HKN ECE 220: Fall 2018 Midterm 1

Xinyi Guo, Michael Chen, Srijan Chakraborty, Siddharth Agarwal

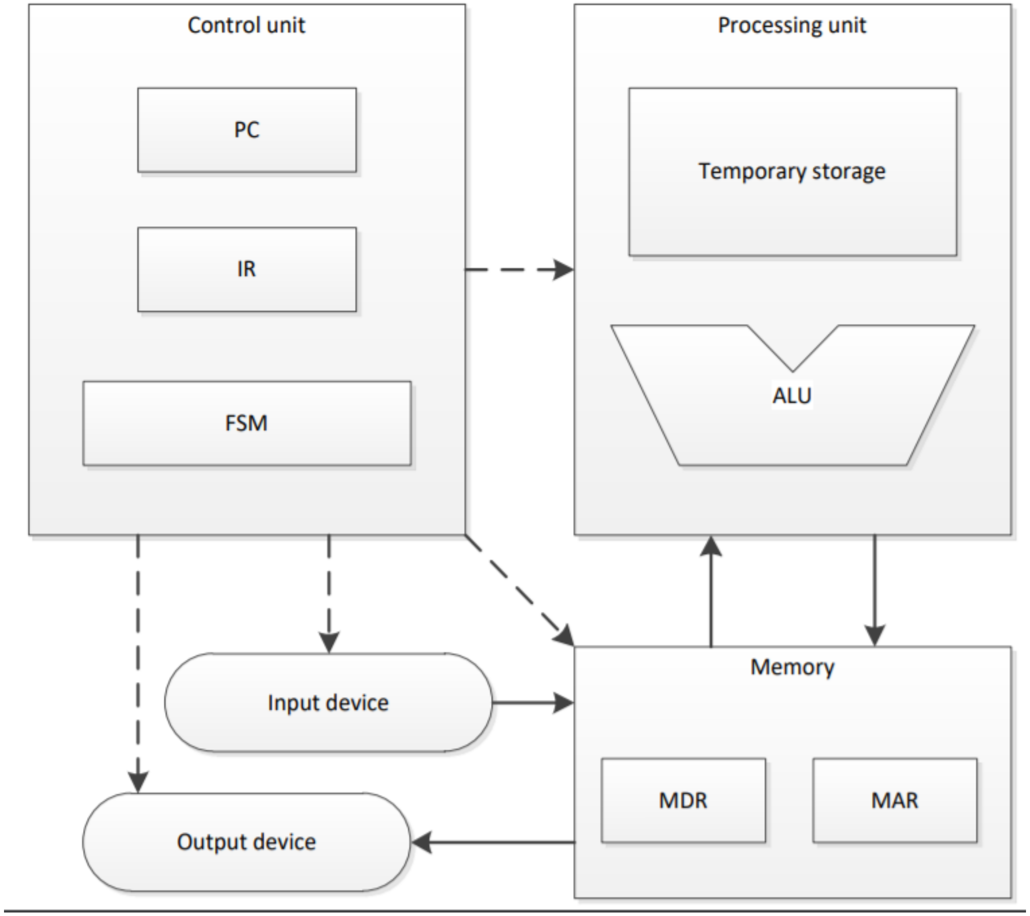
29th September, 2018

ECE ILLINOIS

ILLINOIS

LC3: A Brief Overview

- 16 Bit Data
- 16 Bit Address (coincidence)
- 8 Registers (R0-R7)
- Memory and Mem. Interface
- MAR (Accessing addresses)
- MDR (Accessing actual data)
- Input (KBSR, KBDR)
- Output (DSR, DDR)
- PC and IR
- R7 used for bookkeeping



Operations in LC3

Operations:

ADD, **AND**, **NOT**

Control:

BRnzp, **JSR** (and JSRR), JMP, **RET**, **TRAP**

(Also RTI for interrupts)

Memory Interface:

LD (LDR, LDI), **ST** (STR, STI), **LEA**

Pseudo-Ops

.ORIG x3000	the first instruction should be at x3000
.END	indicate this is the end of the program
.FILL	#-3, #5, #0, xFFC0, xABCD, etc.
.BLKW#3	number of memory locations to reserve
.STRINGZ	"Hello" (Null-terminated)
TRAP x25	same as HALT

