Finally, last concept at Machine Level... The Stack: An abstract data type (ADT)

- An important abstraction you will encounter in many applications
- Abstract Data Type
  - Defined by behavior not implementation
- Stack
  - Last in/First Out
  - Many ways to implement
  - Many uses in CS
    - Interrupt drive I/O, Saving state of Processor, function calls, etc.
    - Push/Pop

Stack: An Abstract Data Type

- An important abstraction that you will encounter in many applications.
- Example: three uses
  - Interrupt-Driven I/O
    - The rest of the story...
  - Evaluating arithmetic expressions
    - Store intermediate results on stack instead of in registers
    - Do this in HW 5
  - Data type conversion
    - 2's comp binary to ASCII strings
    - You will read this on your own and use it in HW 5 and Project 2

Stacks

- Simplest form of storage?
- A LIFO (last-in-first-out) storage structure.
  - The first thing you put in is the last thing you take out.
  - The last thing you put in is the first thing you take out.

- This means of access is what defines a stack, not the specific implementation.
- Two main operations:
  - PUSH: add an item to the stack
  - POP: remove an item from the stack
A Physical Stack

- Coin rest in the arm of an automobile
- First quarter out is the last quarter in.

Initial State After One Push After Three More Pushes After One Pop

A Hardware Implementation

- Data items move between registers

Stack: What’s your model?

What do you think?

- Implement a stack as an array.
  - Push will involve moving each element over 1:
    - a(n+1) = a(n)
    - a(0) = new_element
    - Etc.
A Software Implementation

- Data items don't move in memory, just our idea about where the TOP of the stack is.

<table>
<thead>
<tr>
<th>Initial State</th>
<th>After One Push</th>
<th>After Three More Pushes</th>
<th>After Two Pops</th>
</tr>
</thead>
<tbody>
<tr>
<td>R6</td>
<td>x4000</td>
<td>x3FF</td>
<td>x3FF</td>
</tr>
<tr>
<td>x3FF</td>
<td>x3FFC</td>
<td>x3FFE</td>
<td>R6</td>
</tr>
<tr>
<td>x3FFE</td>
<td>R6</td>
<td>#31</td>
<td>#18</td>
</tr>
<tr>
<td>#18</td>
<td>#18</td>
<td>#5</td>
<td>#5</td>
</tr>
<tr>
<td>#5</td>
<td>#5</td>
<td>TOP</td>
<td>TOP</td>
</tr>
</tbody>
</table>

By convention, in LC3 R6 holds the Top of Stack (TOS) pointer.

Basic Push and Pop Code

- For our implementation, stack grows downward in address space
  - (when item added, TOS moves closer to 0)

- Push
  - ADD R6, R6, #-1 ; decrement stack ptr
  - STR R0, R6, #0  ; store data (R0)

- Pop
  - LDR R0, R6, #0  ; load data from TOS
  - ADD R6, R6, #1  ; decrement stack ptr

 Anything missing...?

- Detect underflow
  - Trying to pop from empty stack

- Detect overflow
  - Trying to push into a “full” stack

- Need to send a “signal” back to calling program indicating success or failure of pop or push operations

Pop with Underflow Detection

- If we try to pop too many items off the stack, an underflow condition occurs.
  - Check for underflow by checking TOS before removing data.
  - Return status code in R5 (0 for success, 1 for underflow)

- POP
  - LD R1, EMPTY ; EMPTY = -x4000
  - ADD R2, R6, R1 ; Compare stack pointer
  - BRs FAIL ;
  - LDR R0, R6, #0
  - ADD R6, R6, #1
  - ADD R5, R5, #0 ; SUCCESS: R5 = 0
  - RET
  - FAIL AND R5, R5, #0 ; FAIL: R5 = 1
  - ADD R5, R5, #1
  - RET
  - EMPTY .FILL xCO00
Push with Overflow Detection

- If we try to push too many items onto the stack, an overflow condition occurs.
  - Check for underflow by checking TOS before adding data.
  - Return status code in R5 (0 for success, 1 for overflow)

```assembly
PUSH LD R1, MAX ; MAX = ~x3FFB
ADD R2, R6, R1 ; Compare stack pointer
BRz FAIL
ADD R6, R6, #-1
STR R0, R6, #0
AND R5, R5, #0 ; SUCCESS: R5 = 0
RET
FAIL AND R5, R5, #0 ; FAIL: R5 = 1
ADD R5, R5, #1
RET
MAX .FILL xC005
```

More Stack Use examples

- Simple calculator program
  - Also illustrates use of subroutines

- Converting ASCII to Binary
  - System always reads ASCII characters from terminal/input
  - System operates on binary numbers
  - Who does the conversion?
    - Need a subroutine

- Interrupts
  - Interrupt driven I/O

Stack uses.

- Will need to understand stack concept and use it later to implement high level language program execution

Arithmetic Using a Stack

- Instead of registers, some ISA's use a stack for source and destination operations: a zero-address machine.
  - Example:
    - ADD instruction pops two numbers from the stack, adds them, and pushes the result to the stack.

