William Stallings
Computer Organization
and Architecture

Chapter 10

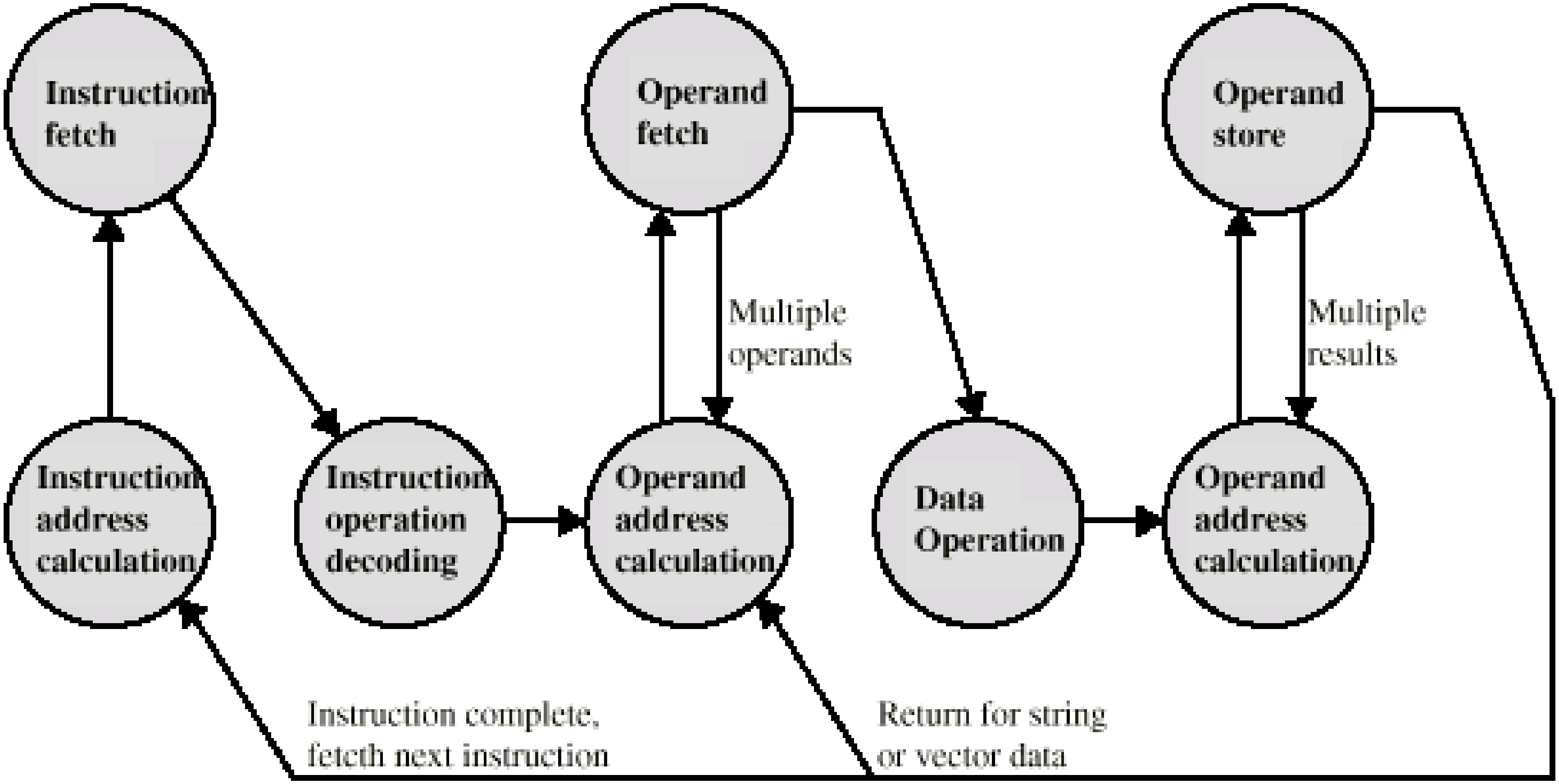
Instruction Sets:

Characteristics and Functions

What is an instruction set?

- The complete collection of instructions that are understood by a CPU
- The instruction set is the specification of the expected behaviour of the CPU
- How this behaviour is obtained is a matter of CPU implementation

Instruction Cycle



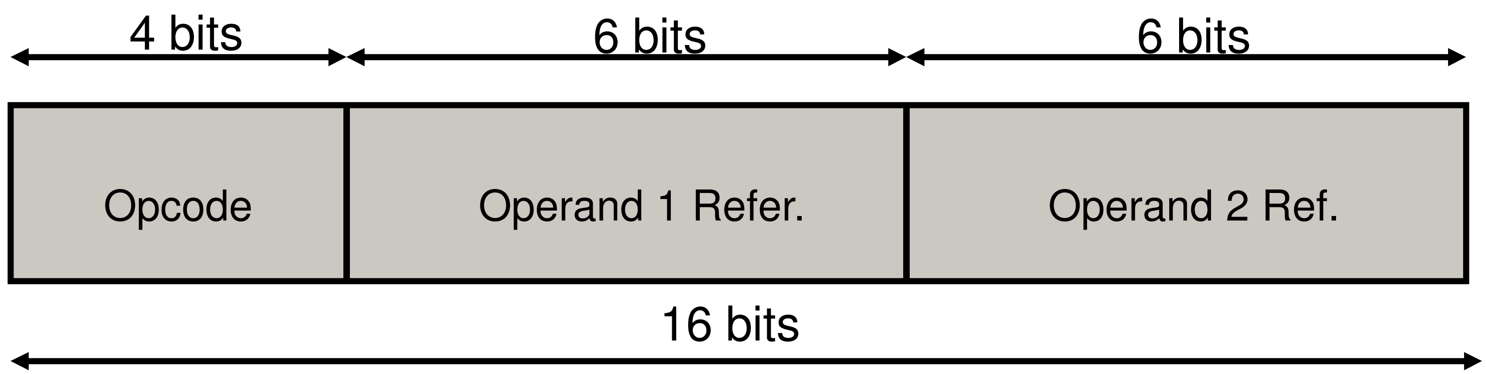
Elements of an Instruction

- Operation code (Opcode)
  - Do this
- Source Operand(s) reference(s)
  - To this (and this …)
- Result Operand reference
  - Put the answer here
- The Opcode is the only mandatory element

Instruction Types

- Data processing
- Data storage (main memory)
- Data movement (internal transfer and I/O)
- Program flow control

Instruction Representation



- There may be many instruction formats
- For human convenience a symbolic representation is used for both opcodes (MPY) and operand references (RA RB)
  - e.g. 0110 001000 001001 MPY RA RB
    (machine code) (symbolic - assembly code)

Design Decisions (1)

- Operation repertoire
  - How many opcodes?
  - What can they do?
  - How complex are they?
- Data types
- Instruction formats
  - Length and structure of opcode field
  - Number and length of reference fields

Design Decisions (2)

- Registers
  - Number of CPU registers available
  - Which operations can be performed on which registers?
- Addressing modes (later…)