Examples

REMEMBER!
-16 <= immediate value <= 15

- How to clear R0?
- AND R0, R0, #0

- How to do copy R1 to R0?
- ADD R0, R1, #0

- How to get –R0?
- NOT R0, R0
- ADD R0, R0, #1

- How to left shift R0?
- ADD R0, R0, R0

LC-3 Review: I/O

I/O Interactions

- Polling vs Interrupts
  - Polling
    - Loop indefinitely until data is available by checking status registers (KBSR, DSR)
  - Interrupts
    - Allows program to perform other work while no data is available
    - Upon reception of interrupt, pause current code execution and execute special interrupt handling functions
    - Return to interrupted code once interrupt has been handled
    - Will be covered in depth in ECE 391!

LC-3 Review: I/O

Memory Mapped I/O

- Map I/O to specific memory addresses
    - Removes the need for dedicated I/O channels
- Accessing the mapped memory address gives access to the input or output device
    - Reading from xFE02 (KBDR) returns a char of what key was pressed on the keyboard
    - Writing ‘a’ to xFE06 (DDR) will display ‘a’ on the display
    - Check the status register (KBSR, DSR) of the respective input/output before reading or writing

LC-3 Review: Keyboard Input

Reading from the keyboard

- Poll KBSR until ready bit is set then access input data stored in lower 8 bits of KBDR

```
POLL        LDI     R1, KBSR            ; Check status register
            BRzp    POLL                ; Loop while ready bit not set
            LDI     R0, KBDR            ; Get keyboard input

KBSR        .FILL   xFE00               ; KBSR address

KBDR        .FILL   xFE02               ; KBDR address
```

LC-3 Review: Display Output

Writing to the display

- Poll DSR until ready bit is set then write display data to DDR

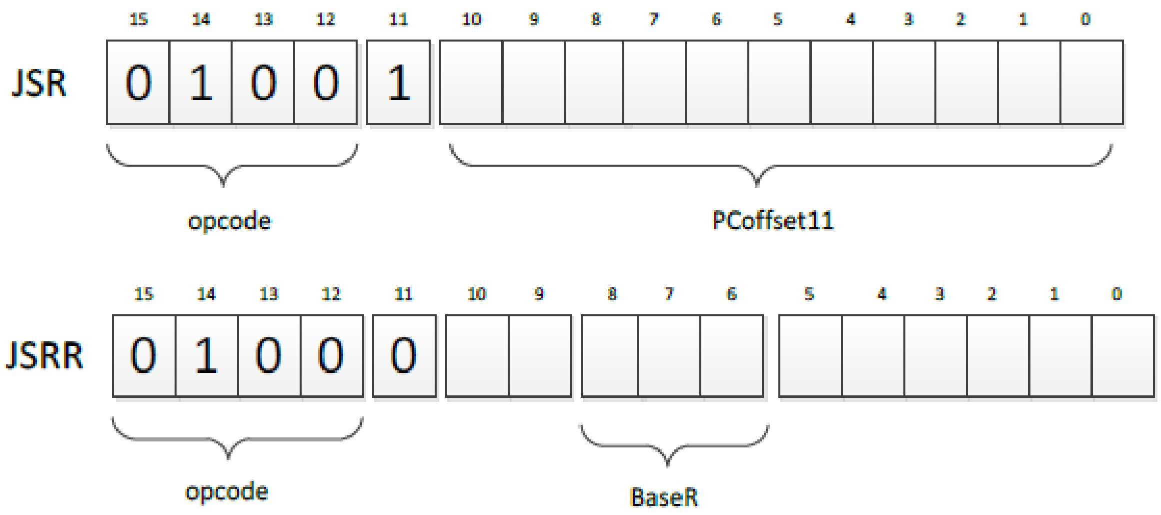
```
POLL    LDI     R1, DSR         ; Check status register
        BRzp    POLL            ; Loop while ready bit not set
        STI     R0, DDR         ; Write display data

DSR     .FILL   xFE04           ; DSR address

DDR     .FILL   xFE06           ; DDR address
```

