UNIVERSITY OF TEXAS ARLINGTON

Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 19: Input/Output (I/O), Exceptions and Interrupts

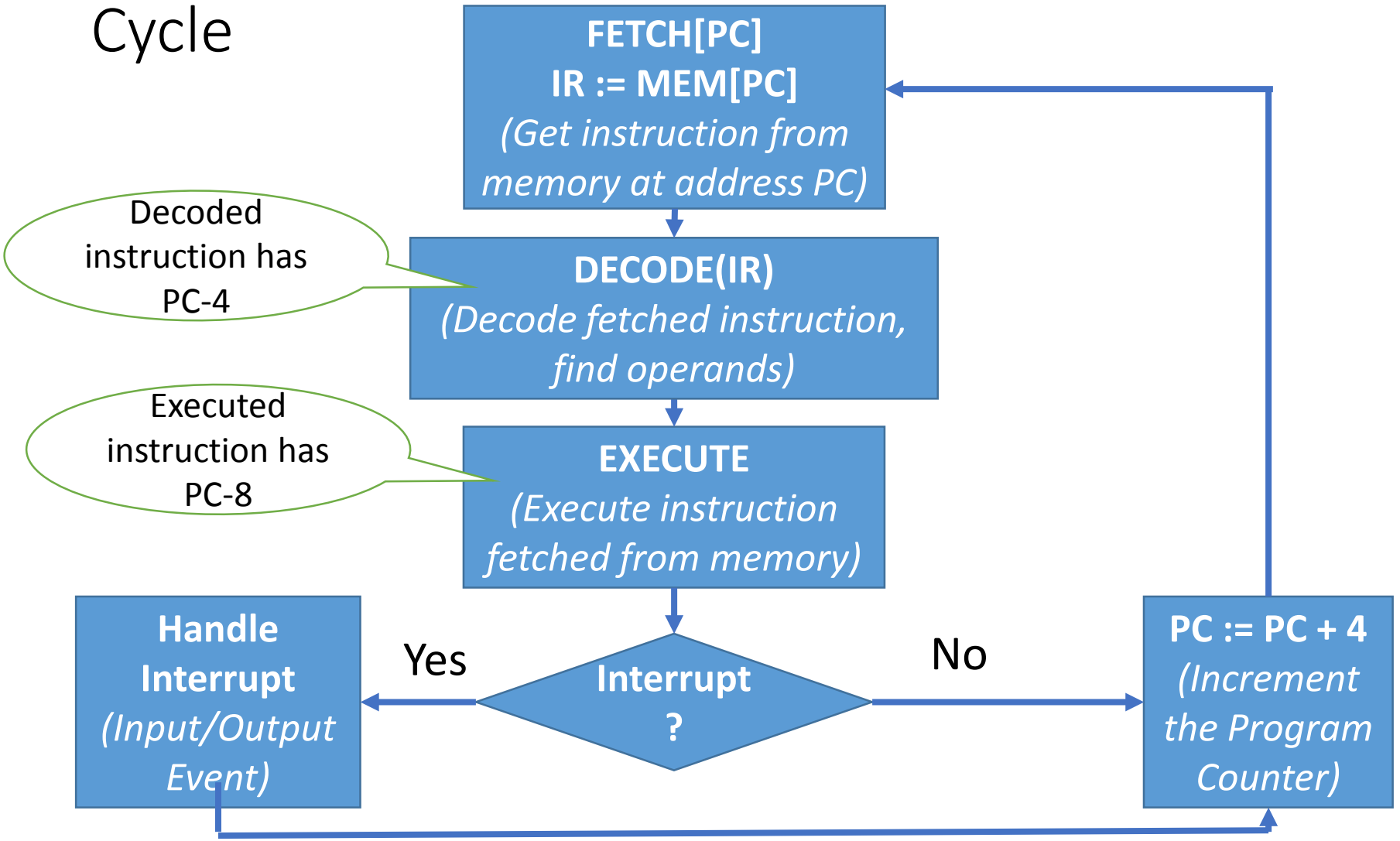
Taylor Johnson

Announcements and Outline

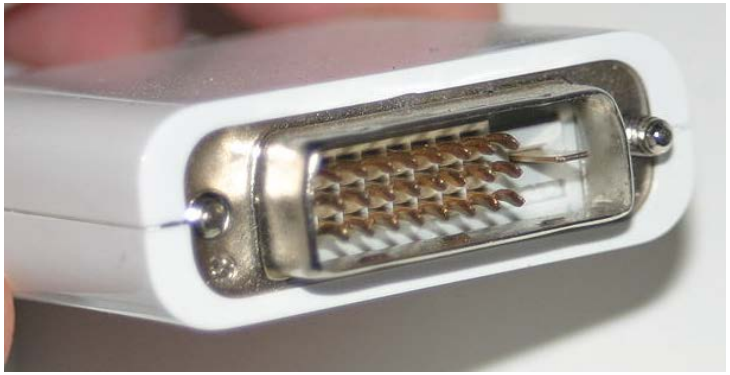
- Programming assignment 1 assigned, due 11/4

- Input/output
- Exceptions and Interrupts

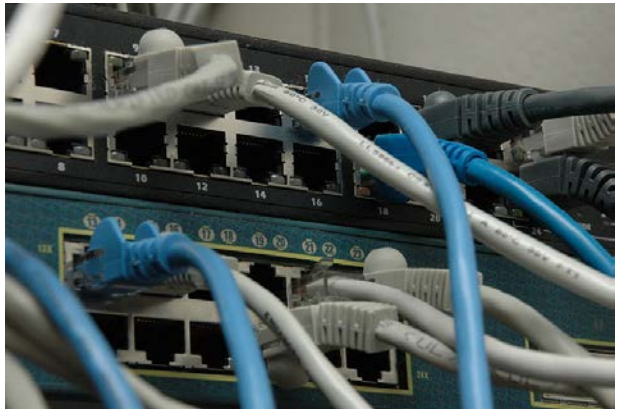
ARM 3-Stage Pipeline Processor Execution Cycle