Types of Operand references

- Main memory
- Virtual memory (usually slower)
- Cache (usually faster)

- I/O device (slower)
- CPU registers (faster)

Number of References/ Addresses/ Operands

- 3 references
  - ADD RA RB RC — $RA+RB \rightarrow RC$
- 2 references (reuse of operands)
  - ADD RA RB — $RA+RB \rightarrow RA$
- 1 reference (some implicit operands)
  - ADD RA — $Acc+RA \rightarrow Acc$
- 0 references (all operands are implicit)
  - S_ADD — $Acc+Top(Stack) \rightarrow Acc$

How Many References

- More references
  - More complex (powerful?) instructions
  - Fewer instructions per program
  - Slower instruction cycle
- Fewer references
  - Less complex (powerful?) instructions
  - More instructions per program
  - Faster instruction cycle

Example

- Compute (A-B)/(A+(C*D)), assuming each of them is in a read-only register which cannot be modified.
- Additional registers X and Y can be used if needed.
- Try to minimize the number of operations
- Incremental constraints on the number of operands allowed for instructions

Example - 3 operands (1)

- Syntax

  <operation><destination><source-1><source-2>

- Meaning

  <source-1><operation><source-2> → <destination>

- Available instructions

  ADD, SUB, MUL, DIV

Example - 3 operands (2)

- Solution
  - MUL X C D   C*D → X
  - ADD X A X   A+X → X
  - SUB Y A B   A-B → Y
  - DIV X Y X   Y/X → X

Example – 2 operands (1)

- Syntax

  <operation><destination><source>

- Meaning (the destination is also the first source operand)

  <destination><operation><source> → <destination>

- Available instructions

  ADD, SUB, MUL, DIV

  MOV <A> <B> (B → A)

Example – 2 operands (2)

- Solution (using the new movement instruction)
  - MOV X C   C → X
  - MUL X D   X*D → X
  - ADD X A   X+A → X
  - MOV Y A   A → Y
  - SUB Y B   Y-B → Y
  - DIV Y X   Y/X → Y

- Can we avoid using MOV ?

Example – 2 operands (3)

- A different solution (a trick avoids using the new movement instruction)
  - SUB X X X-X → X (set X to zero)
  - ADD X C X+C → X (move C to X)
  - MUL X D X*D → X
  - ADD X A X+A → X
  - SUB Y Y Y-Y → Y (set Y to zero)
  - ADD Y A Y+A → Y (move A to Y)
  - SUB Y B Y-B → Y
  - DIV Y X Y/X → Y

Example – 1 operand (1)

- Syntax

  <operation><source>

- Meaning (a given register, e.g. the accumulator, is both the destination and the first source operand)

  ACCUMUL. <operation><source> → ACCUMUL.

- Available instructions

  ADD, SUB, MUL, DIV
  LOAD <X> (X → Acc)
  STORE <X> (Acc → X)

Example – 1 operand (2)

- Solution (using the new instructions to move data to and from the accumulator)
  - LOAD C   C → Acc
  - MUL D   Acc*D → Acc
  - ADD A   Acc+A → Acc
  - STORE X   Acc → X
  - LOAD A   A → Acc
  - SUB B   Acc-B → Acc
  - DIV X   Acc/X → Acc

- Can we avoid using LOAD and STORE?

Example – 1 operand (3)

- A different solution (assumes at the beginning the accumulator stores zero, but STORE is needed since no other instruction moves data towards a register)
  - ADD C — Acc+C → Acc — (move C to Accumul.)
  - MUL D — Acc*D → Acc
  - ADD A — Acc+A → Acc
  - STORE X — Acc → X
  - SUB Acc — Acc-Acc → Acc — (set Acc. to zero)
  - ADD A — Acc+A → Acc — (move A to Accumul.)
  - SUB B — Acc-B → Acc
  - DIV X — Acc/X → Acc

Example – 0 operands (1)

- Syntax

  <operation>

- Meaning (all *arithmetic* operations make reference to pre-defined registers, e.g. the accumulator and the top of the stack)

  ACCUMUL. <operation> TOP(STACK) → ACCUMUL.

- Available instructions

  ADD, SUB, MUL, DIV

  LOAD <X> (X → Acc)

  PUSH <X> (X → STACK)

  POP <X> (TOP(STACK) → X) *the top element is deleted from the stack*

Example – 0 operands (2)

- Here is the solution

LOAD C	C → Acc
PUSH D	D → Top(Stack)
MUL	Acc*Top(Stack) → Acc
PUSH A	A → Top(Stack)
ADD	Acc+Top(Stack) → Acc
PUSH Acc	Acc → Top(Stack)
PUSH B	B → Top(Stack)
LOAD A	A → Acc
SUB	Acc-Top(Stack) → Acc
POP X	Top(Stack) → X
DIV	Acc/Top(Stack) → Acc

- Can we use a LOAD without arguments ?

Example – 0 operands (3)

- Just substitutes the following instruction for a given register R
  - LOAD <R> <R> → Acc

  with the following three equivalent instructions
  - PUSH <R> <R> → Top(Stack)
  - POP X Top(Stack) → X
  - LOAD X → Acc
- Can we avoid using LOAD ?

Example – 0 operands (4)

- This solution uses only PUSH and POP to load values into the Accumulator
  - PUSH C — C → Top(Stack)
  - POP Acc — Top(Stack) → Acc
  - PUSH D — D → Top(Stack)
  - MUL — Acc*Top(Stack) → Acc
  - PUSH A — A → Top(Stack)
  - ADD — Acc+Top(Stack) → Acc
  - PUSH Acc — Acc → Top(Stack)
  - PUSH B — B → Top(Stack)
  - PUSH A — A → Top(Stack)
  - POP Acc — Top(Stack) → Acc
  - SUB — Acc-Top(Stack) → Acc
  - POP X — Top(Stack) → X
  - DIV — Acc/Top(Stack) → Acc

Types of Operand

- Addresses
- Numbers
  - Integer/floating point
- Characters
  - ASCII etc.
- Logical Data
  - Bits or flags
- The type of operand is determined by the used instruction

Instruction Types (more detail)

- Arithmetic
- Logical
- Conversion
- Transfer of data (internal)
- I/O
- System Control
- Transfer of Control