Subroutines

- Useful if there is a code segment that needs to be executed multiple times
- Subroutines can be invoked by JSR or JSRR
- Return is implemented with RET instruction



```
TEMP <- PC

If (IR[11] == 0)
      PC <- BaseR
Else
      PC <- PC + SEXT(PCoffset11)

R7 <- TEMP
```

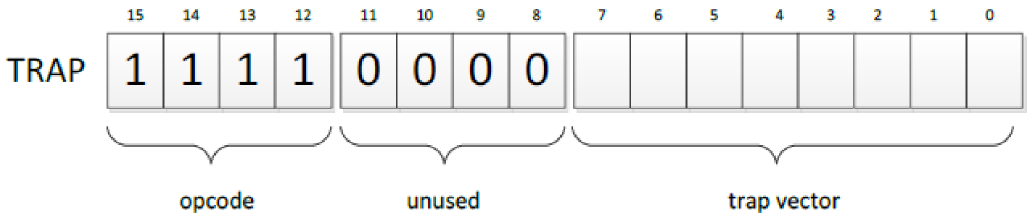
Subroutines: Callee and Caller Save

- Subroutine will save and restore registers that it modifies except for the return values
  - The only visible change should be the return value (if any) upon return
- Caller should save registers that could be modified by the subroutine if they contain important data
  - R7 would need to be saved since JSR and JSRR overwrite its value

```
; Caller-save user program
…
ST R0, SaveR0          ; store R0 in memory
ST R7, SaveR7          ; store R7 in memory
GETC                   ; call TRAP which
                       ; destroys R0 and R7
LD R7, SaveR7          ; restore R7
…                      ; consume input in R0
LD R0, SaveR0          ; restore R0
…
HALT

SaveR0 .BLKW 1
SaveR7 .BLKW 1
```

TRAPS



	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TRAP	1	1	1	1	0	0	0	0								
	opcode				unused				trap vector							

TRAP function

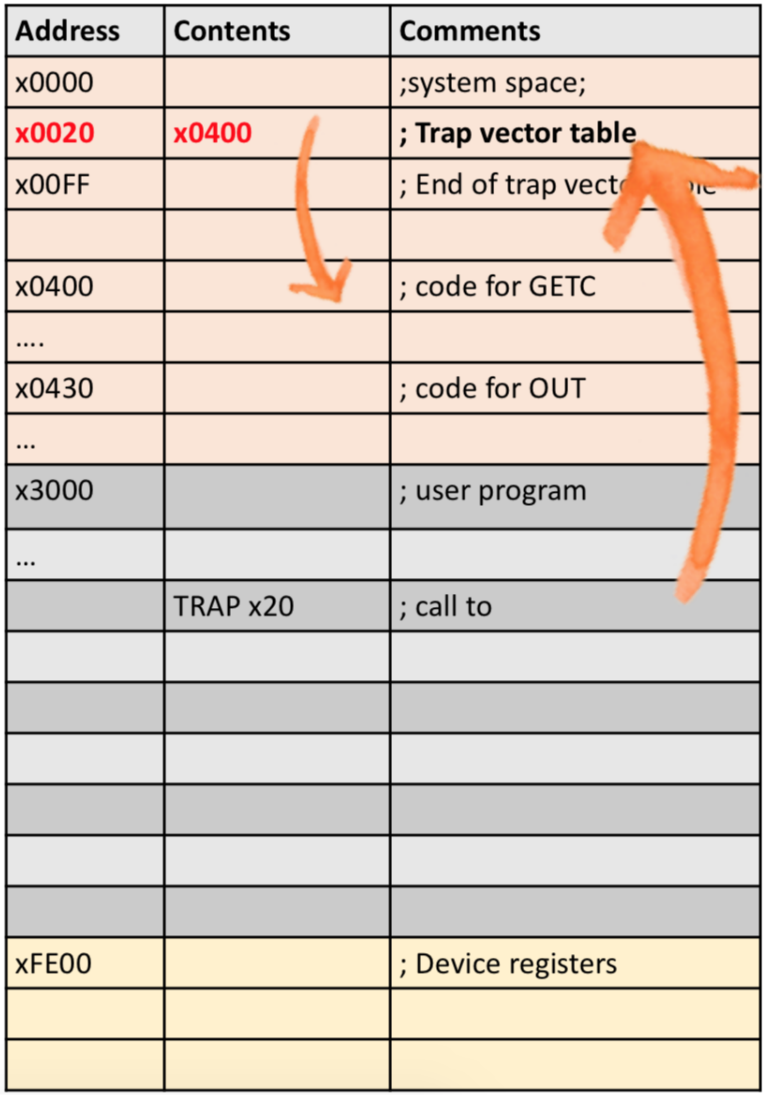
- Passes control to operating system
- Programmers can use complex operations without specialized knowledge