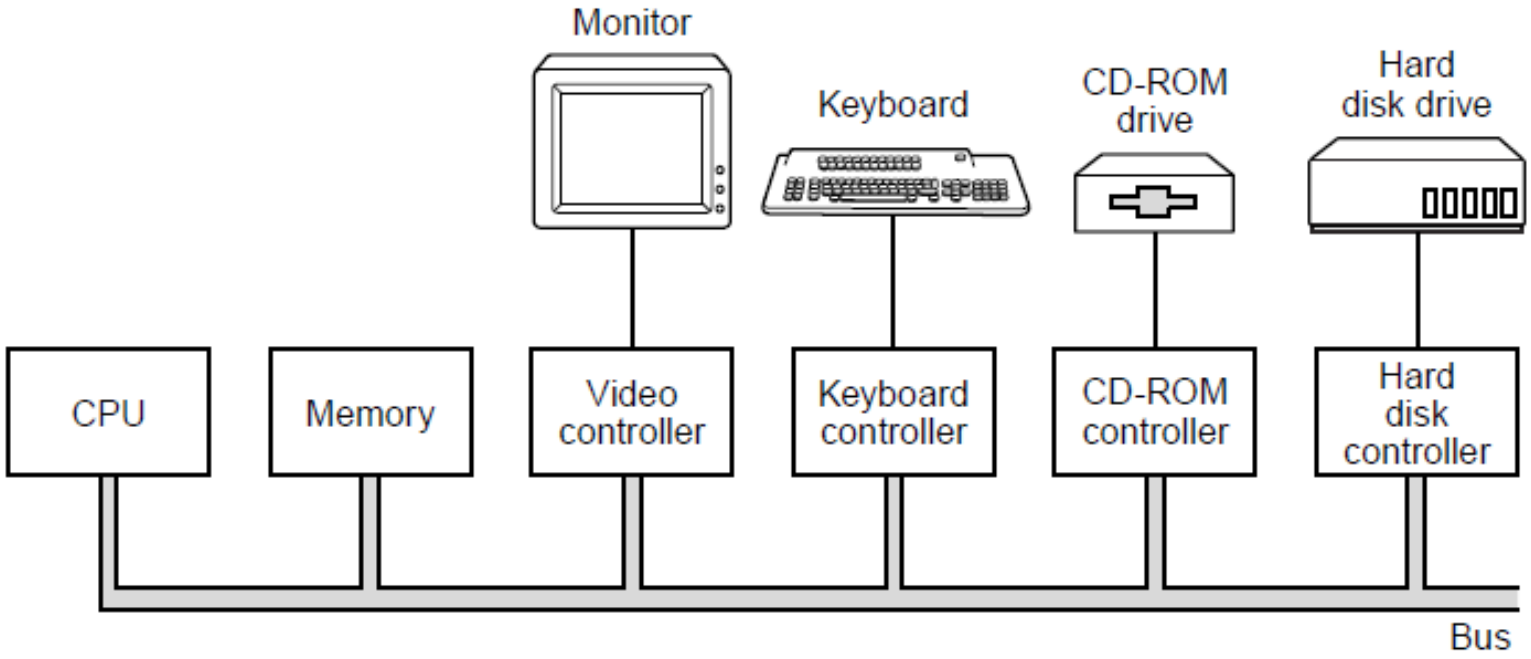
Input / Output



Networking



Logical Structure of a Computer



Logical structure of a simple personal computer.

Controllers

- The job of a controller is to:
  - Control a specific I/O device (hence the name).
  - Handle bus access for the I/O device.
  - Communicate with the CPU so as to allow the CPU (and thus software) to use the I/O device.

Example of CPU/Controller Interaction

- Example: if a program wants to read data from a disk:
  - The CPU gives a read command to the controller, specifying what data it wants to read. The program gets suspended.
  - The controller issues seeks and other commands, as necessary, to the drive.
  - When the drive begins outputting data, the controller takes that data, assembles them into words, and writes them on memory.
  - When the transfer is complete, the controller issues an interrupt.
  - The interrupt forces the CPU to stop what it is doing, and run a special procedure, called an interrupt handler, to check for errors, take any other action needed, and let the program resume.

Why Use Controllers?

- The alternative to using controllers would be for the CPU to directly communicate with the I/O devices.
- What is the benefit of using a controller?

Why Use Controllers?

- The alternative to using controllers would be for the CPU to directly communicate with the I/O devices.
- What is the benefit of using a controller?
- It greatly simplifies the task of the CPU, and even the task of the high-level computer programmer.
  - The CPU, as well as higher-level programming languages, do not have to worry about how exactly individual devices get implemented.
- It also allows device makers to make new designs.
  - The controller can "hide" the new design, and provide the same old interface to the CPU.
- Example: introducing RAID systems.
  - The controller makes them look to the CPU like regular hard drives.
  - Thus, RAID systems could be integrated seamlessly with existing machines and existing software.

Sharing the Bus

- Some controllers can access memory directly (through the bus) to read and write data.
  - This is called **Direct Memory Access (DMA)**.
- There are times when a CPU and/or some controllers all want to use the bus at the same time.
- A chip, called **bus arbiter**, decides who goes next.
  - Typically, I/O devices are given preference, because disks and other moving devices cannot be stopped, and waiting would result in losing data.
  - **Cycle stealing** is the situation where some I/O device takes control of the bus from the CPU, and thus slows down program execution.

The ISA Bus

- Cycle stealing slows down machines.
- One solution: design a new and faster bus.
- Problem: new buses are typically incompatible with old devices.
- Example: the IBM PS/2 family of computers (1987).
  - The PS/2 was supposed to be the "successor" of the PC.
  - It had a new, faster bus.
  - Disk and I/O device makers kept producing devices for the old bus. Why? Because there was so much demand from current PC owners.
  - IBM was the only PC maker that was not IBM-compatible.

The PCI and PCIe Buses

- The PS/2 example shows the risks and pitfalls of introducing a new bus, especially without consensus.
- The old PC bus, called ISA (Industry Standard Architecture) survived a bit longer.
  - Note: ISA stands for "Industry Standard Architecture" in the context of buses, but it also stands for "Instruction Set Architecture". In this course, unless specified otherwise, we use the second meaning.

The PCI Bus

- The PS/2 example shows the risks and pitfalls of introducing a new bus, especially without consensus.
- The old PC bus, called ISA (Industry Standard Architecture) survived a bit longer.
- Eventually, the ISA bus was replaced (as it was too slow) by the **PCI (Peripheral Component Interconnect)** bus.
- The PCI architecture allows the CPU-memory traffic to bypass the bus.

The PCI Bus

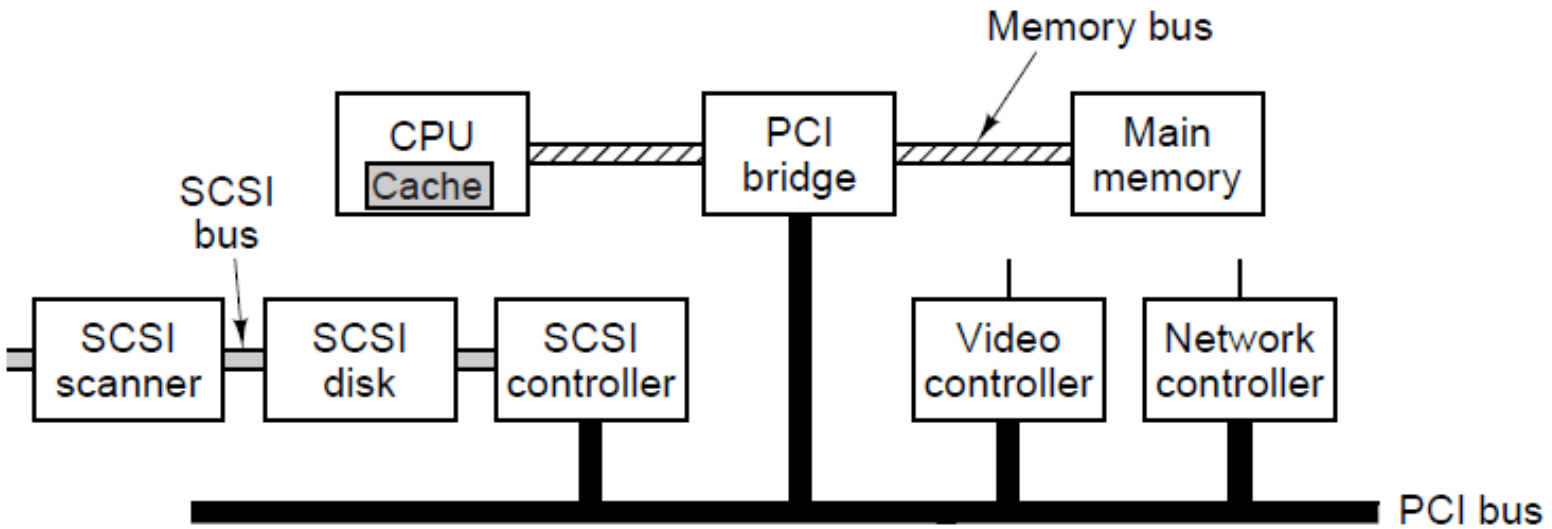


Figure 2-31. A typical PC built around the PCI bus. The SCSI controller is a PCI device.