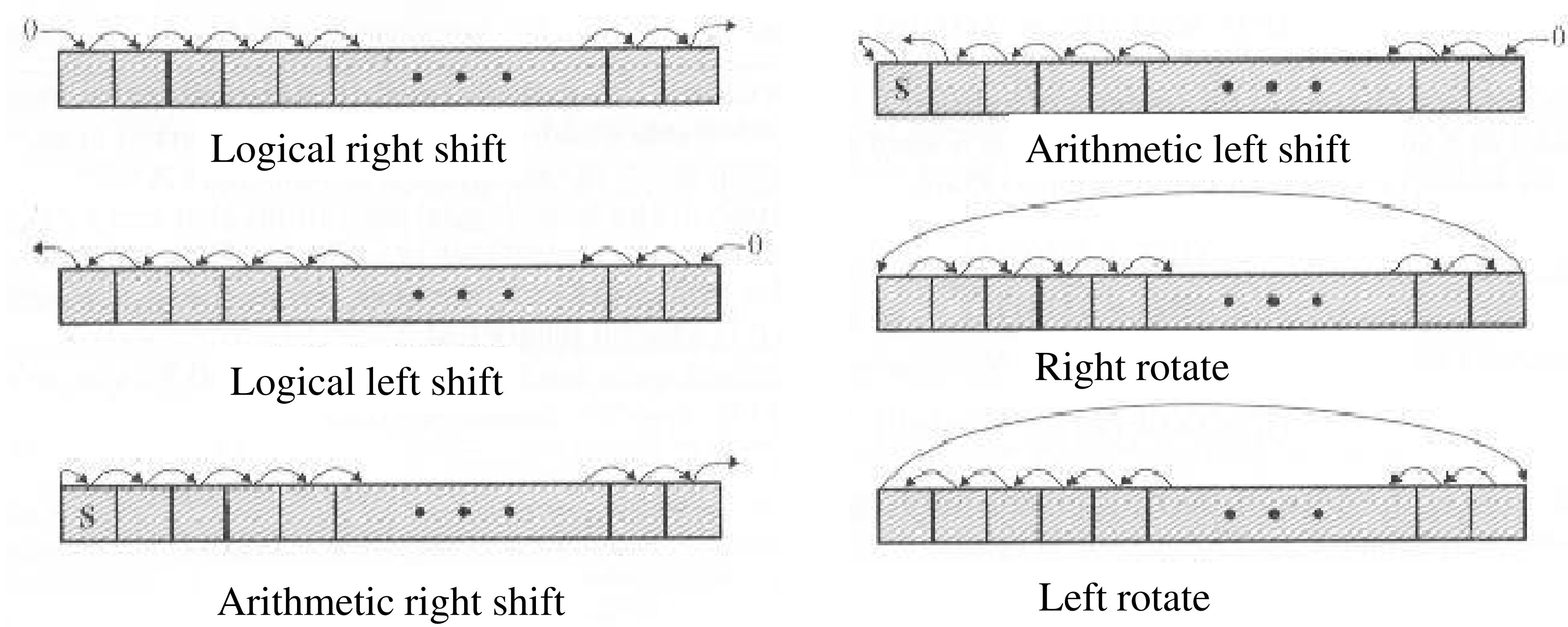
Arithmetic

- Add, Subtract, Multiply, Divide
- Signed Integer
- Floating point ?
- May include
  - Increment (a++)
  - Decrement (a--)
  - Negate (-a)
  - Absolute (|a|)
  - Arithmetic shift (take care of sign bit and overflow!)

Logical

- Bit manipulation operations
  - shift, rotate, …
- Boolean logic operations (bitwise)
  - AND, OR, NOT, …
- Test operations
  - To set (indirectly through the ALU) control bits in the Program Status Word

Shift and rotate operations



Conversion

- e.g. Binary to Decimal

Transfer of data

- Specify
  - Source and Destination
  - Amount of data
- May be different instructions for different movements
  - e.g. MOVE, STORE, LOAD, PUSH
- Or one instruction and different addresses
  - e.g. MOVE B C, MOVE A M, MOVE M A, MOVE A S

Input/Output

- May be specific instructions
- May be done using data movement instructions (memory mapped)
- May be done by a separate controller (DMA)

System Control

- For managing the system is convenient to have *reserved* instruction executable only by some programs with special privileges (e.g., to halt a running program)
- These privileged instructions may be executed only if CPU is in a specific state (or mode)
- *Kernel* or *supervisor* or *protected* mode
- Privileged programs are part of the *operating system* and run in protected mode

Transfer of Control (1)

- Needed to
  - Take decisions (branch)
  - Execute repetitive operations (loop)
  - Structure programs (subroutines)
- Branch (examples)
  - BRA X: branch (i.e., go) to X (unconditional jump)
  - BRZ X: branch to X if accumulator value is 0
  - BRE R1, R2, X: branch to X if (R1)=(R2)

An example

```
200      ...
201      ...
202  SUB X, Y
203  BRZ 211
...      ...
...      ...
...      ...
210  BRA 202
211      ...
...      ...
...      ...
225  BRE R1, R2, 235
...  ...
...  ...
235
```

unconditional branch (210 → 202)

conditional branch (203 → 211)

conditional branch (225 → 235)

Transfer of control (2)

- Skip (example)
  - Increment register R and skip next instruction if result is 0

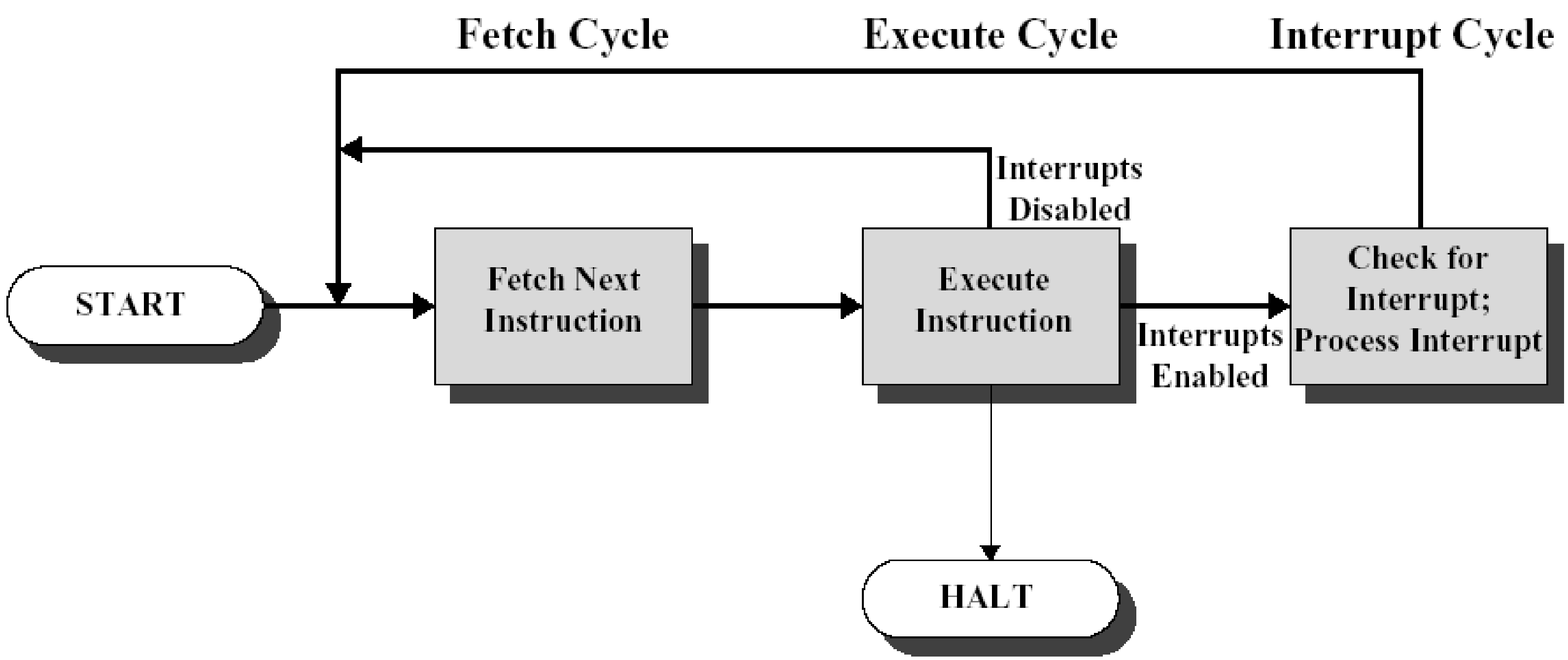
```
X:   …
     …
     ISZ R
     BRA X (loop)
     …       (exit)
```

