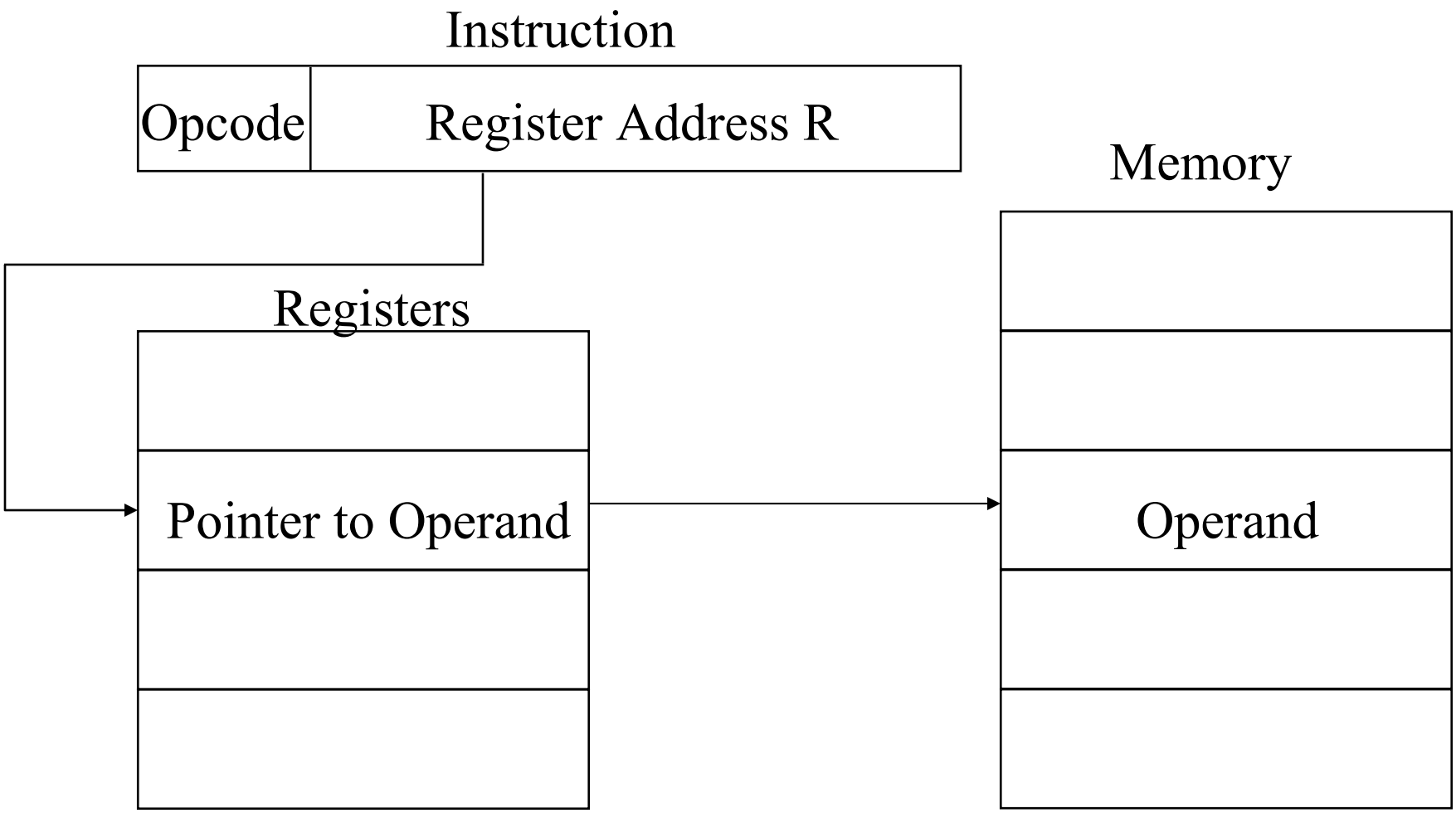
Register Addressing Diagram



Register Indirect Addressing

- Cf. indirect addressing
- EA = (R)
- Operand is in memory cell pointed to by contents of register R
- Large address space (2^n)
- One fewer memory access than indirect addressing