- Evaluating \( (A+B) \cdot (C+D) \) using a stack:
  - \( (A,B,+)(C,D,+)* \)
  - (1) push A
    - (2) push B
    - (3) ADD
    - (4) push C
    - (5) push D
    - (6) ADD
    - (7) MULTIPLY
    - (8) pop result

Why use a stack?
- Limited registers.
- Convenient calling convention for subroutines.
- Algorithm naturally expressed using FIFO data structure.
Example: OpAdd

- POP two values, ADD, then PUSH result.

START

POP

OK?

POP

OK?

ADD

Range

OK?

Yes

ADD

Yes

PUSH

Put back first

Put back both

RETURN

Example: OpAdd

- \text{OpAdd} \quad \text{JSR POP} ; \text{Get first operand.}
- \text{ADD R5,R5,#0} ; \text{Check for POP success.}
- \text{BRp Exit} ; \text{If error, bail.}
- \text{ADD R1,R0,#0} ; \text{Make room for second.}
- \text{JSR POP} ; \text{Get second operand.}
- \text{ADD R5,R5,#0} ; \text{Check for POP success.}
- \text{BRp Restore1} ; \text{If err, restore \& bail.}
- \text{ADD R0,R0,R1} ; \text{Compute sum.}
- \text{JSR RangeCheck} ; \text{Check size.}
- \text{BRp Restore2} ; \text{If err, restore \& bail.}
- \text{JSR PUSH} ; \text{Push sum onto stack.}
- \text{RET}
- \text{Restore2 ADD R6,R6,#-1} ; \text{Decr stack ptr (undo POP)}
- \text{Restore1 ADD R6,R6,#-1} ; \text{Decr stack ptr}
- \text{Exit RET}

Data Type Conversion

- LC3 keyboard input routines read ASCII characters, not binary values.
- Similarly, LC3 output routines write ASCII.
- You’ve seen this program:
  - \text{IN} ; \text{input from keybd into R0}
  - \text{ADD R1, R0, #0} ; \text{move to R1}
  - \text{IN} ; \text{read next input from keybd}
  - \text{ADD R2, R0, #0} ; \text{move to R2 add two inputs}
  - \text{ADD R0, R1,R2} ; \text{add the two inputs}
  - \text{OUT} ; \text{print result}
- What happened?
  - Why? ASCII ‘2’ (x32) + ASCII ‘3’ (x33) = ASCII ‘e’ (x65)

ASCII to Binary

- Useful to deal with multi-digit decimal numbers
- Assume we’ve read three ASCII digits (e.g., “259”) into a memory buffer.
- How do we convert this to a number we can use?
  - Convert first character to digit (subtract x30) and multiply by 100.
  - Convert second character to digit and multiply by 10.
  - Convert third character to digit.
  - Add the three digits together.
**Multiplication via a Lookup Table**

- How can we multiply a number by 100?
  - One approach: Add number to itself 100 times.
  - Another approach: Add 100 to itself \(<\text{number}\) times. (Better if number < 100.)

- Since we have a small range of numbers (0-9), use number as an index into a lookup table.
  - Entry 0: \(0 \times 100 = 0\)
  - Entry 1: \(1 \times 100 = 100\)
  - Entry 2: \(2 \times 100 = 200\)
  - Entry 3: \(3 \times 100 = 300\)
  etc.

**Binary to ASCII Conversion**

- Converting a 2's complement binary value to a three-digit decimal number
  - Resulting characters can be output using OUT

- Instead of multiplying, we need to divide by 100 to get hundreds digit.
  - Why wouldn’t we use a lookup table for this problem?
    - Subtract 100 repeatedly from number to divide.

- First, check whether number is negative.
  - Write sign character (+ or -) to buffer and make positive.

**Team HW: Calculator using Stack**

- Simulate a calculator using LC3
- You MUST read the Example in Chapter 10
  - You will be required to implement it in Team homework
  - Various portions of the source code have been provided, but you have to put them together and get them to work
    - There are some integration issues you need to worry about – i.e., ignoring them will lead to bugs 🙄

**More Stack uses…**

- Interrupt processing
- Will need to understand stack concept and use it later to implement high level language program execution
Recall: Interrupt-Driven I/O

- Two ways to process I/O
  - Polling: CPU waits for input device
  - Interrupt driven: device signals when it is ready

- Why?
  - Polling consumes a lot of cycles, especially for rare events – these cycles could be used to do useful computations.
  - Example: Process previous input while waiting for network/disk.

Interrupt Processing

External device can:
1. Force currently executing program to stop;
2. Have the processor satisfy the device’s needs; and
3. Resume the stopped program as if nothing happened.

Interrupt-Driven I/O

To implement an interrupt mechanism, we need:
- A way for the I/O device to signal the CPU that an interesting event has occurred.
- A way for the CPU to test whether the interrupt signal is set and whether its priority is higher than the current program.