The PCIe Bus

- The PCI bus is now also considered slow, and it is being replaced by the **PCI Express (PCIe)** bus.
- Many machines today have both buses.
  - Older, slower devices plug in to the PCI bus.
  - Newer devices plug in to the PCIe bus.
- The PCIe bus is a rather different design than PCI:
- PCI (and previous designs): A single bus line.
  - Data is broadcasted and visible to all devices.
  - The device that needs the data gets it, the other ones ignore it.
- PCIe: a point-to-point network.

The PCIe Network

- In a network, data goes from point A to point B through some intermediate stops.
- Switches are used to decide where to direct each packet of data.
  - Thus, data is not broadcast, it is only received by the target device.
- Why is this more efficient than a broadcast model?

The PCIe Network

- In a network, data goes from point A to point B through some intermediate stops.
- Switches are used to decide where to direct each packet of data.
  - Thus, data is not broadcast, it is only received by the target device.
- Why is this more efficient than a broadcast model?
- In a bus following the broadcast model:
  - if we send a data packet from A to B through the bus, no other data can be go through the bus at the same time. Cycle stealing occurs is frequent.
- In a bus following the network model:
  - multiple data packets can go through the bus at the same time, as long as they do not go through the same link at the same time. Cycle stealing is much less likely than in the broadcast model.

An Example

- An example of a PCIe system is shown on Figure 2-32 (next slide).
- In that example, it is possible to have the following two data packets going through the bus simultaneously:
  - one data packet going from the CPU to a PCIe device connected to the switch on Port 1.
  - one data packet going from the PCIe device on Port 2 to the memory.

The PCIe Bus

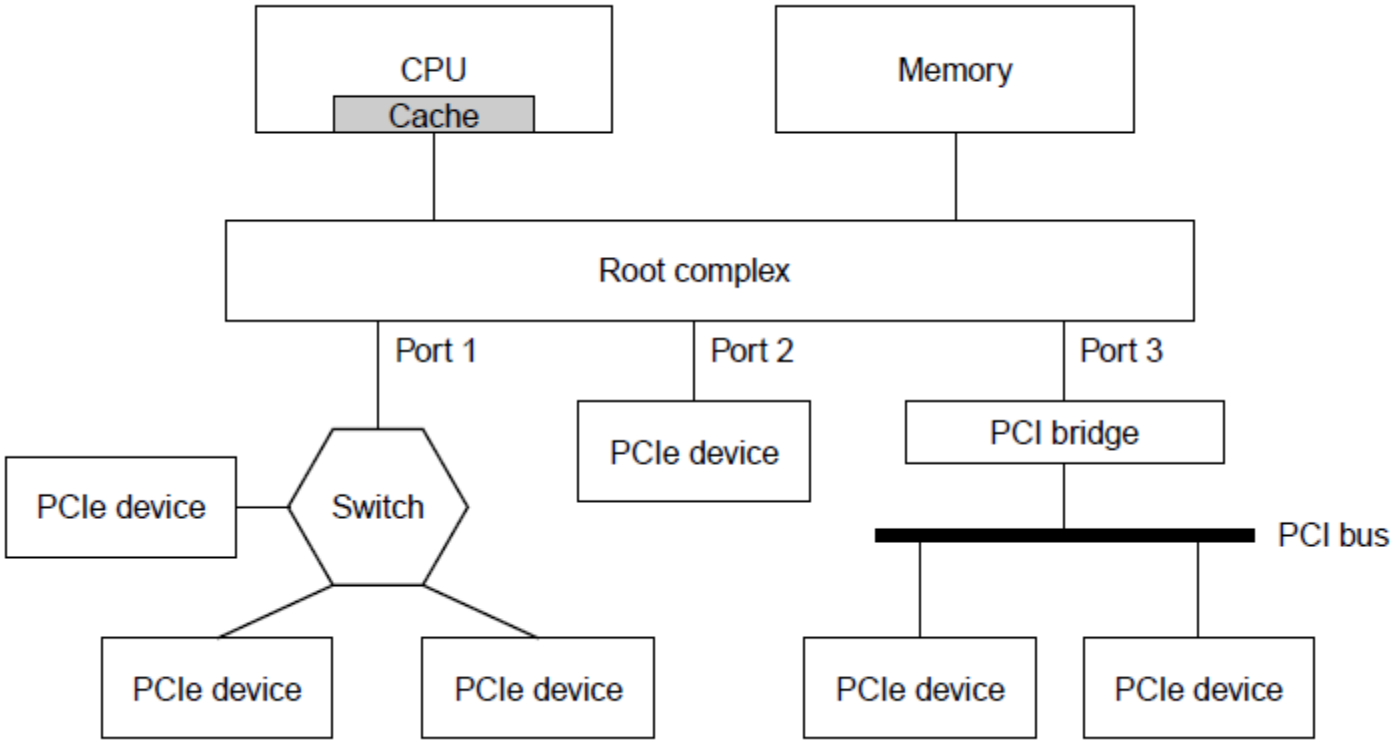


Figure 2-32. Sample architecture of a PCIe system with three PCIe ports.

PCIe: 1-Bit Lanes

- Traffic in the PCIe bus is separated into individual lanes, that are 1-bit wide.
  - An individual device can have up to 32 such lanes.
- This means that data goes through each lane bit-by-bit, not in parallel.
- Reason: sending data in parallel (say using a 32-bit lane) can be tricky, because we need to make sure all data arrive at the same time.
  - The clock rate is significantly slowed down, to ensure that.
- With a 1-bit lane, there is no need for synchronization among bits, so the clock rate is much higher.

Benefits of 1-Bit Lanes

- Example:
  - A PCI bus has:
    - maximum clock rate 66MHz.
    - a single 64-bit wide lane.
  - Consequently, a PCI bus can support a data rate of at most ??? MB/sec.
  - A PCIe bus has a clock rate of 8GHz.
  - Thus, a PCIe bus can support a data rate of ??? on a single lane.
- Multiple lanes mean even faster data rates.
- The graphics card can have 16 lanes, getting ??? /sec.

Benefits of 1-Bit Lanes

- Example:
  - A PCI bus has:
    - maximum clock rate 66MHz.
    - a single 64-bit wide lane.
  - Consequently, a PCI bus can support a data rate of at most 528 MB/sec (528=66*64/8).
  - A PCIe bus has a clock rate of 8GHz.
  - Thus, a PCIe bus can support a data rate of 1GB/sec on a single lane.
- Multiple lanes mean even faster data rates.
- The graphics card can have 16 lanes, getting 16GB/sec.

Terminals

- Terminals used to be devices used to access a main computer.
- A terminal would consist of a keyboard and a monitor, often integrated into a single device.
- This terminal would connect to the main computer by a serial line or over a telephone line.
- Terminals supported having multiple users use a single, powerful computer.
- Terminals still used in airline reservations, banking, and some other industries.

Keyboards

- When a key is depressed, the keyboard controller generates an interrupt.
- The keyboard interrupt handler reads a hardware register inside the keyboard controller, that contains the code of the key that was pressed.
  - A number between 1 and 102.
- When a key is released, a second interrupt is caused.