- Interrupts (the basic form of control transfer)
- Subroutine call (a kind of interrupt serving)

Interrupts

- Mechanism by which other modules (e.g. I/O) may interrupt normal sequence of processing
- Program error
  - e.g. overflow, division by zero
- Time scheduling
  - Generated by internal processor timer
  - Used to execute operations at regular intervals
- I/O operations (usually much slower)
  - from I/O controller (end operation, error, ...)
- Hardware failure
  - e.g. memory parity error, power failure, ...

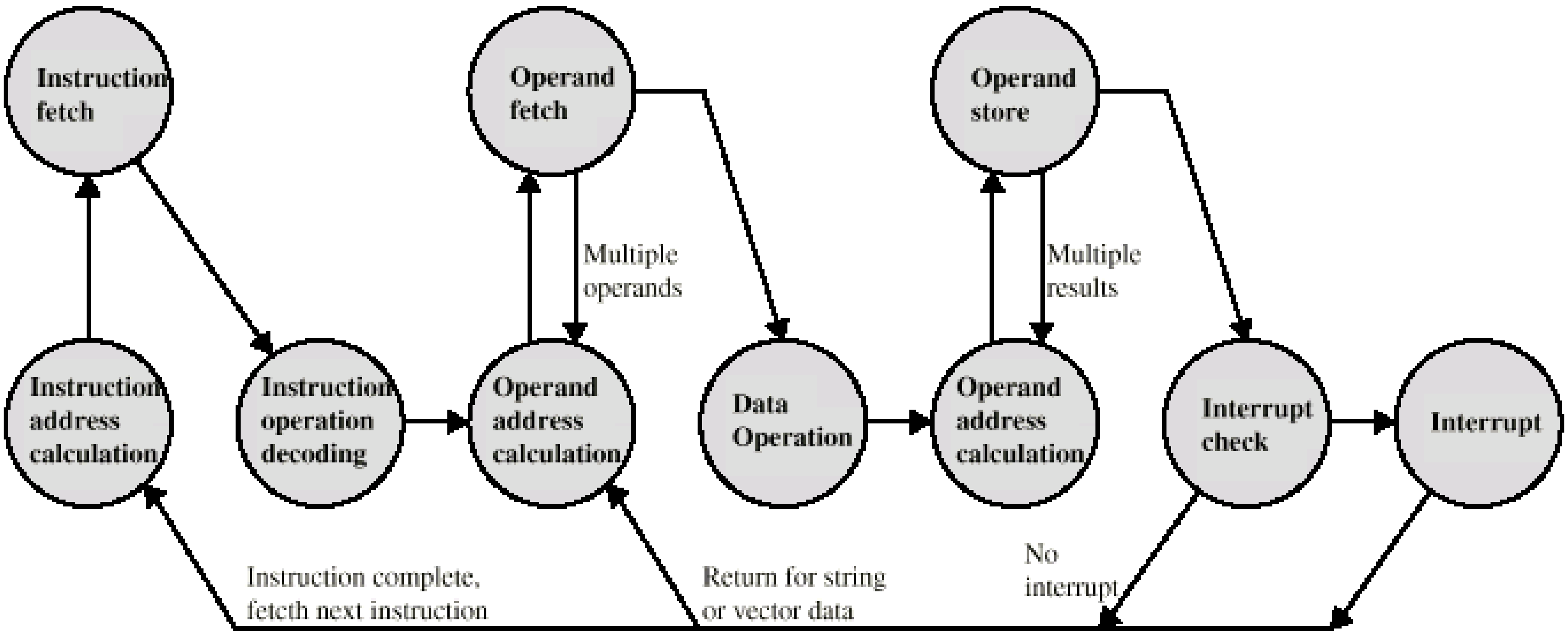
Instruction Cycle with Interrupt



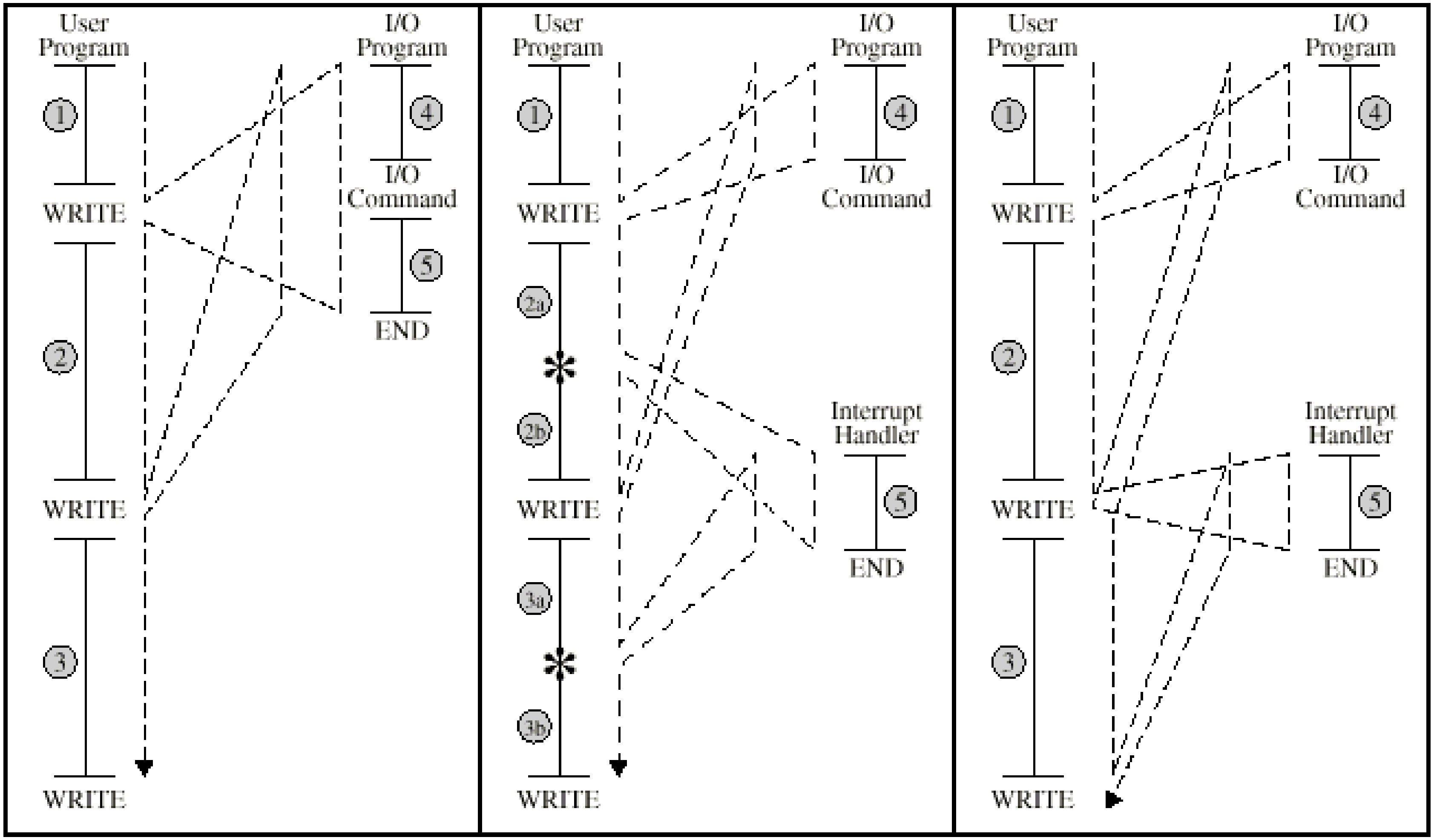
Interrupt Cycle

- Added to instruction cycle
- Processor checks for interrupt
  - Indicated by an interrupt signal
- If no interrupt, fetch next instruction
- If interrupt pending:
  - Suspend execution of current program
  - Save context
  - Set PC to start address of interrupt handler routine
  - Process interrupt
  - Restore context and continue interrupted program