Generating Signal
- Software sets “interrupt enable” bit in device register.
- When ready bit is set and IE bit is set, interrupt is signaled.

Priority

- Every instruction executes at a stated level of urgency.
- LC-3: 8 priority levels (PL0-PL7)
  - Example:
    - email program runs at PL0
    - Nuclear power correction program runs at PL6
  - It’s OK for PL6 device to interrupt PL0 program, but not the other way around.

- Priority encoder selects highest-priority device, compares to current processor priority level, and generates interrupt signal if appropriate.
Testing for Interrupt Signal

- CPU looks at signal between STORE and FETCH phases.
- If not set, continues with next instruction.
- If set, transfers control to interrupt service routine.

More details in Chapter 10.

Full Implementation of LC-3 Memory-Mapped I/O

Because of interrupt enable bits, status registers (KBSR/DSR) must be written, as well as read.

Interrupt-Driven I/O

- Interrupts processing:
  1. External device signals need to be serviced.
  2. Processor saves state and starts service routine.
  3. When finished, processor restores state and resumes program.

  
  Interrupt is an unscripted subroutine call, triggered by an external event.

  
  How do steps (2) and (3) occur, involves a stack.

Processor State

- What state is needed to completely capture the state of a running process?

  Processor Status Register
  - Privilege [15], Priority Level [10:8], Condition Codes [2:0]

  Program Counter
  - Pointer to next instruction to be executed.

  Registers
  - All temporary state of the process that’s not stored in memory.
    - Can be saved by interrupt processing routine (similar to TRAP routines)
Where to Save Processor State?

- Can’t use registers.
  - Programmer doesn’t know when interrupt might occur, so she can’t prepare by saving critical registers.
  - When resuming, need to restore state exactly as it was.

- Memory allocated by service routine?
  - Must save state before invoking routine, so we wouldn’t know where.
  - Also, interrupts may be nested – that is, an interrupt service routine might also get interrupted!

- Use a stack!
  - Location of stack “hard-wired”.
  - Push state to save, pop to restore.

Supervisor Stack

- A special region of memory used as the stack for interrupt service routines.
  - Initial Supervisor Stack Pointer (SSP) stored in Saved.SSP.
  - Another register for storing User Stack Pointer (USP): Saved.USP.

- Want to use R6 as stack pointer.
  - So that our PUSH/POP routines still work.

- When switching from User mode to Supervisor mode (as result of interrupt), save R6 to Saved.USP.

Invoking the Service Routine – The Details

1. If Priv = 1 (user), Saved.USP = R6, then R6 = Saved.SSP.
2. Push PSR and PC to Supervisor Stack.
5. Set PSR[2:0] = 0.
6. Set MAR = x01vv, where vv = 8-bit interrupt vector provided by interrupting device (e.g., keyboard = x80).
7. Load memory location (M[x01vv]) into MDR.
8. Set PC = MDR; now first instruction of ISR will be fetched.

Note: This all happens between the STORE RESULT of the last user instruction and the FETCH of the first ISR instruction.

Returning from Interrupt

- Special instruction – RTI – that restores state.

  RTI 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1. Pop PC from supervisor stack. (PC = M[R6]; R6 = R6 + 1)
2. Pop PSR from supervisor stack. (PSR = M[R6]; R6 = R6 + 1)
3. If PSR[15] = 1, R6 = Saved.USP.
   (If going back to user mode, need to restore User Stack Pointer.)

- RTI is a privileged instruction.
  - Can only be executed in Supervisor Mode.
  - If executed in User Mode, causes an exception.
    (More about that later.)
Example (1)

Executing ADD at location x3006 when Device B interrupts.

Example (2)

Saved. SSP = R6. R6 = Saved.SSP. Push PSR and PC onto stack, then transfer to Device B service routine (at x6200).

Example (3)

Executing AND at x6202 when Device C interrupts.

Example (4)

Push PSR and PC onto stack, then transfer to Device C service routine (at x6300).
**Example (5)**

Execute RTI at x6315; pop PC and PSR from stack.

**Example (6)**

Execute RTI at x6210; pop PSR and PC from stack.
Restore R6. Continue Program A as if nothing happened.

---

**Exception: Internal Interrupt**

- When something unexpected happens *inside* the processor, it may cause an exception.

- Examples:
  - Privileged operation (e.g., RTI in user mode)
  - Executing an illegal opcode
  - Divide by zero
  - Accessing an illegal address (e.g., protected system memory)

- Handled just like an interrupt
  - Vector is determined internally by type of exception
  - Priority is the same as running program
Problem Transformation - levels of abstraction

Interaction and Translation from High level to ISA

Natural Language
Algorithm
Program
Machine Architecture
Micro-architecture
Logic Circuits
Devices

How do user programs get executed?

What to study next..

- How are high level programs executed on the processor?
  - Review some aspects of C
  - How are operators implemented?
  - Function calls and recursion
    - How does the system support this?
    - Any guesses?
  - Memory
    - Concept of virtual memory
- Performance
  - How “efficient is your program”?