Handling Multikey Combinations

- How do we get a capital M?
  - We press SHIFT.
  - With SHIFT down, we press the 'm' key.
  - Then we release both keys (the order does not matter).
- What does the keyboard controller send to the operating system?
  - SHIFT pressed
  - 'm' pressed
  - 'm' released
  - SHIFT released.
- The mapping of these four events into a capital M is done by the software .
- This software can be the operating system, or another program that runs on top of the operating system.

Handling Multikey Combinations

- How do we get a capital M?
  - We press SHIFT.
  - With SHIFT down, we press the 'm' key.
  - Then we release both keys (the order does not matter).
- What does the keyboard controller send to the operating system?
  - SHIFT pressed
  - 'm' pressed
  - 'm' released
  - SHIFT released.
- The mapping of these four events into a capital M is done by the software .
- What are the pros and cons of doing the mapping in software vs. doing it in hardware?

Handling Multikey Combinations

- How do we get a capital M?
  - We press SHIFT.
  - With SHIFT down, we press the 'm' key.
  - Then we release both keys (the order does not matter).
- What does the keyboard controller send to the operating system?
  - SHIFT pressed
  - 'm' pressed
  - 'm' released
  - SHIFT released.
- The mapping of these four events into a capital M is done by the software .
- Advantages of doing the mapping in software:
  - Simplifies the hardware. Also, ensures that all keyboards behave similarly.
  - Allows support of multiple alphabets/languages by the same keyboard.

Displays and Memory

- Most monitors are **refreshed** 60-100 times per second.
- What does refreshing mean? It means redrawing the image on the monitor.
- The image is determined pixel-by-pixel.
- A typical 1920x1080 monitor has about 2 million pixels (1920 * 1080 = 2,073,600).
- To describe each pixel, we need three bytes, i.e., three numbers from 0 to 255, to describe the color.
  - Every color can be decomposed into red, green, blue parts.

Data Rate for Displays

- In total, we need ???MB of memory to store what we see in the monitor at one moment.
- This data is usually stored in a special memory, called the **Video RAM**.
- For regular movie-quality video, the image must be refreshed at least 30 times per second.
- This translates into sending to the monitor ???MB/sec.
- PCIe buses can easily handle such a load.

Data Rate for Displays

- In total, we need 6MB of memory to store what we see in the monitor at one moment.
- This data is usually stored in a special memory, called the **Video RAM**.
- For regular movie-quality video, the image must be refreshed at least 30 times per second.
- This translates into sending to the monitor 180MB/sec.
- PCIe buses can easily handle such a load.

Mice

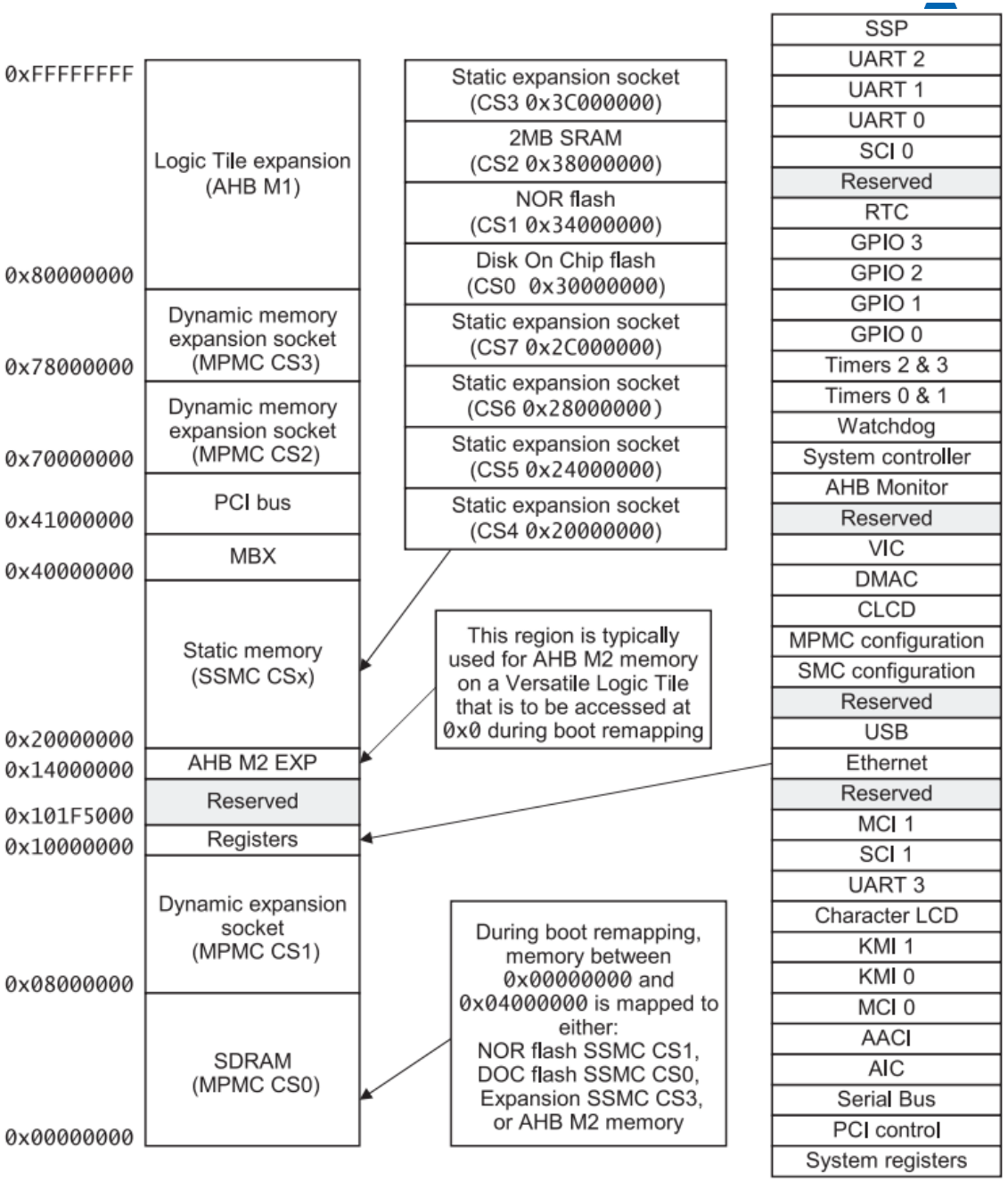
- When a mouse moves more than a certain minimum distance (e.g., 0.01 inches), the mouse sends 3 bytes to the computer:
  - The number of units the mouse moved in the x direction.
  - The number of units the mouse moved in the y direction.
  - The current state of the mouse buttons.
- Information is also sent when buttons are pressed and released.
- This information generates interrupts.
- Interpreting this information (e.g., as clicks, double clicks, drags, drops) is done by the operating system.