Instruction Cycle (with Interrupts) - State Diagram

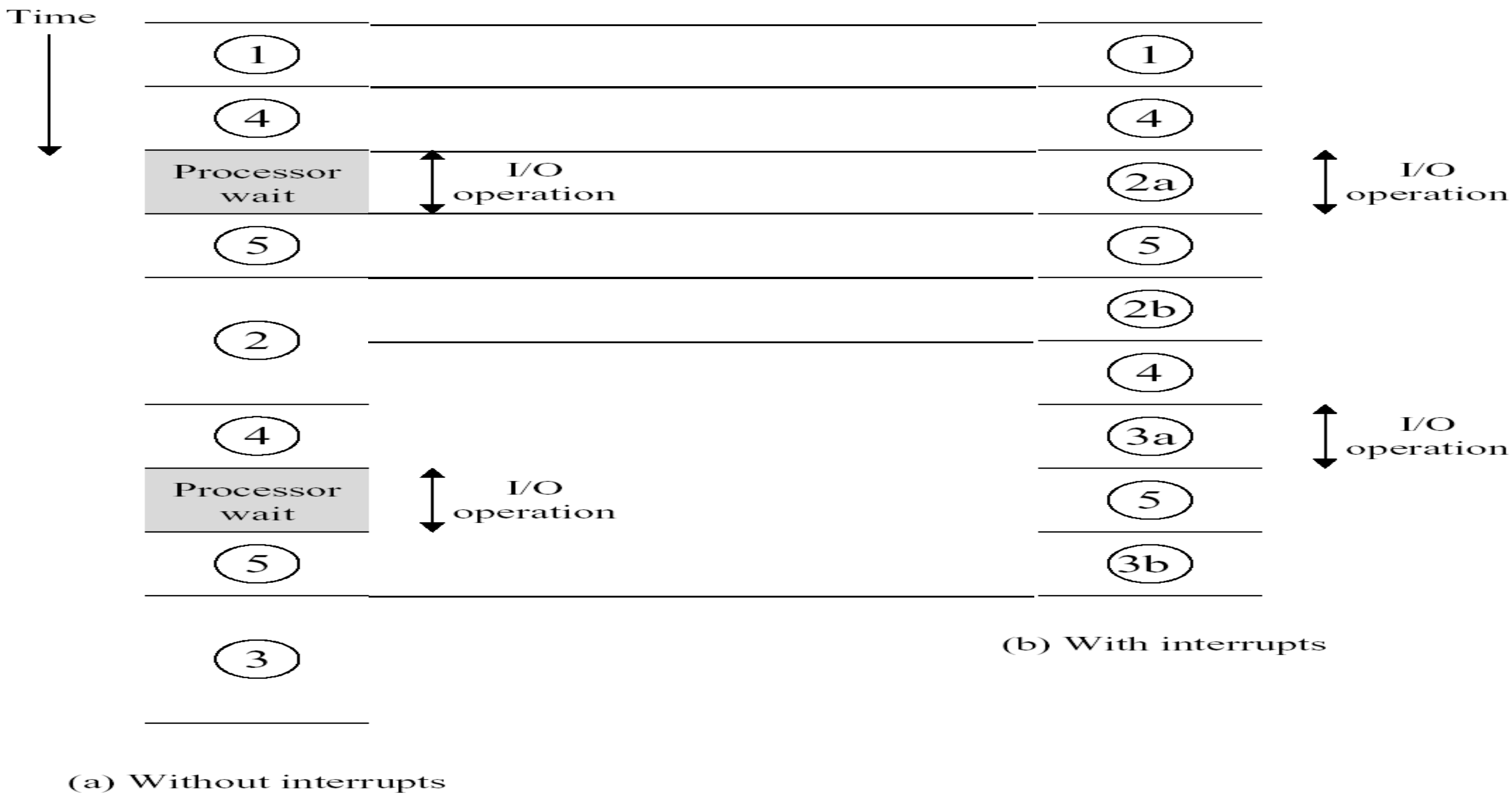


Program Flow Control

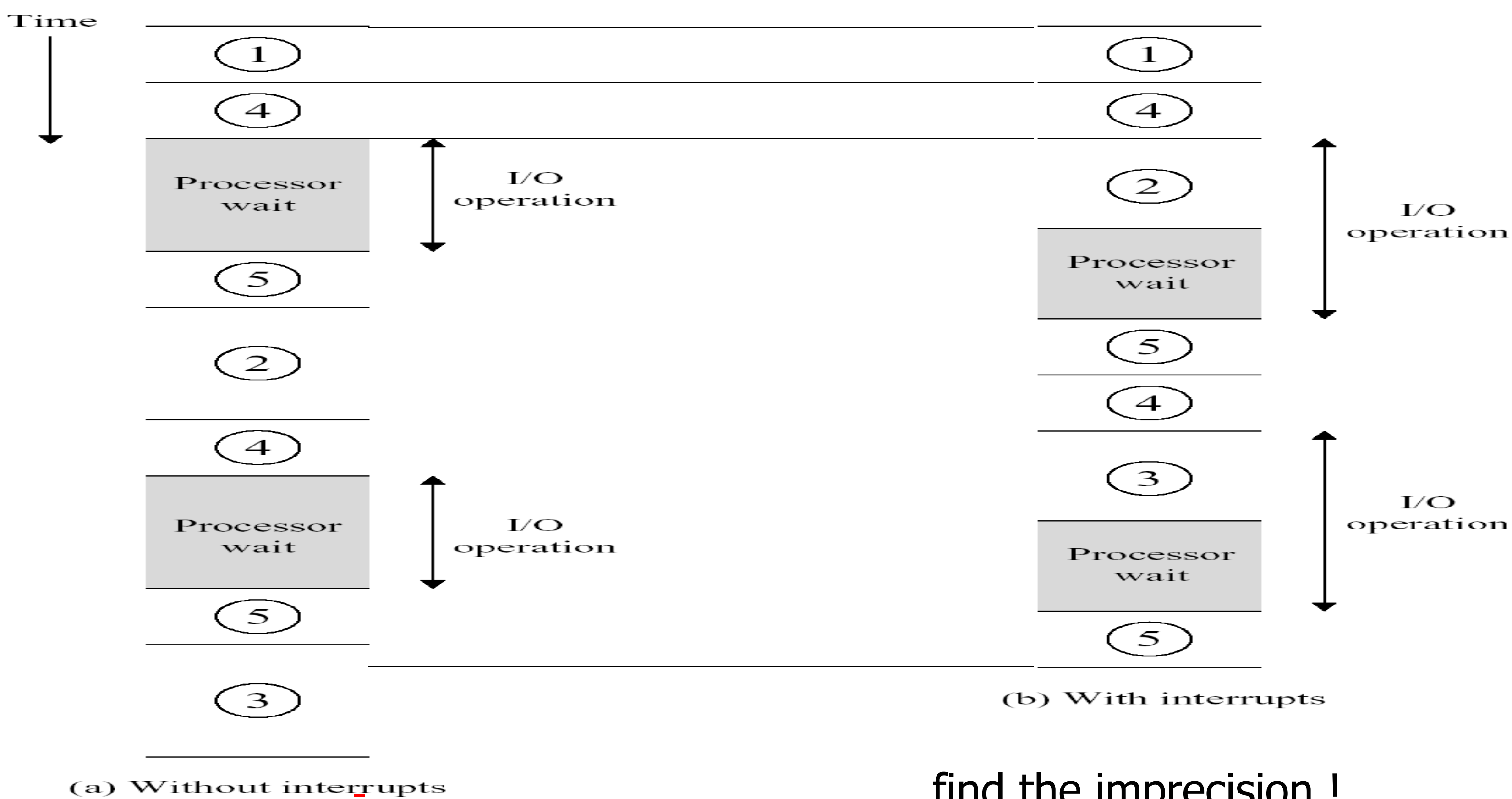


(a) No interrupts (b) Interrupts; short