Trap Vector	Assembler Name	Description
x20	GETC	Read single character from keyboard into R0
x21	OUT	Write character from R0 to display
x22	PUTS	Write null terminated string of characters to display starting from memory location at R0
x23	IN	Prompts for input; Reading char from keyboard and echo input to console
x24	PUTSP	Same as puts but use characters from both lower and upper 8 bits
x25	HALT	Halts program execution

TRAPS: How they work

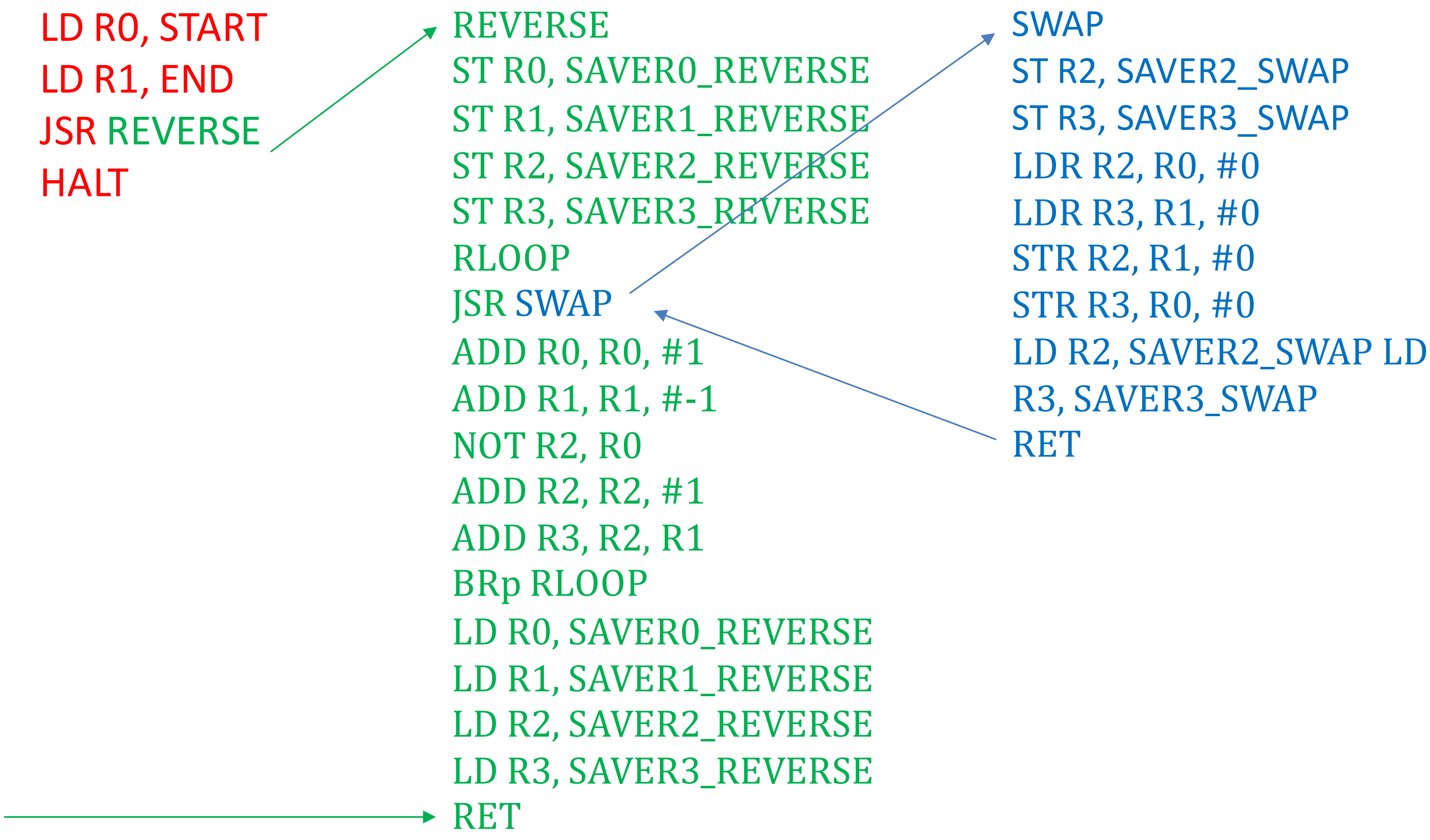
- TRAP function is called by the user
- The 8-bit trap vector is used as the index of the service routine's address in the trap vector table
- The PC is loaded with the address of the service routine
- After executing the service routine, control returns to the user program

```
MAR <- ZEXT(trapvector)
MDR <- MEM[MAR]
R7 <- PC
PC <- MDR
```