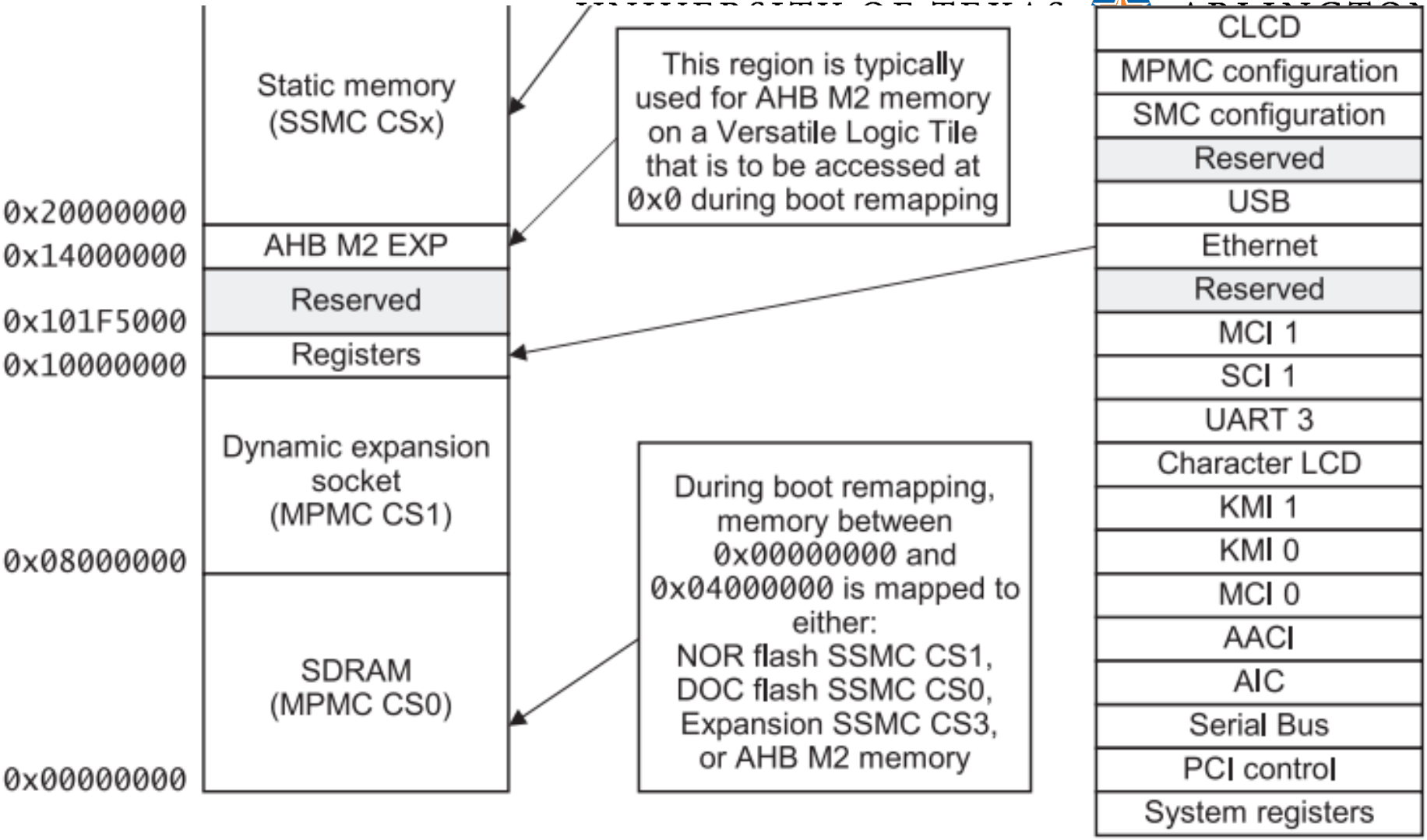
Memory-Mapped I/O

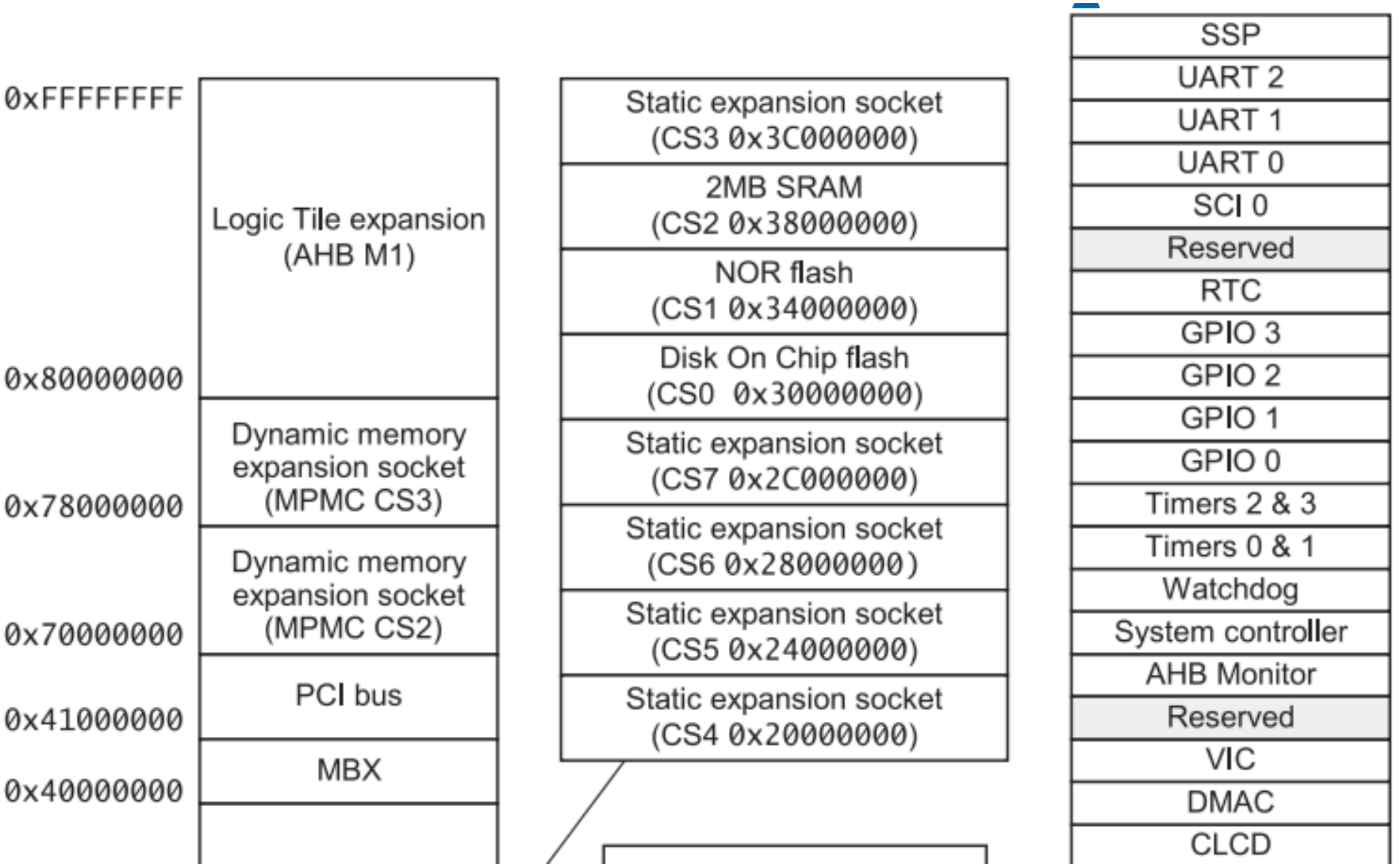
- Some of our original examples displayed output to console by writing to a special memory address

```
.equ    ADDR_UART0, 0x101f1000
ldr     r0,=ADDR_UART0@ r0 := 0x 101f 1000
mov     r2,#'a'             @ R2 := 'a'
str     r2,[r0]             @ MEM[r0] := r2
```

- How does this work? Memory Mapped I/O
  - Registers on peripheral devices (keyboards, monitors, network controllers, etc.) are addressable in same address space as main memory, and their values are mapped (i.e., readable / writeable at certain addresses)
  - How to read input values?
    - Polling vs. interrupts