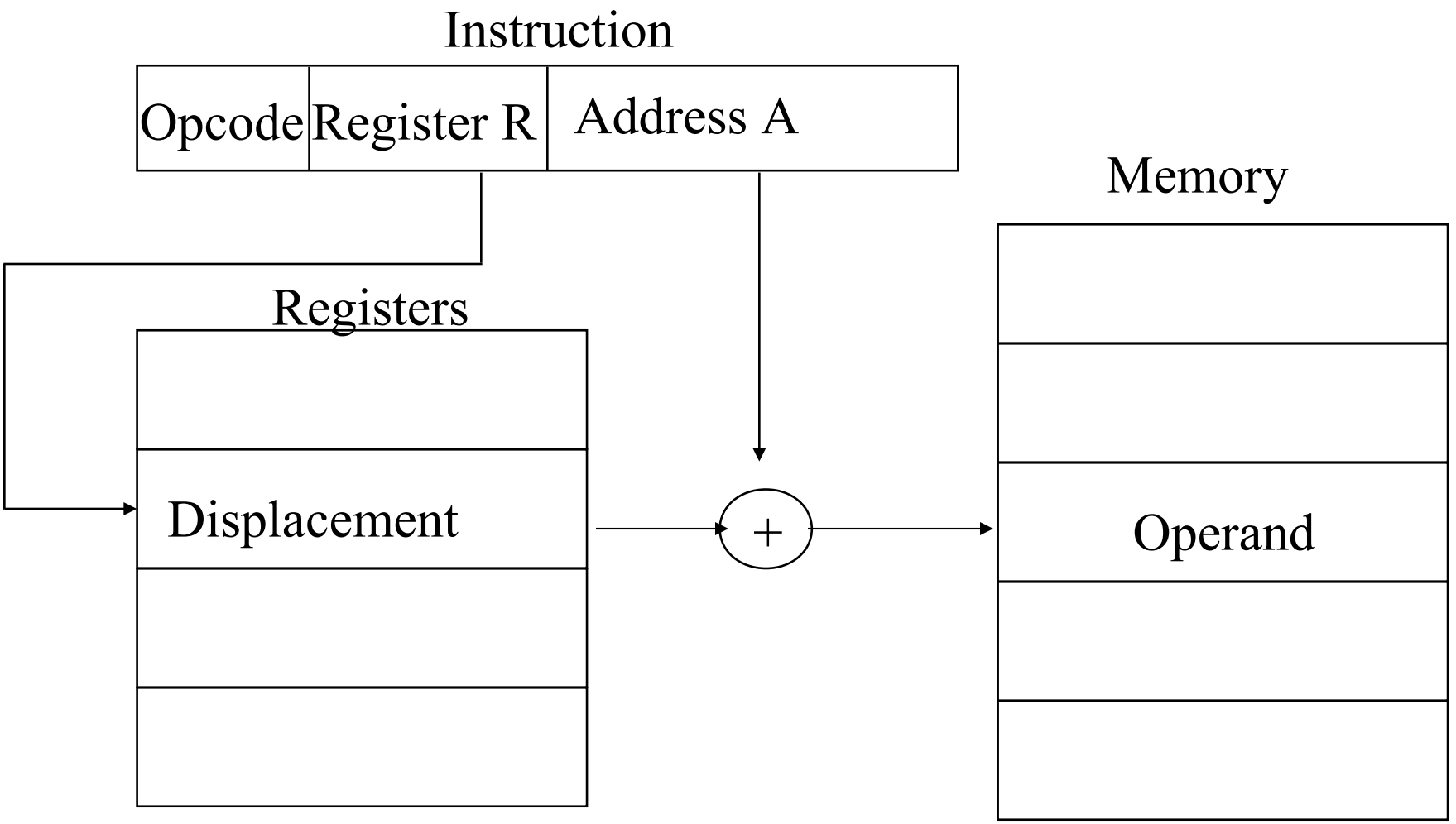
Register Indirect Addressing Diagram



Displacement Addressing

- EA = A + (R)
- Address field hold two values
  - A = base value
  - R = register that holds displacement
  - or vice versa
- See segmentation

