TOPICS COVERED

- C to LC-3
- Privilege
- Interrupts
- Scope
- Storage class
- I/O

```
FOO                 ; topic: C to LC-3 ; code from p18 of old problems
ADD R6,R6,#-5 ; three for linkage, two for local vars
STR R5,R6,#2        ; save caller's frame pointer
ADD R5,R6,#1        ; our frame pointer (FOO's)
STR R7,R5,#2        ; save return address
; stack frame is now ready for executing code
; here is where we will write the C code from foo
TEARDOWN_STACK_FRAME
LDR R7, R5, #2      ; restore return address to R7
LDR R5, R5,#1       ; restore caller's frame pointer
ADD R6,R6,#4        ; why 4?  Need to leave return value on stack
RET
```

```
; Z is at R5+0, VAL is at R5 – 1, X is at R5 + 4, Y is at R5+5
LDR R0,R5,#4 ; copy X to R0
STR R0,R5,#-1          ; store into VAL (VAL = X)
LDR R0,R5,#4 ; copy X to R0 (if 0 > x, return -1)
BRn RETURN_N1      ; X <0, so skip return -1
LDR R0,R5,#5 ; copy Y to R0 (or if 0 > y, return -1)
BRzp NO_RET_N1
RETURN_N1
AND R0,R0,#0           ; put -1 in R0
ADD R0,R0,#-1
STR R0,R5,#3 ; store -1 in return value location
BRnzp TEARDOWN_STACK_FRAME
NO_RET_N1              ; x >= 0 and y >=0 … proceed to next if
```

```
; Z is at R5+0, VAL is at R5 – 1, X is at R5 + 4, Y is at R5+5
; x > y ??   x – y > 0
LDR R0,R5,#4        ; copy X into R0
LDR R1,R5,#5        ; copy Y into R1
NOT R1,R1           ; negate Y
ADD R1,R1,#1
ADD R0,R0,R1        ; calculate X – Y
BRnz NOT_THEN    ; x <= y, so do not execute THEN code
; DIVIDE … DIVIDE R0 by R1, return quotient in R0
; MULT … MULT R0 by R1, return product in R0
```

```
; Z is at R5+0, VAL is at R5 – 1, X is at R5 + 4, Y is at R5+5
; DIVIDE … DIVIDE R0 by R1, return quotient in R0
; MULT … MULT R0 by R1, return product in R0
; z = x / y
LDR R0,R5,#4        ; copy X into R0
LDR R1,R5,#5        ; copy Y into R1
JSR DIVIDE          ; R0 = R0 / R1
STR R0,R5,#0
```

```
; Z is at R5+0, VAL is at R5 – 1, X is at R5 + 4, Y is at R5+5
; DIVIDE … DIVIDE R0 by R1, return quotient in R0
; MULT … MULT R0 by R1, return product in R0
; val = x – y * z;
LDR R0,R5,#5     ; copy Y into R0
LDR R1,R5,#0     ; copy Z into R1
JSR MULT                 ; R0 = R0 * R1
NOT R0,R0                ; negate Y * Z
ADD R0,R0,#1
LDR R1,R5,#4     ; copy X into R1
ADD R1,R1,R0     ; calculate X – Y * Z into R1
STR R1,R5,#-1
NOT_THEN                 ; now we are going to return val
LDR R0,R5,#-1            ; copy VAL into R0
STR R0,R5,#3             ; store val into return value slot
… back to TEARDOWN_STACK_FRAME
```

Most ISAs have two levels privilege

1. privileged: can do anything, touch any I/O registers, execute all possible instructions, and so forth; this is for the operating system
2. Unprivileged: for user programs; not allowed to touch other programs' memory, other users' memory, I/O registers, the operating system memory or code, certain instructions (like return from interrupt)

How does unprivileged code do I/O? Using OS subroutines, such as GETC, OUT, PUTS

Interrupts…

Devices (and especially) are slow compared to processors

When you interact with devices

- Polling (reading the status register repeatedly) is a waste of time
- Better to do other work if you have other work
- But need to be responsive to devices

Interrupts are like doorbells

When a device raises an interrupt, processor immediately pays attention

In the state machine for LC-3, for example, the first fetch stage goes off to handle an incoming interrupt

Handling an interrupt requires

(1) Saving all state of the processor (the code executing doesn't expect the interrupt), including condition codes

(2) Execute an interrupt handler (a subroutine). In LC-3, there's an interrupt vector table x0100 to x01FF, save return address to stack (do not overwrite R7), then PC <- M[0x100 + ZEXT(int-vec-8)]

(3) Subroutine (interrupt handler) ends in RTI, which restores state including condition codes

In LC-3, both device status registers are extended to use bit 14 to request interrupts

In other words, write a 1 to bit 14 of DSR to turn on interrupts from the display (when the display becomes ready for a new character)

And write a 1 bit to bit 14 of KBSR to turn on keyboard interrupts (when the human types a key)

Stores to these registers do NOT overwrite bit 15.