0xFFFFFFFF
Logic Tile expansion (AHB M1)
0x80000000
Dynamic memory expansion socket (MPMC CS3)
0x78000000
Dynamic memory expansion socket (MPMC CS2)
0x70000000
PCI bus
0x41000000
MBX
0x40000000
Static memory (SSMC CSx)
0x20000000
AHB M2 EXP
0x14000000
Reserved
0x101F5000
Registers
0x10000000
Dynamic expansion socket (MPMC CS1)
0x08000000
SDRAM (MPMC CS0)
0x00000000
Static expansion socket (CS3 0x3C000000)
2MB SRAM (CS2 0x38000000)
NOR flash (CS1 0x34000000)
Disk On Chip flash (CS0 0x30000000)
Static expansion socket (CS7 0x2C000000)
Static expansion socket (CS6 0x28000000)
Static expansion socket (CS5 0x24000000)
Static expansion socket (CS4 0x20000000)
This region is typically used for AHB M2 memory on a Versatile Logic Tile that is to be accessed at 0x0 during boot remapping
During boot remapping, memory between 0x00000000 and 0x04000000 is mapped to either: NOR flash SSMC CS1, DOC flash SSMC CS0, Expansion SSMC CS3, or AHB M2 memory
SSP
UART 2
UART 1
UART 0
SCI 0
Reserved
RTC
GPIO 3
GPIO 2
GPIO 1
GPIO 0
Timers 2 & 3
Timers 0 & 1
Watchdog
System controller
AHB Monitor
Reserved
VIC
DMAC
CLCD
MPMC configuration
SMC configuration
Reserved
USB
Ethernet
Reserved
MCI 1
SCI 1
UART 3
Character LCD
KMI 1
KMI 0
MCI 0
AACI
AIC
Serial Bus
PCI control
System registers


Static memory
(SSMC CSx)
AHB M2 EXP
Reserved
Registers
Dynamic expansion
socket
(MPMC CS1)
SDRAM
(MPMC CS0)
0x20000000
0x14000000
0x101F5000
0x10000000
0x08000000
0x00000000
This region is typically used for AHB M2 memory on a Versatile Logic Tile that is to be accessed at 0x0 during boot remapping
During boot remapping, memory between 0x00000000 and 0x04000000 is mapped to either:
NOR flash SSMC CS1,
DOC flash SSMC CS0,
Expansion SSMC CS3,
or AHB M2 memory
CLCD
MPMC configuration
SMC configuration
Reserved
USB
Ethernet
Reserved
MCI 1
SCI 1
UART 3
Character LCD
KMI 1
KMI 0
MCI 0
AACI
AIC
Serial Bus
PCI control
System registers


0xFFFFFFFF
Logic Tile expansion (AHB M1)
0x80000000
Dynamic memory expansion socket (MPMC CS3)
0x78000000
Dynamic memory expansion socket (MPMC CS2)
0x70000000
PCI bus
0x41000000
MBX
0x40000000
Static expansion socket (CS3 0x3C000000)
2MB SRAM (CS2 0x38000000)
NOR flash (CS1 0x34000000)
Disk On Chip flash (CS0 0x30000000)
Static expansion socket (CS7 0x2C000000)
Static expansion socket (CS6 0x28000000)
Static expansion socket (CS5 0x24000000)
Static expansion socket (CS4 0x20000000)
SSP
UART 2
UART 1
UART 0
SCI 0
Reserved
RTC
GPIO 3
GPIO 2
GPIO 1
GPIO 0
Timers 2 & 3
Timers 0 & 1
Watchdog
System controller
AHB Monitor
Reserved
VIC
DMAC
CLCD

Address from Memory-Map in Manual

Programmer's Reference

Table 4-1 Memory map (continued)

Peripheral	Location	Interrupt[a] PIC and SIC	Address	Region size
UART 0 Interface	Dev. chip	PIC 12	0x101F1000-0x101F1FFF	4KB
UART 1 Interface	Dev. chip	PIC 13	0x101F2000-0x101F2FFF	4KB
UART 2 Interface	Dev. chip	PIC 14	0x101F3000-0x101F3FFF	4KB

http://infocenter.arm.com/help/topic/com.arm.doc.dui0224i/DUI0224I_realview_platform_baseboard_for_arm926ej_s_ug.pdf

UART Register Map

Table 3-1 Register summary

Offset	Type	Width	Reset value	Name	Description
0x000	RW	12/8	0x---	UARTDR	*Data register, UARTDR* on page 3-5
0x004	RW	4/0	0x0	UARTRSR/ UARTECR	*Receive status register/error clear register, UARTRSR/UARTECR* on page 3-6
0x008-0x014	-	-	-	-	Reserved
0x018	RO	9	0b-10010---	UARTFR	*Flag register, UARTFR* on page 3-8
0x01C	-	-	-	-	Reserved

http://infocenter.arm.com/help/topic/com.arm.doc.ddi0183f/DDI0183.pdf

UART Flag Register Bits

3.3.3 Flag register, UARTFR

The UARTFR register is the flag register. After reset TXFF, RXFF, and BUSY are 0, and TXFE and RXFE are 1. Table 3-4 shows the bit assignment of the UARTFR register.

Table 3-4 UARTFR register

Bits	Name	Function
15:9	-	Reserved, do not modify, read as zero.
8	RI	Ring indicator. This bit is the complement of the UART ring indicator (**nUARTRI**) modem status input. That is, the bit is 1 when the modem status input is 0.
7	TXFE	Transmit FIFO empty. The meaning of this bit depends on the state of the FEN bit in the UARTLCR_H register. If the FIFO is disabled, this bit is set when the transmit holding register is empty. If the FIFO is enabled, the TXFE bit is set when the transmit FIFO is empty. This bit does not indicate if there is data in the transmit shift register.
6	RXFF	Receive FIFO full. The meaning of this bit depends on the state of the FEN bit in the UARTLCR_H register. If the FIFO is disabled, this bit is set when the receive holding register is full. If the FIFO is enabled, the RXFF bit is set when the receive FIFO is full.
5	TXFF	Transmit FIFO full. The meaning of this bit depends on the state of the FEN bit in the UARTLCR_H register. If the FIFO is disabled, this bit is set when the transmit holding register is full. If the FIFO is enabled, the TXFF bit is set when the transmit FIFO is full.
4	RXFE	Receive FIFO empty. The meaning of this bit depends on the state of the FEN bit in the UARTLCR_H register. If the FIFO is disabled, this bit is set when the receive holding register is empty. If the FIFO is enabled, the RXFE bit is set when the receive FIFO is empty.
3	BUSY	UART busy. If this bit is set to 1, the UART is busy transmitting data. This bit remains set until the complete byte, including all the stop bits, has been sent from the shift register. This bit is set as soon as the transmit FIFO becomes non-empty (regardless of whether the UART is

http://infocenter.arm.com/help/topic/com.arm.doc.ddi0183f/DDI0183.pdf

Reading Input from UART (*POLLING*)

```
.equ IO_ADDRESS, 0x101f1000    @ uart memory-map address
.equ OFFSET_FR, 0x018          @ flag register offset from uart
.equ RXFE, 0x10                @ receive status bit
.equ TXFF, 0x20                @ transmit status bit

get_char:
      push {r2, r3, r4, lr}    @ preamble
      ldr r4,=IO_ADDRESS       @ r4 := 0x 101f 1000

get_char_wait:
      ldr r2,[r4,#OFFSET_FR]   @ load IO flag register to r2
      and r3,r2,#RXFE          @ mask non receive fifo empty bits
      cmp r3, #0               @ check if r3 == 0
      bne get_char_wait        @ wait if not ready (if r3 != 0)

      ldr r0, [r4]             @ read character
      str r0, [r4]             @ echo character to screen

      pop {r2, r3, r4, lr}     @ wrap up
      bx lr
```

