Recap

Switch, LED interface
Real board debugging
if-then statements

Overview

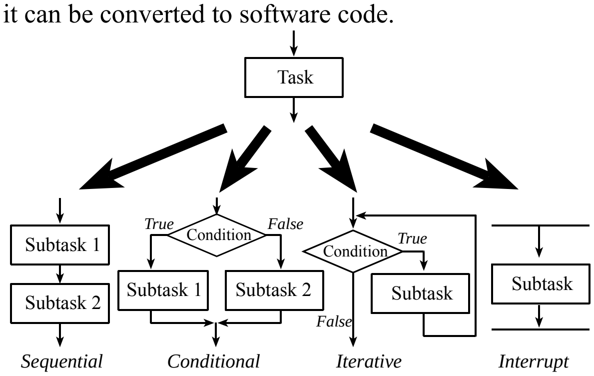
Successive refinement
Modular programming
Subroutines, parameter passing
Debugging dump

When we solve problems on the computer, we need to answer these questions:

- What does being in a state mean? List state parameters
- What is the starting state of the system? Define the initial state
- What information do we need to collect? List the input data
- What information do we need to generate? List the output data
- How do we move from one state to another? Actions we could do
- What is the desired ending state? Define the ultimate goal

Successive refinement, *stepwise refinement*, and *systematic decomposition*

- Start with a task and decompose the task into a set of simpler subtasks
- Subtasks are decomposed into even simpler sub-subtasks.
- Each subtask is simpler than the task itself.
- Make design decisions
- Subtask is so simple, it can be converted to software code.



We need to recognize these phrases that translate to four basic building blocks:

- "do A then do B" → sequential
- "do A and B in either order" → sequential (parallel)
- "if A, then do B" → conditional
- "for each A, do B" → iterative
- "do A until B" → iterative
- "repeat A over & over forever" → iterative (condition always true)
- "on external event do B" → interrupt
- "every t msec do B" → interrupt

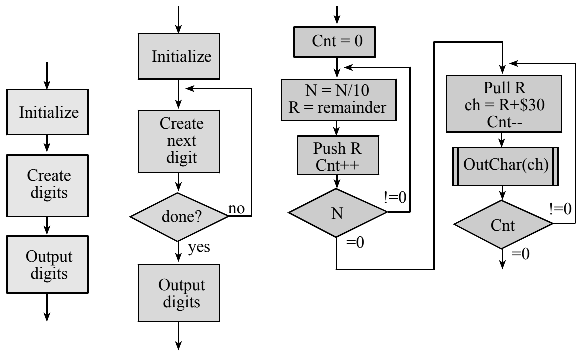


Figure 5.6. Successive refinement method for the iterative approach.

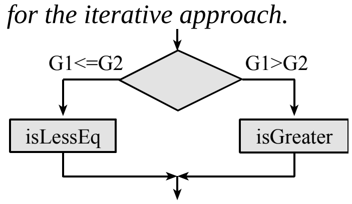


Figure 5.3. Flowchart of an if-then-else structure.

```
      ldaa G1
      cmpa G2
      bhi  high  ; branch if G1>G2
low   jsr  isLessEq   ; G1<=G2
      bra  next

high  jsr  isGreater  ; G1>G2
next
```

```
if(G1>G2){

  isGreater();
}
else{
  isLessEq();
}
```

Program 5.1. An unsigned if-then-else structure.

```
      ldaa G1
      cmpa G2
      bls  low  ; branch if G1≤G2
high  jsr  isGreater  ; G1>G2
      bra  next

low   jsr  isLessEq   ; G1<=G2
next
```

```
if(G1>G2){

  isGreater();
}
else{
  isLessEq();
}
```

Alternative unsigned if-then-else structure.

```
      while(G2 > G1){Body();}
```

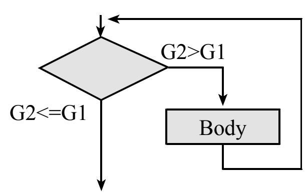


Figure 5.4. Flowchart of a while structure.

The program begins with a test of **G2>G1**. If **G2<=G1** then the body of the while loop is skipped.

```
loop ldaa G2
     cmpa G1
     bls  next  ;stop if G2≤G1
     jsr  Body  ;body of loop
     bra  loop
next
```

```
while(G2 > G1){
  Body();
}
```

Program 5.2. A while loop structure.

Question 1. Assume PT0 is an input. Draw a flowchart describing software that waits until PT0 is a 1 (loops back over and over if PT0 is a 0). Next, write it in C. Finally, write it in assembly.

5.2.5. For loops

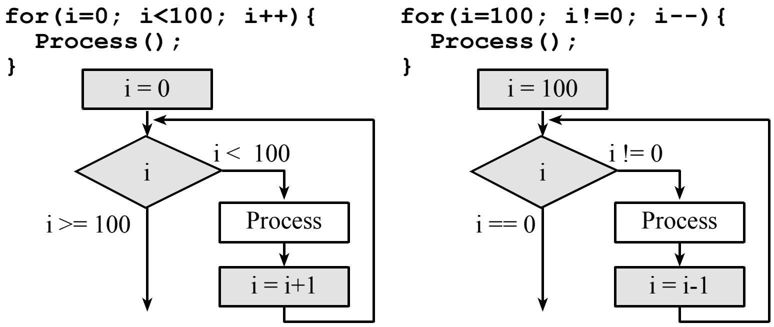


Figure 5.5. Two flowcharts of a `for-loop` structure.

The first implementation places the loop counter in the Register B, as shown in Program 5.3.

`ldab #0 ; Reg B is i, i=0`
`loop cmpb #100`
`bhs done`
`jsr Process`
`incb ; i=i+1`
`bra loop`
`done`
	`for(i=0; i<100; i++){`
	`Process();`
	`}`

Program 5.3. A simple `for`-loop.