Address	Contents	Comments
x0000		;system space;
x0020	x0400	; Trap vector table
x00FF		; End of trap vector table
x0400		; code for GETC
….		
x0430		; code for OUT
…		
x3000		; user program
…		
	TRAP x20	; call to
xFE00		; Device registers

Problem with nested calls



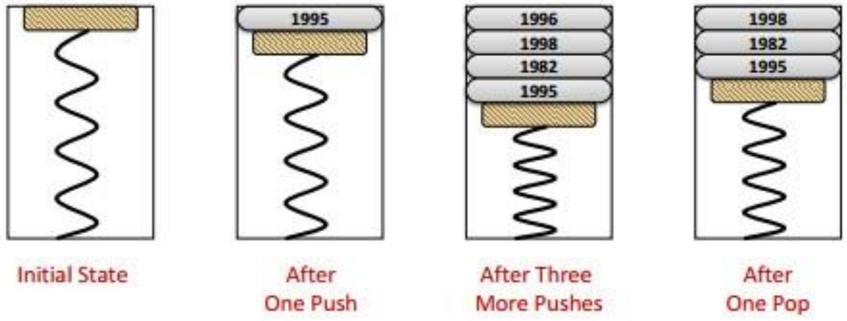
```
LD R0, START
LD R1, END
JSR REVERSE
HALT
```

```
REVERSE
ST R0, SAVER0_REVERSE
ST R1, SAVER1_REVERSE
ST R2, SAVER2_REVERSE
ST R3, SAVER3_REVERSE
RLOOP
JSR SWAP
ADD R0, R0, #1
ADD R1, R1, #-1
NOT R2, R0
ADD R2, R2, #1
ADD R3, R2, R1
BRp RLOOP
LD R0, SAVER0_REVERSE
LD R1, SAVER1_REVERSE
LD R2, SAVER2_REVERSE
LD R3, SAVER3_REVERSE
RET
```

```
SWAP
ST R2, SAVER2_SWAP
ST R3, SAVER3_SWAP
LDR R2, R0, #0
LDR R3, R1, #0
STR R2, R1, #0
STR R3, R0, #0
LD R2, SAVER2_SWAP LD
R3, SAVER3_SWAP
RET
```

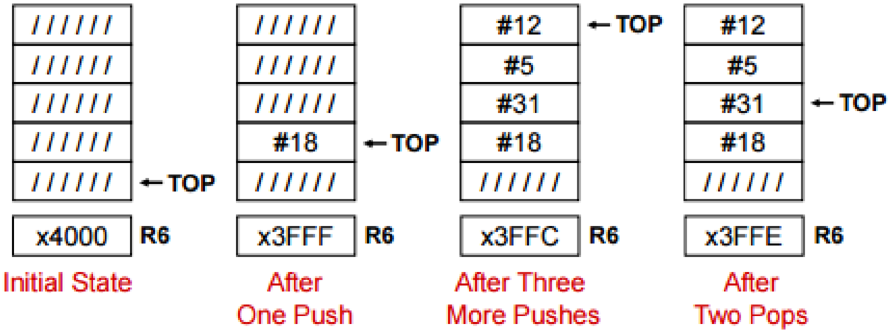
Stacks

- Last-In-First-Out (LIFO)
- Stack operations
  - Push: puts a new thing on top of the stack
  - Pop: removes whatever is on the top of the stack
  - IsEmpty: checks if the stack is empty
  - IsFull: checks if the stack is full
- Example:



Stacks(continued)

- Implementation
  - Keep elements stationary, just move the pointer
  - More efficient than moving everything



- Example: Calculator
- Questions?