I/O wait (c) Interrupts; long I/O wait

Temporal view of control flow (short I/O wait)



Temporal view of control flow (long I/O wait)

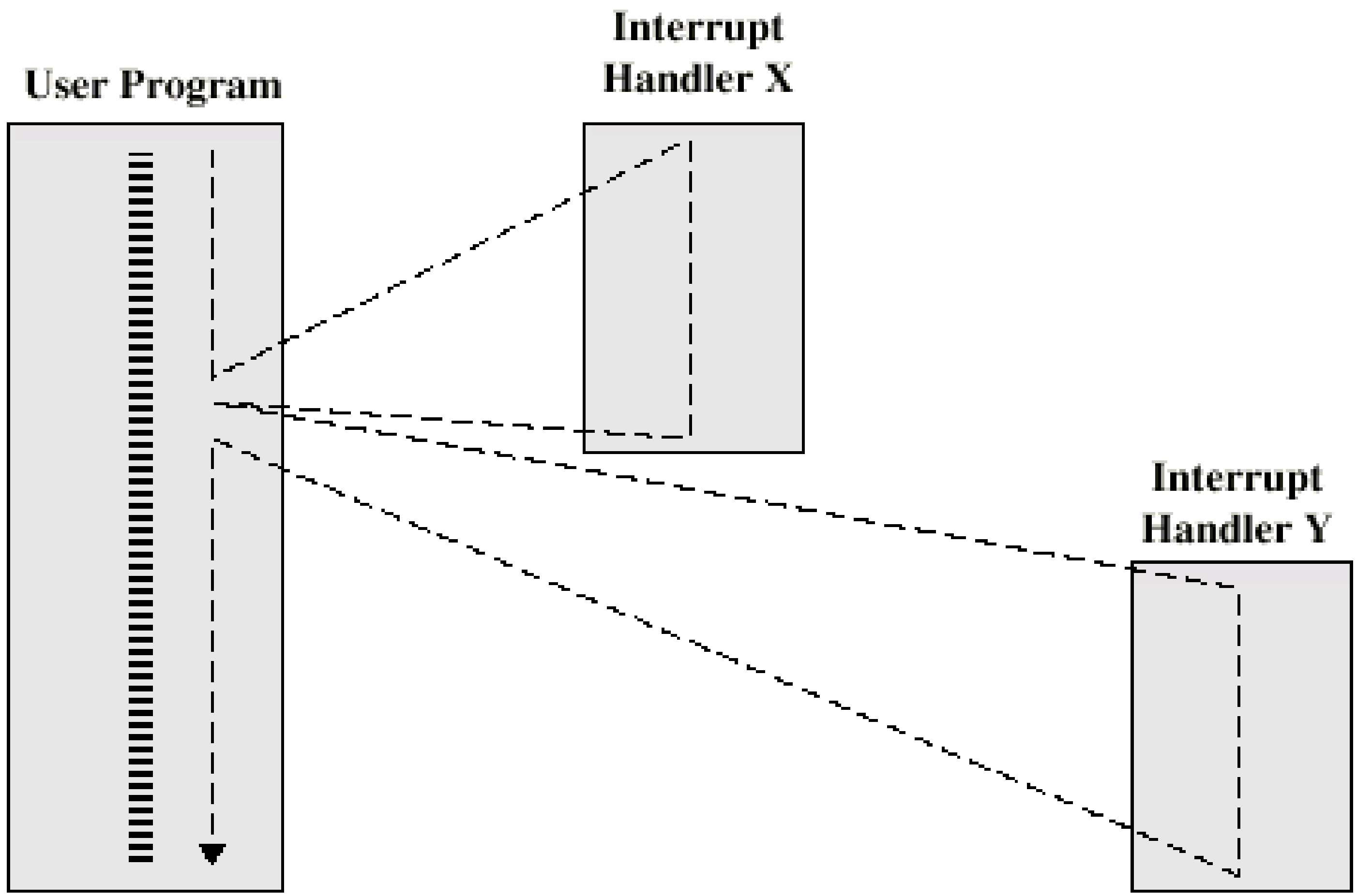


find the imprecision !

