Lecture 12 Input/Output (programmer view)

Computing platforms

Novosibirsk State University
University of Hertfordshire

D. Irtegov, A.Shafarenko

2018

Memory-mapped input/output

- Device can be mapped to memory address
- In next semester we will see what “mapping” actually means
- For now, let’s imagine that memory cell 0xf3 is not a memory cell
- But a register of external device
- When you read or write this register, device can perform some action
- Or vice versa, when a device performs some action, data are written to the register (and you can read it later)

Simple device – a keyboard

- Actually, not so simple (schematics is present in CdM-8 book)
- When you press the key, 7-bit ASCII code is written to latch register
- And 8-th bit of the register is set to 1 (strobe bit).
- When you (CPU) read the register (cell at 0xf3), strobe bit is cleared
- This way you can know if the new key was pressed
- Or the same key was pressed several times

Things to consider

- For memory-mapped I/O, CdM-8 reserves upper 16 bits of memory
- This allows for 16 devices, or, more specifically, 16 I/O registers
- Single device can have several I/O registers
- Or two devices can share one address for their registers
  (e.q. one device maps register for write operations and another for reading)
- We must move stack below i/o page before doing any push and pop
- Use addsp, not 16 push commands!

How to actually input the data?

```
ldi r0, 0xf3
while
    ld r1, r0
    tst r1
stays gt
wend
```

- This is called *busy wait*

Why busy wait is good?

- It works even for simplest hardware
  (better ways require special hardware support)
- It is simple to program and debug
- It is fast

Why busy wait is bad?

- You cannot do anything else when busy waiting for a single event
- You must adapt your code when you wait for several events
- You cannot stop CPU while busy waiting
  - For CdM-8, this is not a problem
  - For real CPU, it leads to high power consumption and heating
  - If all CPU cores of typical modern smartphone will busy wait,
    they will eat all the battery in < than a hour

Interrupt

- Interrupt is a hardware mechanism implemented in CdM-8 and most ”real” modern CPU
- Interrupts allow hardware devices to call software routines
- Typically, interrupt signals that device has some data for you
- For example, keyboard has a new key pressed
- Or network interface has a new data packet received
- This is different from pure software call, but also very similar
- Details are different between CPUs and systems
- Let’s discuss CdM-8 interrupts

Interrupts from software point of view

- Every interrupt-capable device has unique number on range 0 to 7
- Every possible value of device numbers selects a byte pair, called *interrupt vector*
- By default, interrupt vectors are mapped to upper 16 bytes of memory
- In Manchester architecture, these are same bytes as used for memory mapped I/O, so you cannot use all 7 interrupts and all 16 register addresses
- In Harvard architecture, I/O is mapped to data memory, and vectors to program memory

But what happens when interrupt occurs?

- Device sets IRQ request on CPU input line
- When CPU finishes every instruction, it polls IRQ request line
- If interrupts are enabled (we will discuss this later), it retrieves device number
- Then, instead of next instruction at mem[PC], **ioi** instruction is executed
    - In some sence, **ioi** is “normal” instruction: it has an opcode, it can be inserted in a machine code and executed like any other command
    - This is called “software interrupt”
- But during interrupt, no **ioi** instruction is present at mem[PC]
- But CPU behaves like it fetched this instruction

Ioi instruction

- **Phase 1** decrement SP for stack push
- **Phase 2** store PC on stack; decrement SP for stack push
- **Phase 3** store PS on stack
- **Phase 4** fetch new PC value from vector’s first cell (0xf0 + 2R)
- **Phase 5** fetch new PS value from vector’s second cell (0xf1 + 2R)
- It is similar to **jsr**, but two registers are saved (PC and PS)
- You need to use **rti** instruction to return from **ioi** routine
- And call target depends on hardware (device number R)
- So you can write separate handler routine for every device

What you can do in interrupt handler?

- Typically, interrupt signals that device has some data for you
- So you must retrieve the data
- Some devices require further instructions, what to do next
- For example, when you read data from the disk, you must tell the disk what sector to read or write next (or not tell anything and disk will be idle)
- Then you must set some flags so main program will know the data are ready
- Then you must return to main program (execute an **rti** instruction)
- Or you can do something else
- (we will discuss it in Operating Systems course)

Why interrupts are bad?