Control Structure in C

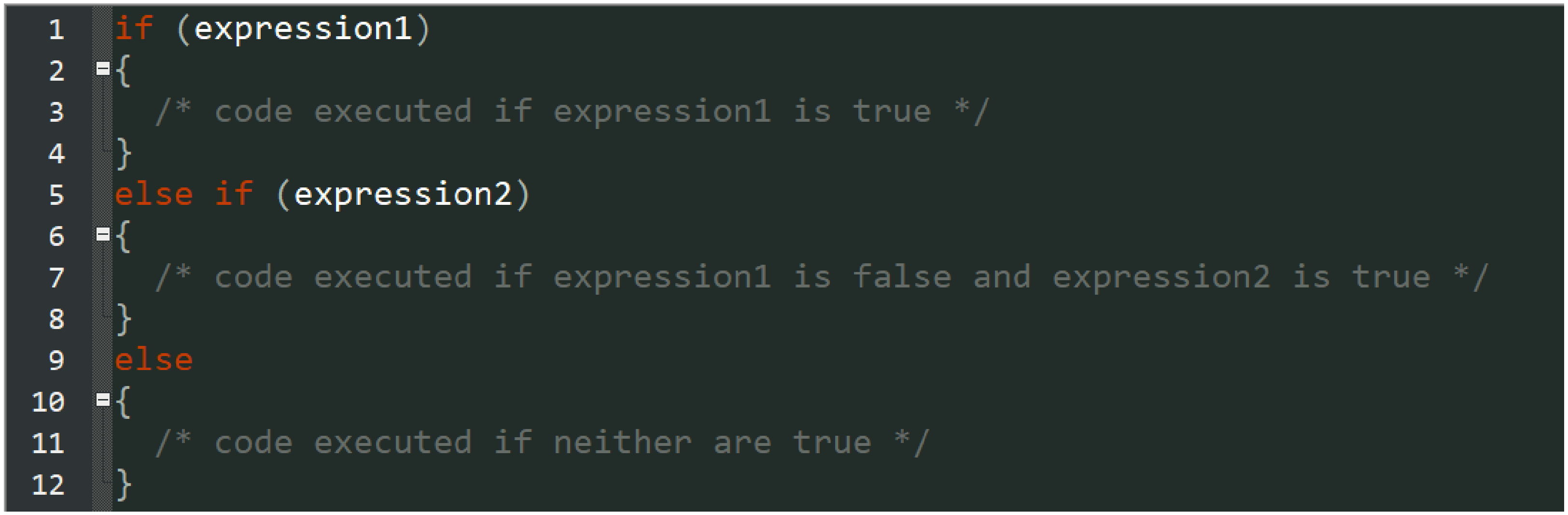
Conditional construct:

- -if
- -if – else
- -switch

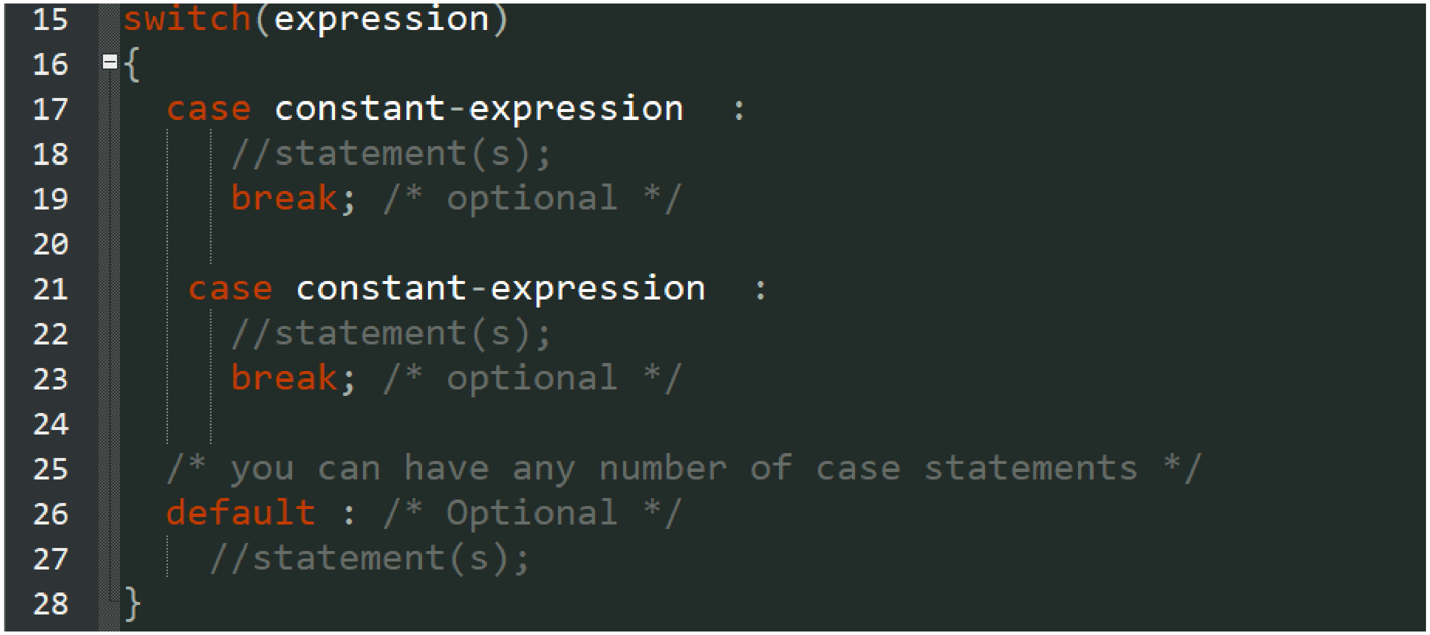
Iterative constructs (loop):

- -while
- -do while
- -for

Conditional Constructs



```
if (expression1)
{
  /* code executed if expression1 is true */
}
else if (expression2)
{
  /* code executed if expression1 is false and expression2 is true */
}
else
{
  /* code executed if neither are true */
}
```