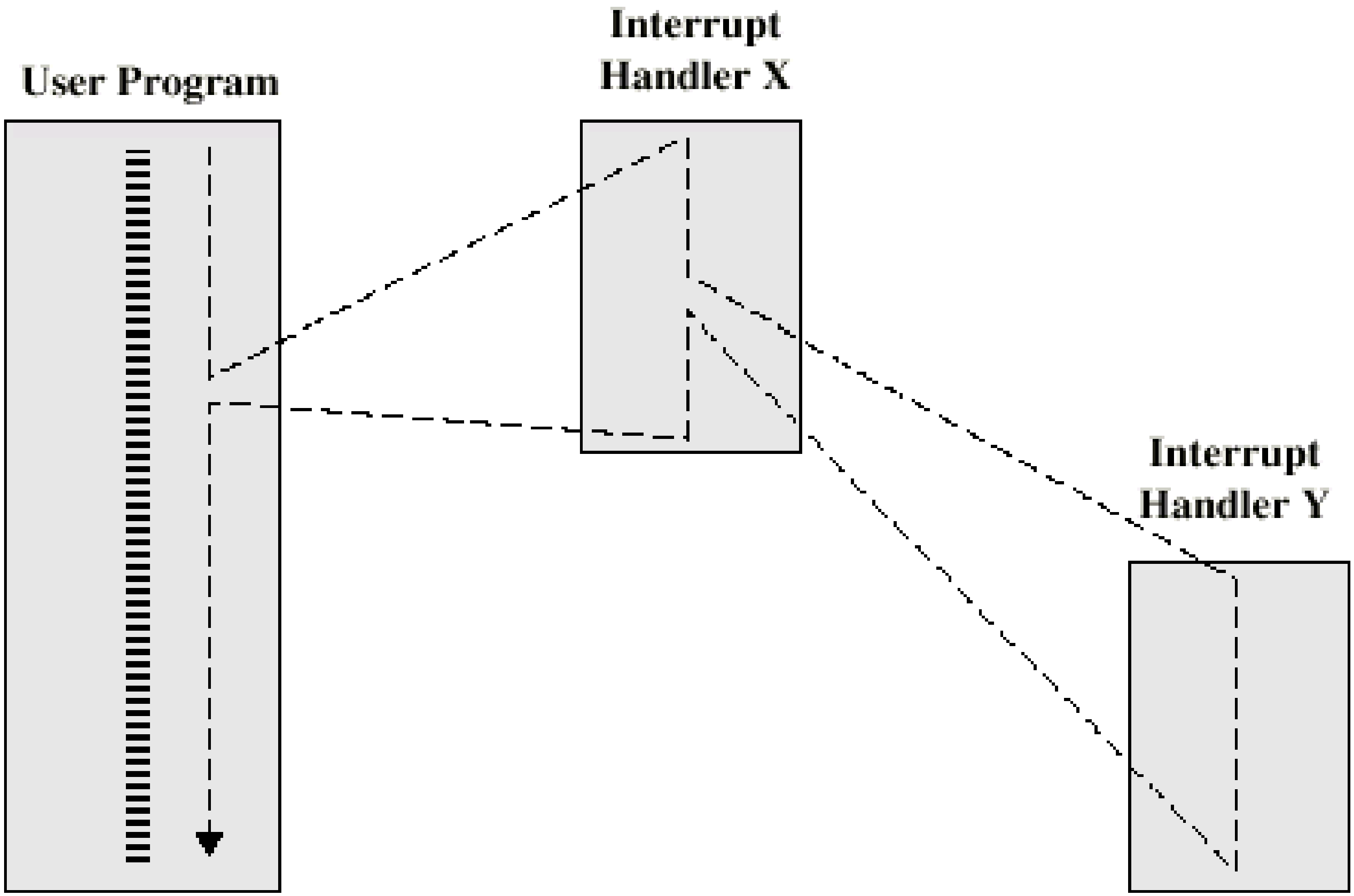
Multiple Interrupts

- 1st solution: Disable interrupts
  - Processor will ignore further interrupts whilst processing one interrupt
  - Interrupts remain pending and are checked after first interrupt has been processed
  - Interrupts handled in sequence as they occur
- 2nd solution: Define priorities
  - Low priority interrupts can be interrupted by higher priority interrupts
  - When higher priority interrupt has been processed, processor returns to previous interrupt