- They are asynchronous
- They can occur in any moment of program execution
- It is very easy to write a handler that can break a main program (damage its data)
- And it is very hard to catch this condition by testing
- So, there is a mechanism to disable interrupts (a flag in PS register)
- Interrupt handling is a simplest (and historically first) form of parallel programming, and parallel programming has many pitfalls
- And most of these pitfalls are hard to avoid
- There will be courses on concurrency and parallel programming further in our curriculum

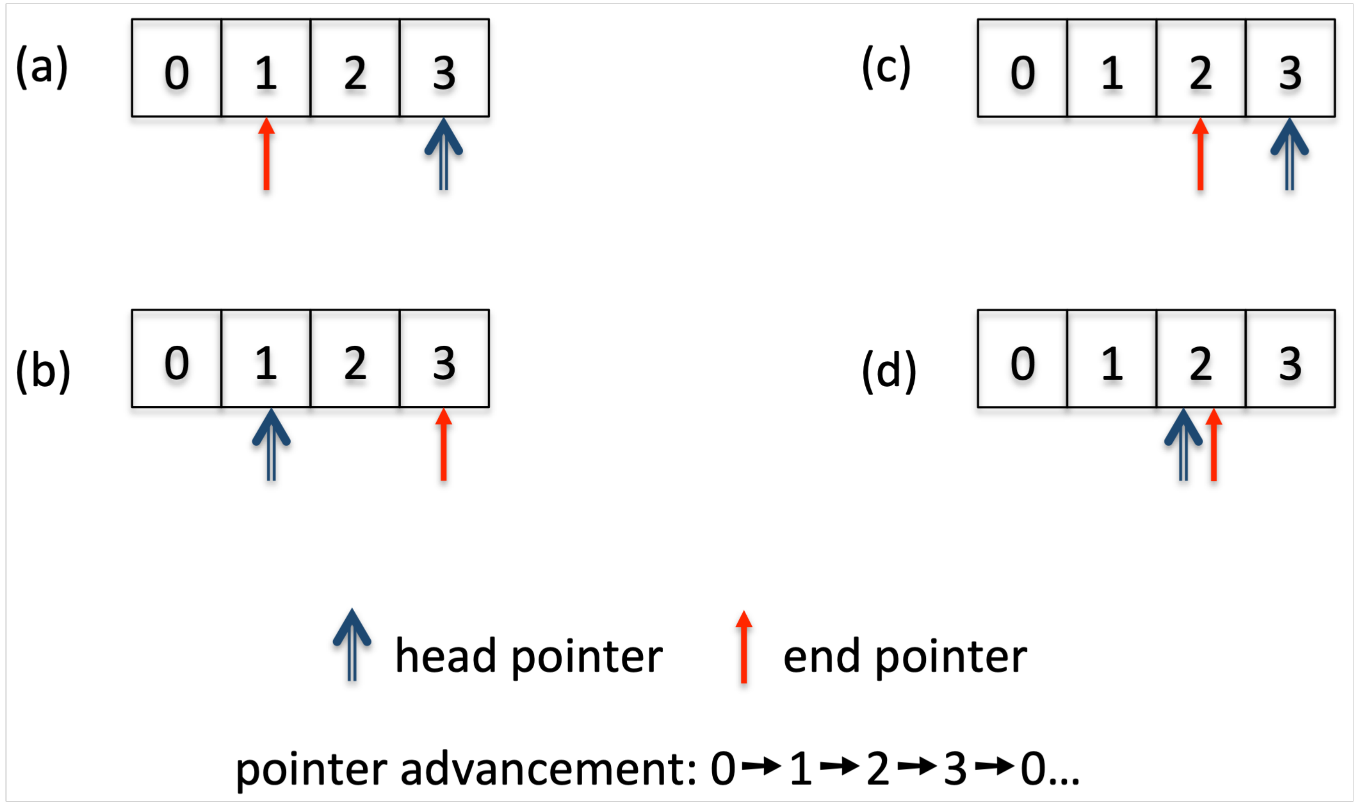
Why interrupts are good?

- You can handle several event sources at same time
- You do not need to rewrite your program to add another event source
- You can do something useful when waiting for an event
- Operating systems use interrupts to implement multithreading and multitasking

Ring buffer

- A simple technique which helps to avoid many pitfalls of parallel (asynchronous) programming
- We will hear about ring buffers and queues in Operating Systems course
- Ring buffer is very easy to implement in assembler
- In C course you’ve seen queue and stack on linked lists
- In our course, we’ve seen stack implementation on array
- Ring buffer is queue implemented on the array

Ring buffer (continued)



How it works?

- When interrupt handler retrieves the data, it advances end pointer and stores the data. When array ends, it wraps over (this is why it’s called a *ring* buffer)
- When main program processed the data, it advances head pointer (probably wrapping over) and retrieves the data
- When head pointer meets end pointer, queue is considered empty (main program must wait using busy wait or **wait** instruction)
- Instead of busy wait, main program can tell OS not to schedule it (but again this is topic of another course. We have no OS in our course)
- When end pointer meets head pointer, queue is considered full (interrupt handler cannot store data and must drop new data)

Ring buffers and queues are everywhere

- Many smart peripherial devices have internal buffers or queues, so they can send interrupts not so often, and CPU can process their data in batches
- Many things you will see in networks and operating systems (disc caches, pipes, sockets, network switches and routers) are ring buffers or queues or something built around a ring buffer or queue