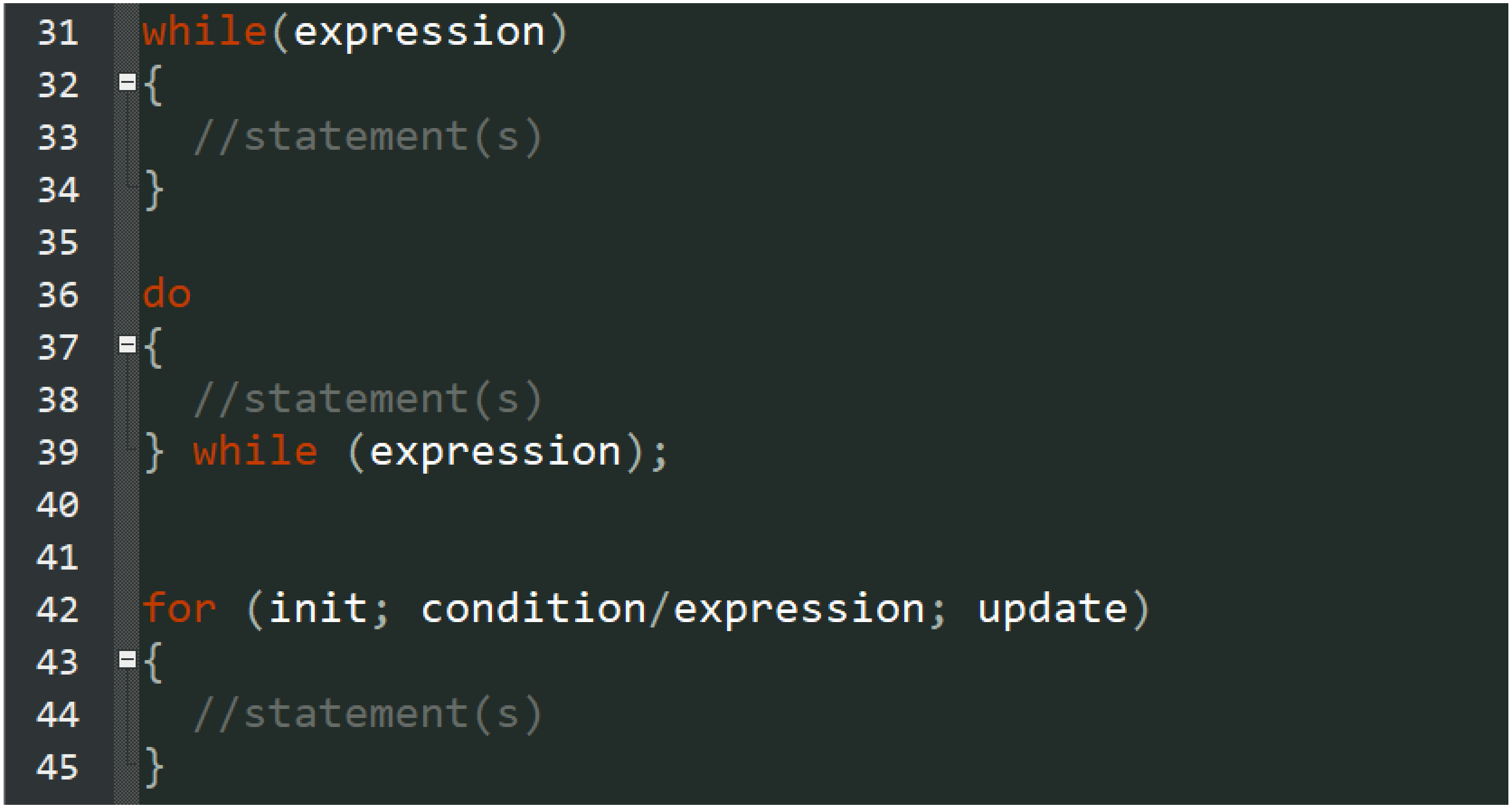


```
switch(expression)
{
  case constant-expression  :
     //statement(s);
     break; /* optional */

   case constant-expression  :
     //statement(s);
     break; /* optional */

  /* you can have any number of case statements */
  default : /* Optional */
    //statement(s);
}
```

Iterative Constructs



```
while(expression)
{
  //statement(s)
}

do
{
  //statement(s)
} while (expression);


for (init; condition/expression; update)
{
  //statement(s)
}
```

Practice Questions

Assuming 3 items have been pushed onto the stack. After a POP operation, will the last item pushed onto the stack be erased from memory? Explain.

Is polling I/O is more efficient than interrupt-driven I/O? Explain.

Explain what is a stack underflow.

The input stream of a stack is a list of all the elements we pushed onto the stack, in the order that we pushed them. If the input stream is ZYXWVUTSR, create a sequence of pushes and pops such that the output stream is YXVUWZSRT.

How many instructions, in terms of SOME_NUMBER, are run in this program?

```
        LD R0, OP1
        LD R2, OP2
        ADD R1, R0, #0

TOP
        ADD R2, R2, R0
        ADD R1, R1, #-1
        BRp TOP

        HALT

OP1
        .FILL #SOME_NUMBER
OP2
        .FILL #10
```

Tips

- .asm (PASS 1) : a symbol table is created (PASS2): .obj (the excutable)
- Use LABELS
- Use semicolon to comment
- BR = BRnzp
- Draw a flow chart if necessary
- Try to remember what kind of numbers are in the registers that you are using. Write them down when calculation gets complicated.
- Assign different registers to specific functionality when the task is complex (R1 for row count, R2 for column count, etc)
- Make **register table.** It’s extremely useful.
- R7 should not be changed. Ever!!!
- Don’t get frustrated, breathe and start over.

GOOD LUCK!

HKN offers peer-to-peer tutoring if you need any help, just go to this website and email/contact any of us: https://hkn.illinois.edu/service/

All slides posted on HKN website

You can do it!