`ldab #100 ; i=100`
`L1 jsr Process`
`dbne` `B,L1 ; i=i-1`
	`for(i=100; i!=0; i--){`
	`Process();`
	`}`

*Program 5.4. The **dbne** instruction optimizes this for-loop implementation.*

Question 2. Assume PT0 is an output. Draw a flowchart describing software that toggles PT0 1000 times (set PT0=1, then PT0=0 500 times). Next, write it in C. Finally, write it in assembly.

5.1. Modular design

Goal
- Clarity
- Create a complex system from simple parts

Definition of modularity
- Maximize number of modules
- Minimize bandwidth between them

Entry point (where to start)
- The label of the first instruction of the subroutine

Exit point (where to end)
- The **rts** instruction
- Good practice, one **rts** as the last line

Public (shared, called by other modules)
- Add underline in the name, module name before

Private (not shared, called only within this module)
- No underline in the name
- Helper functions

Coupling (amount of interaction between modules)
- Data passed from one to another (bandwidth)
- Synchronization between modules

3.3.5. Subroutines and the stack

classical definition of the stack

- push saves data on the top of the stack,
- pull removes data from the top of the stack
- stack implements last in first out (LIFO) behavior
- stack pointer (SP) points to top element

many uses of the stack

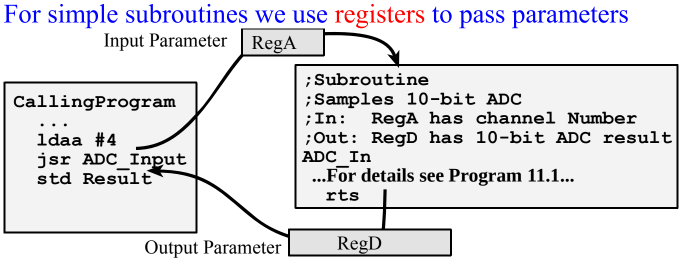
- temporary calculations
- subroutine (function) return addresses
- subroutine (function) parameters
- local variables

The push and pull instructions

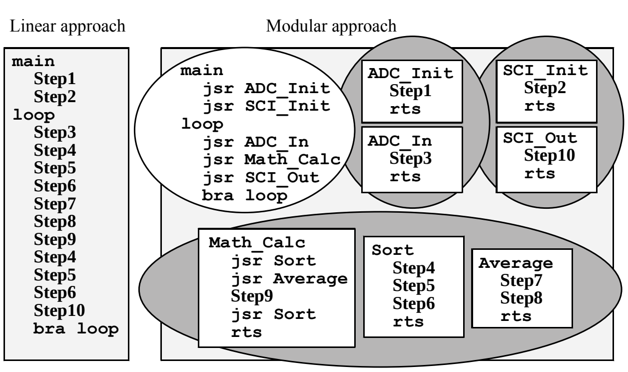
```
psha        ;push Register A on the stack
pshb        ;push Register B on the stack
pshx        ;push Register X on the stack
pshy        ;push Register Y on the stack
des                ;S =S-1 (reserve space)

pula        ;pull from stack into A
pulb        ;pull from stack into B
pulx        ;pull from stack into X
puly        ;pull from stack into Y
ins                ;S=S+1 (discard top of stack)
```

For simple subroutines we use registers to pass parameters



High level program	Subroutine
1) Sets Registers to contain inputs 2) Calls subroutine 6) Registers contain outputs	 3) Sees the inputs in registers 4) Performs the action of the subroutine 5) Places the outputs in registers



Introduction to pointers

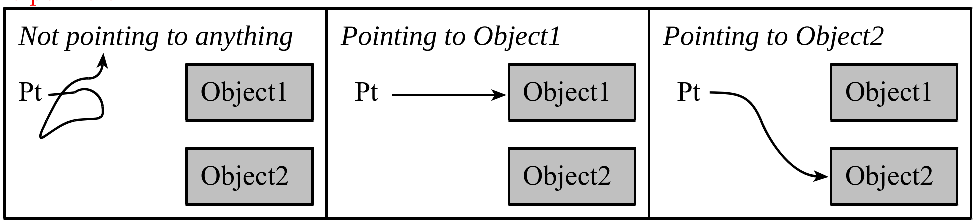
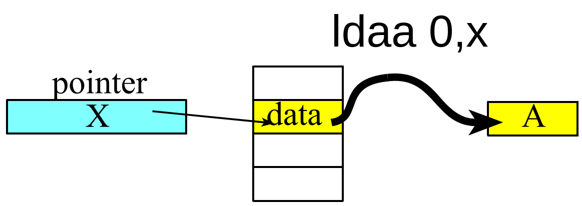


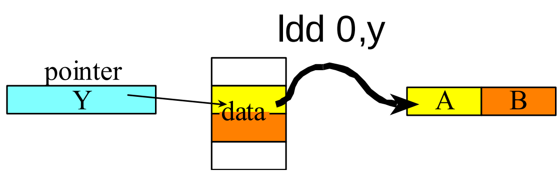
Figure 6.1. Pointers are addresses pointing to objects. The objects may be data, functions, or other pointers.

If Register X or Y contains an address, we say it points into memory

```
;read 8-bit contents pointed to by X
```




```
;read 16-bit contents pointed to by Y
```



The bottom line

Stack is used for return address, temporary storage
Subroutines provide a means for modular code
For now, we pass parameters in registers
Pointers are addresses
- **Set a pointer to point to data**
- **Read the data at that pointer**
- **Write data through the pointer**
- **Change the pointer to next element**

8-bit or 16-bit data?
Signed or unsigned numbers?