Viewing Memory-Mapped Registers with gdb

```
.equ IO_ADDRESS, 0x101f1000   @ uart memory-map address
.equ OFFSET_FR, 0x018         @ flag register offset from uart
.equ RXFE, 0x10               @ receive status bit
```

```
(gdb) x /16x 0x101f1000                    <- View all registers
0x101f1000:     0x00000000      0x00000000      0x00000000
     0x00000000
0x101f1010:     0x00000000      0x00000000      0x00000090
     0x00000000
0x101f1020:     0x00000000      0x00000000      0x00000000
     0x00000000
0x101f1030:     0x00000300      0x00000012      0x00000000
     0x00000020
(gdb) x /1x 0x101f1000+0x018               <- View Flag Register
0x101f1018:     0x00000090
(gdb) x /1t 0x101f1000+0x018               <- View Flag Register
0x101f1018:     00000000000000000000000010010000
(gdb) x /1t 0x101f1000+0x018    <- Character entered
0x101f1018:     00000000000000000000000011000000
```

String Output

- So far we have seen character input/output
- That is, one char at a time

- What about strings (character arrays, i.e., multiple characters)?

- Recall that strings are stored in memory at consecutive addresses

```
string_abc:
.asciz "abcdefghijklmnopqrstuvwxyz\n\r"
.word 0x00
```

ADDR	Byte 3	Byte 2	Byte 1	Byte 0
0x1000	'd'	'c'	'b'	'a'
0x1004	'h'	'g'	'f'	'e'
0x1008	'l'	'k'	'j'	'i'
0x100c	'p'	'o'	'n'	'm'
0x1010	't'	's'	'r'	'q'
0x1014	'x'	'w'	'v'	'u'
0x1018	'\0'	'\0'	'z'	'y'

Assembler Output

```
0001012e <string_abc>:
   1012e: 64636261   strbtvs   r6, [r3], #-609; 0x261
   10132: 68676665   stmdavs   r7!, {r0, r2, r5, r6, r9, sl, sp,
lr}^
   10136: 6c6b6a69   stclvs    10, cr6, [fp], #-420; 0xfffffe5c
   1013a: 706f6e6d   rsbvc     r6, pc, sp, ror #28
   1013e: 74737271   ldrbtvc   r7, [r3], #-625; 0x271
   10142: 78777675   ldmdavc   r7!, {r0, r2, r4, r5, r6, r9, sl,
ip, sp, lr}^
   10146: 0d0a7a79   vstreq    s14, [sl, #-484]    ; 0xfffffe1c
   1014a: 00000000   andeq     r0, r0, r0
```

ASCII

Binary	Octal	Decimal	Hex	Glyph
110 0000	140	96	60	`
110 0001	141	97	61	a
110 0010	142	98	62	b
110 0011	143	99	63	c
110 0100	144	100	64	d
110 0101	145	101	65	e
110 0110	146	102	66	f
…				…
111 1000	170	120	78	x
111 1001	171	121	79	y
111 1010	172	122	7A	z

Printing Strings

```
@ assumes r0 contains uart data register address
@ r1 should contain first character of string to display
print_string: push  {r1,r2,lr}
str_out: ldrb  r2,[r1]
     cmp  r2,#0x00  @ '\0' = 0x00: null character?
     beq  str_done  @ if yes, quit
     str  r2,[r0]   @ otherwise, write char of string
     add  r1,r1,#1  @ go to next character
     b    str_out   @ repeat
str_done: pop  {r1,r2,lr}
     bx   lr
```

Character Codes

- Computers and devices oftentimes use text as part of the messages they exchange.
- To understand each other, different devices must represent text the same way.
- A few standard conventions have been established.
  - ASCII
  - Unicode
  - UTF-8

How Many Letters Do We Need?

- To accommodate text written in the English language, can you make a very rough estimation of how many characters we need?

How Many Letters Do We Need?

- To accommodate text written in the English language, can you make a very rough estimation of how many characters we need?
  - 26 letters * 2 (upper and lower case) = 52.
  - 10 numbers.
  - 30 punctuation symbols and other commonly used symbols, such as $, #, @, %.
- We get a rough estimate of about 90 letters.
- How many bits do we need per letter then?

How Many Letters Do We Need?

- To accommodate text written in the English language, can you make a very rough estimation of how many characters we need?
  - 26 letters * 2 (upper and lower case) = 52.
  - 10 numbers.
  - 30 punctuation symbols and other commonly used symbols, such as $, #, @, %.
- We get a rough estimate of about 90 letters.
- How many bits do we need per letter then?
  - ceil($\log_2(90)$) = 7.

The ASCII Code

- This is the most well-known and widely used convention for text representation.
- 7 bits per character, stored in a byte.
- Plenty of room for 26 upper case letters, 26 lower case letters, numbers, punctuation, and some special symbols.
- Codes 0-31 are special codes, such as:
  - 0: NULL character (for terminating strings)
  - 9: Horizontal tab…
- Some of these special codes are rarely used. E.g.:
  - 1: SOH (start of heading), 2: STX (start of text).

ASCII Limitations

- 7 (or 8) bits cannot support:
  - Alphabets with thousands of characters, such as Chinese and Japanese.
  - Multiple small alphabets used at the same time: for example, English (Latin), Russian (Cyrillic), Greek, Arabic, Hebrew…
- The ASCII convention is English-centric.

Unicode

- A character is assigned a 16-bit value.
- Codes 0-255 match ASCII codes.
- Diacritical marks (accents, umlauts) get their own code.
  - It is up to the software to combine a diacritical mark and a letter into a single character on the screen.
- Supported by Windows, Java, and many applications
- Covers "all" alphabets simultaneously.
- This gives room for 65,536 unique characters.
  - The world's alphabets collectively use about 200,000 symbols, so even this is not quite enough.
  - Many Chinese/Japanese characters still left out.

UTF-8

- UTF-8 characters are variable length, from 7 to 31 bits.
  - Max number of bytes for a character is 6 (= 48 bits), due to additional "header bits".
- Codes 0-127 are the ASCII characters.
  - If the most significant bit of the first byte is 0, then the byte is treated as an ASCII code.
- Overall, the first byte of every character has "header" bits that specify how many bytes that character has.
- Continuation bytes of a UTF-8 character always start with "header" bits 10, whereas the initial byte never does.
  - Makes it easy to re-synchronize after a communication error.
- Currently, UTF-8 supports 1,114,112 characters.
  - However, it has room for a lot more characters.
- Everyone is happy with UTF-8, so far…