Displacement Addressing Diagram



Relative Addressing

- A version of displacement addressing
- R = Program counter, PC
- EA = A + (PC)
- i.e., get operand from A cells away from current location pointed to by PC
- cf. locality of reference & cache usage

Base-Register Addressing

- A holds displacement
- R holds pointer to base address
- R may be explicit or implicit
- e.g., segment registers in 80x86

Indexed Addressing

- A = base
- R = displacement
- EA = A + R
- Good for iteration, e.g., accessing arrays
  - EA = A + R
  - R++
- Sometimes automated: autoindexing (signalled by one bit in instruction)

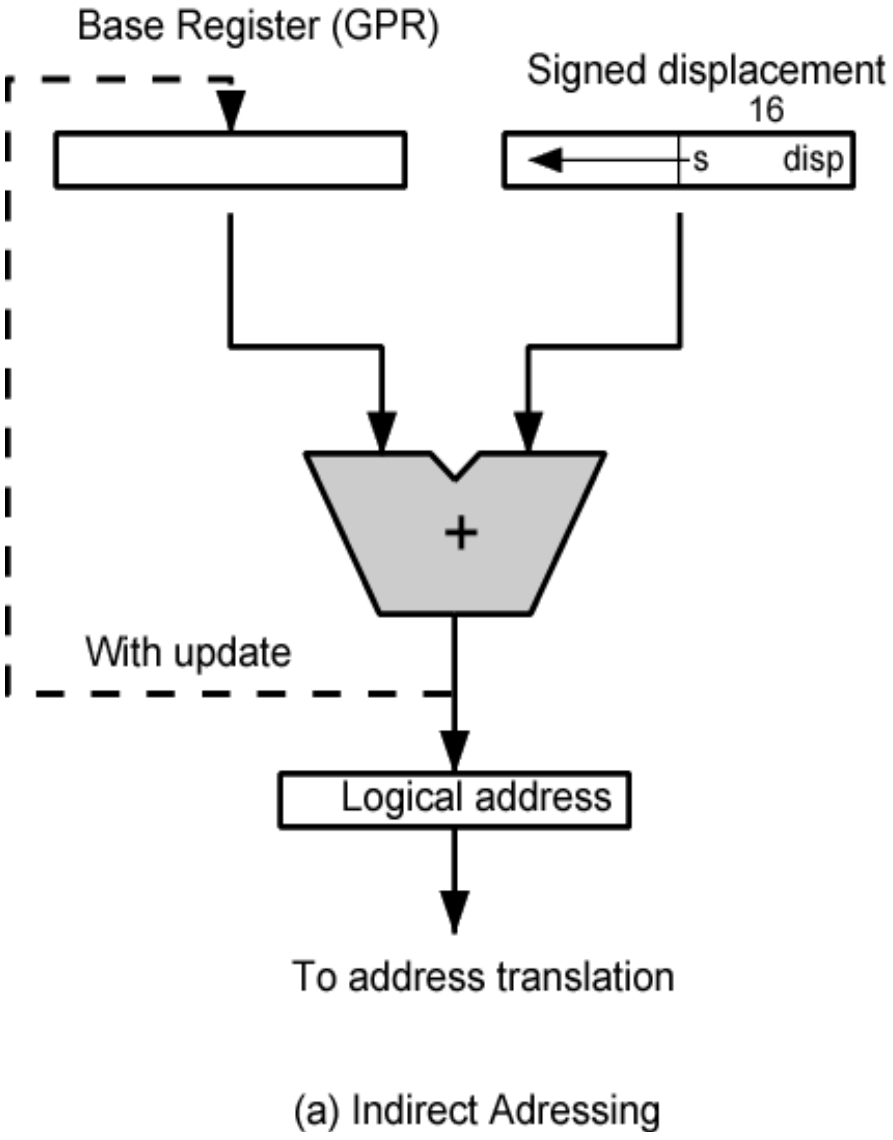
Stack Addressing

- Operand is (implicitly) on top of stack
- e.g.
  - ADD Pop top two items from stack and add and push result on top