Multiple Interrupts - Sequential



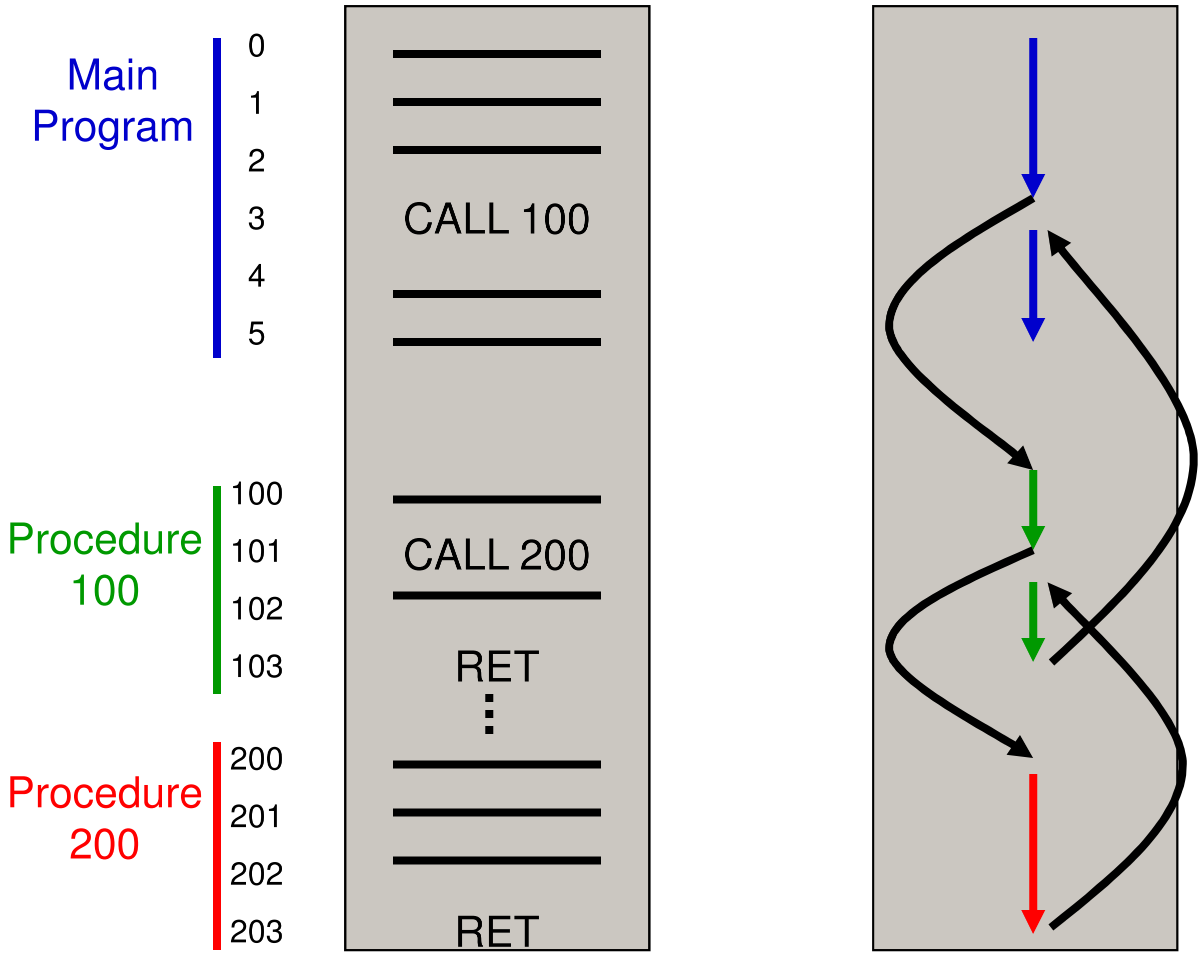
Multiple Interrupts - Nested



Subroutine (or procedure) call

- Why?
  - economy
  - modularity

Subroutine (or procedure) call



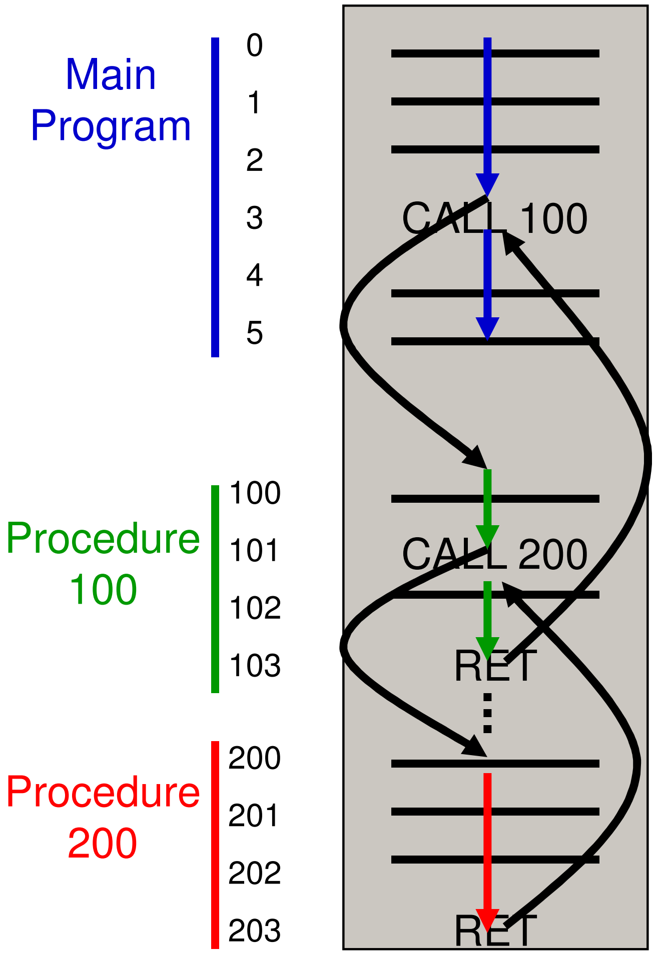
Alternative for storing the return address from a subroutine

- In a pre-specified register
  - Limit the number of nested calls since for each successive call a different register is needed
- In the first memory cell of the memory zone storing the called procedure
  - Does not allow recursive calls
- At the top of the stack (more flexible)

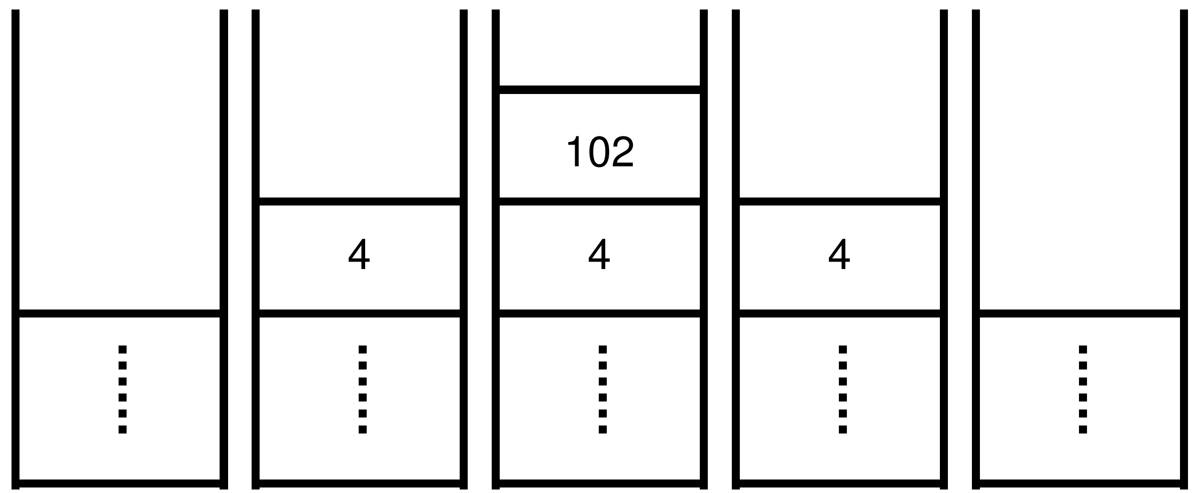
Return using the stack (1)

- Use a reserved zone of memory managed with a *stack* approach (last-in, first-out)
  - In a stack of dirty dishes the last to become dirty is the first to be cleaned
- Each time a subroutine is called, before starting it the return address is put on top of the stack
- Even in the case of multiple calls or recursive calls all return addresses keep their correct order

Return using the stack (2)



- The stack can be used also to pass parameters to the called procedure



Passing parameters to a procedure

- In general, parameters to a procedure might be passed
  - Using registers
    - Limit the number of parameters that can be passed, due to the limited number of registers in the CPU
    - Limit the number of nested calls, since each successive calls has to use a different set of registers
  - Using pre-defined zone of memory
    - Does not allow recursive calls
  - Through the stack (more flexible)

Byte Order

- What order do we read numbers that occupy more than one cell (byte)
- 12.345.678 can be stored in 4 locations of 8 bits each as it follows

Address	Value (1)	Value(2)
184	12	78
185	34	56
186	56	34
187	78	12

- i.e. read top down or bottom up ?

Byte Order Names

- The problem is called **Endian**
- The reference point is the INITIAL address of bytes
- The system on the left has the MOST significant byte in the INITIAL address
- This is called *big-endian*
- The system on the left has the LEAST significant byte in the INITIAL address
- This is called *little-endian*

Standard...What Standard?

- Pentium (80x86), VAX are little-endian
- IBM 370, Motorola 680x0 (Mac), and most RISC are big-endian
- Internet is big-endian
  - Makes writing Internet programs on PC more awkward!
  - WinSock provides *htoi* and *itoh* (Host to Internet & Internet to Host) functions to convert