Scope…

Scope of a variable is the part of a program in which the variable can be accessed by name

In C, we have three scopes

Local scope – the most closely enclosing compound statement

One exception: for (int X = 0; X < 10; X++) { X is scoped here and in () }

File scope –the file in which the variable is defined

static int x; // declared outside of any function

Global scope – available in any file

Storage class … somewhat tied to scope in C

Automatic storage (on the stack—in the stack frame of a function)

1. Local variables go here (unless you move them to static storage by writing ‘static’ before the declaration).
2. Created during stack frame setup (function starts)
3. Filled with bits unless you initialize
4. Destroyed when stack frame torn down (function finished)
5. One copy per function execution (may be >1 at once, even)

Static storage (in the global data area—also stored on disk as part of executable)

1. Initialized to 0 by default (but don’t use that fact)
2. Exists for the lifetime of the program.
3. Exactly one copy (variable has one value; if you change it, it stays changed)

Dynamic storage (in the heap)

1. Created on demand.
2. Destroyed on demand.
3. No name.

Parameters are automatic storage

They are created by a caller just before calling the function, but they are part of the stack frame of the function.

They are discarded just after the function finishes.

The compiler can use the stack to save (and restore) any information that it needs to save/restore, such as…

Callee-saved registers at the start and end of the function (none in LC-3)

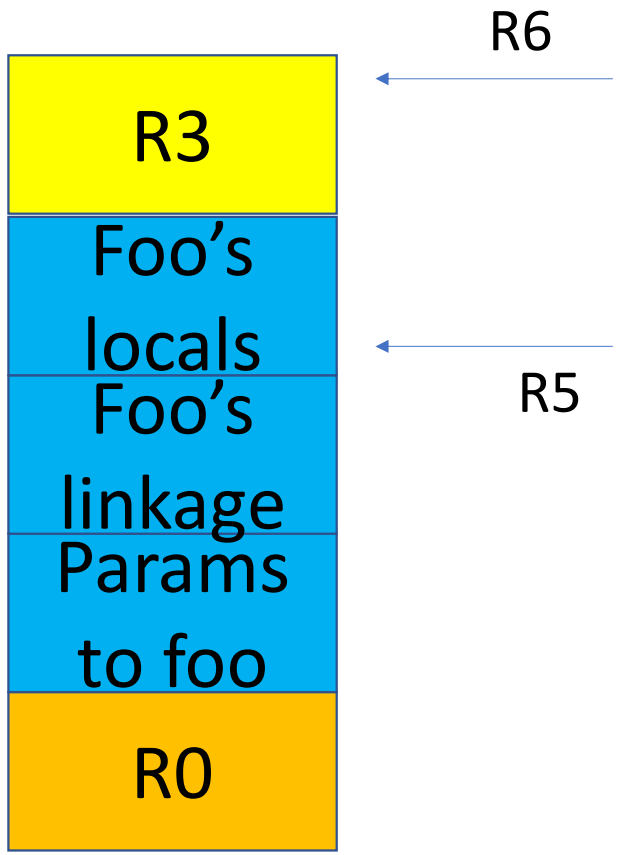
Caller-saved registers before doing a function call.

For example, if compiler had an important value in R0 but needed to call another subroutine, it would

1. Push R0 onto stack
2. Push parameters onto stack
3. Call subroutine
4. Read return value
5. Pop parameters and return value (destroy the params)
6. Pop R0 off of stack

1. Push R0 onto stack (R0 is caller-saved; that's why the compiler needs to save it)
2. Push parameters onto stack
3. Call subroutine foo
4. Read return value
5. Pop parameters and return value (destroy the params)
6. Pop R0 off of stack

7. PRETEND THAT R3 is CALLEE SAVED, BUT foo needs to use it
8. At start of foo (between setting up stack frame and executing code), push R3
9. Just before tearing down stack frame (after executing code), pop R3



I/O

LC-3 I/O is memory-mapped

- Use load and store instructions to access I/O registers
- Give up some memory addresses for this purpose
- xFE00 to xFFFF are reserved for I/O registers in LC-3

Two devices available for LC-3

Keyboard and display

Without a shared clock, need synchronization to exchange data

For each producer-consumer pair

Each side sends a one-bit SYNC signal to the other

Let's say both start at 0

The producer follows this protocol:

- Put data on the data wires
- Flip the SYNC bit from producer to consumer
- Consumer can tell that the two SYNC bits are now different
- Consumer safely (DATA wires are held constant) reads the data
- After it's done, it tells the producer that it's done by flipping its SYNC bit
- (so the SYNC values both change to 1, then both change to 0, and so forth, over time)

In the book, they only talk about the XOR of the SYNC bits

This is the “status” bit – means that producer has put data
(or that the consumer is ready for more data, in case of display)

Protocol at processor level:

Check whether device is ready

Send/receive data to/from device

DSR / KBSR bit 15 is the status bit

There so you can read from the register and then use the negative condition code (N)

KBDR and DDR basically use ASCII (8-bit extended ASCII)

Everything is a 16-bit register, so 8 bits are not used

Write 8 bits (one character) to DDR to send it to the monitor
(after waiting for DSR status to be 1)

Read 8 bits (one character) from KBSR to read from the keyboard (after waiting for KBSR status to be 1)

Use STI/LDI … put the I/O register address into a .FILL with a label and use the label name with LDI/STI