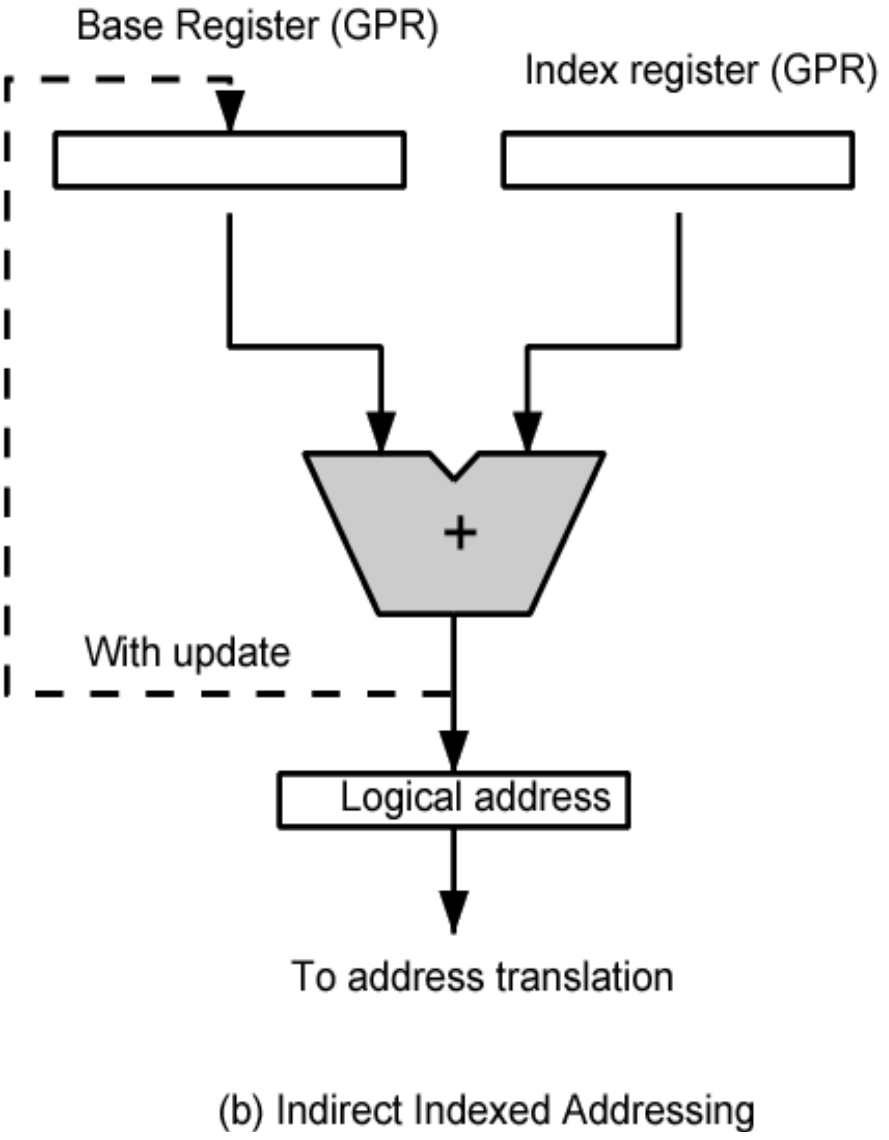
PowerPC Addressing Modes

- Load/store architecture (see next slide):
  - Displacement and indirect indexed
  - EA = base + displacement/index
  - with updating base by computed address
- Branch address
  - Absolute
  - Relative (see loops): (PC) + I
  - Indirect: from register
- Arithmetic
  - Operands in registers or part of instruction
  - For floating point: register only

PowerPC Memory Operand Addressing Modes



(a) Indirect Adressing



(b) Indirect Indexed Addressing