UTF-8 Header Bits

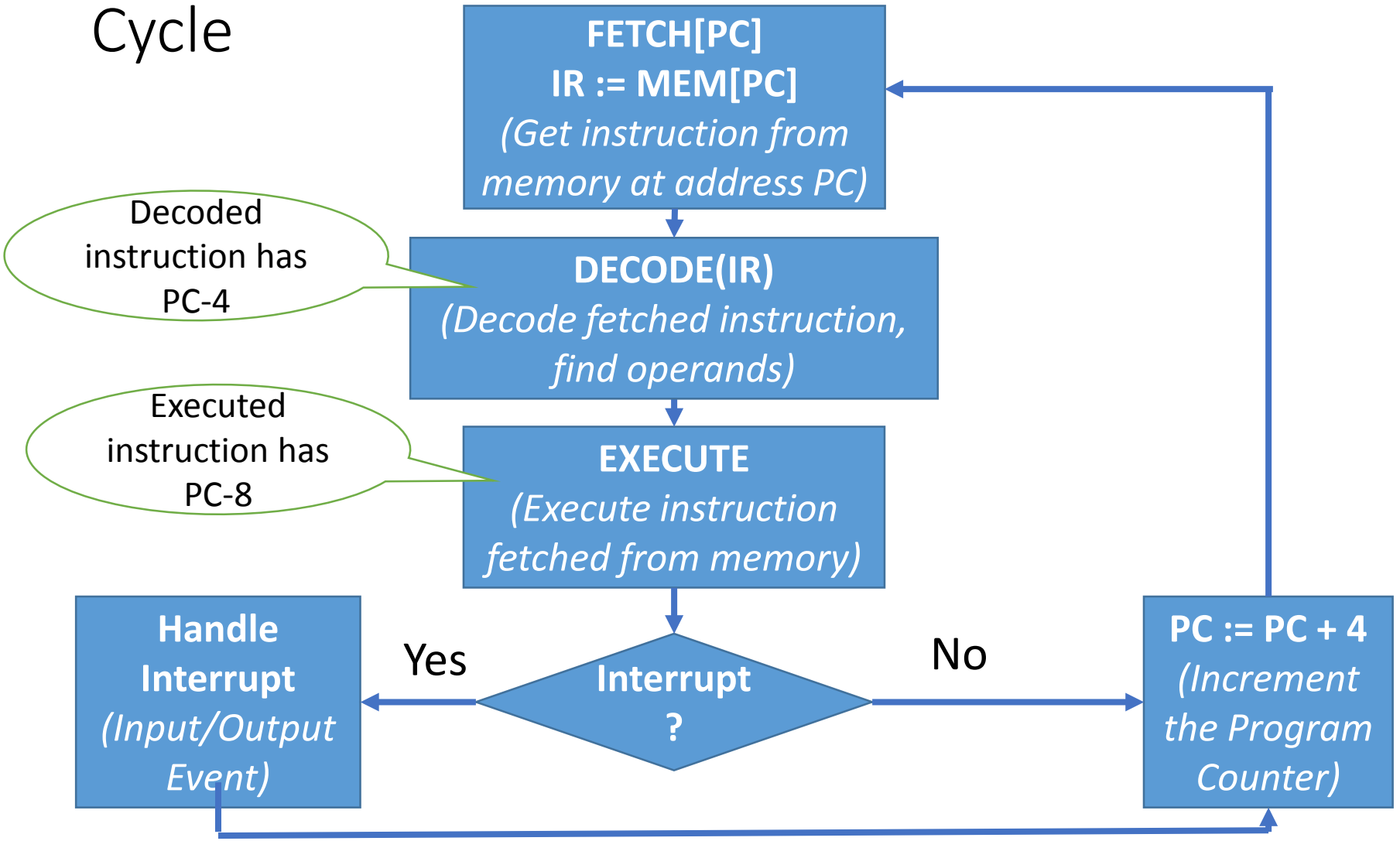
Header	Length of Header	Meaning
0	1	This byte contains an ASCII code
110	3	This is the first byte of a 2-byte character.
1110	4	This is the first byte of a 3-byte character.
11110	5	This is the first byte of a 4-byte character.
111110	6	This is the first byte of a 5-byte character.
1111110	7	This is the first byte of a 6-byte character.
10	2	This is a continuation byte (not a first byte).

The UTF-8 Format

Bits	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
7	0ddddddd					
11	110ddddd	10dddddd				
16	1110dddd	10dddddd	10dddddd			
21	11110ddd	10dddddd	10dddddd	10dddddd		
26	111110dd	10dddddd	10dddddd	10dddddd	10dddddd	
31	1111110x	10dddddd	10dddddd	10dddddd	10dddddd	10dddddd

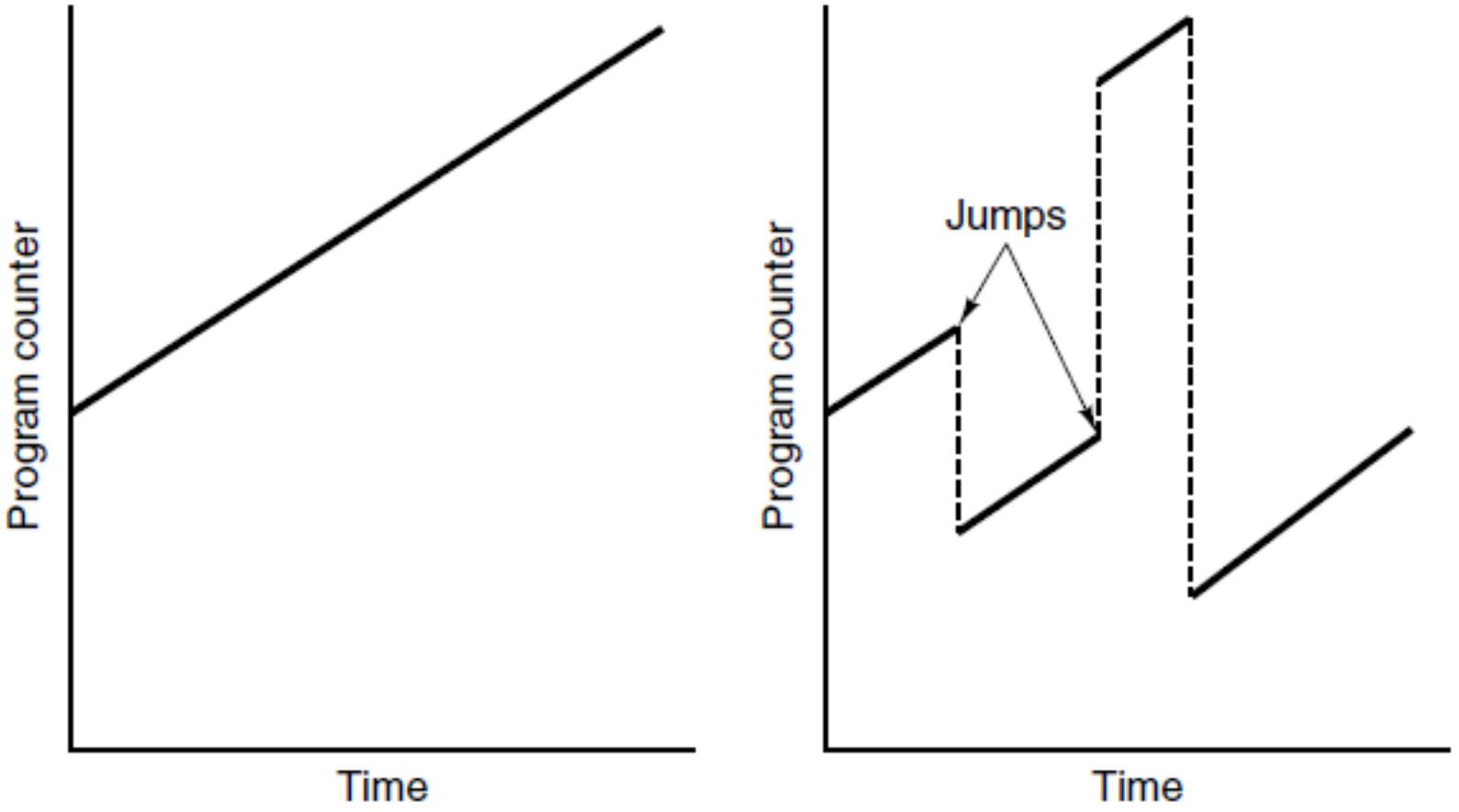
Figure 2-45. The UTF-8 encoding scheme.

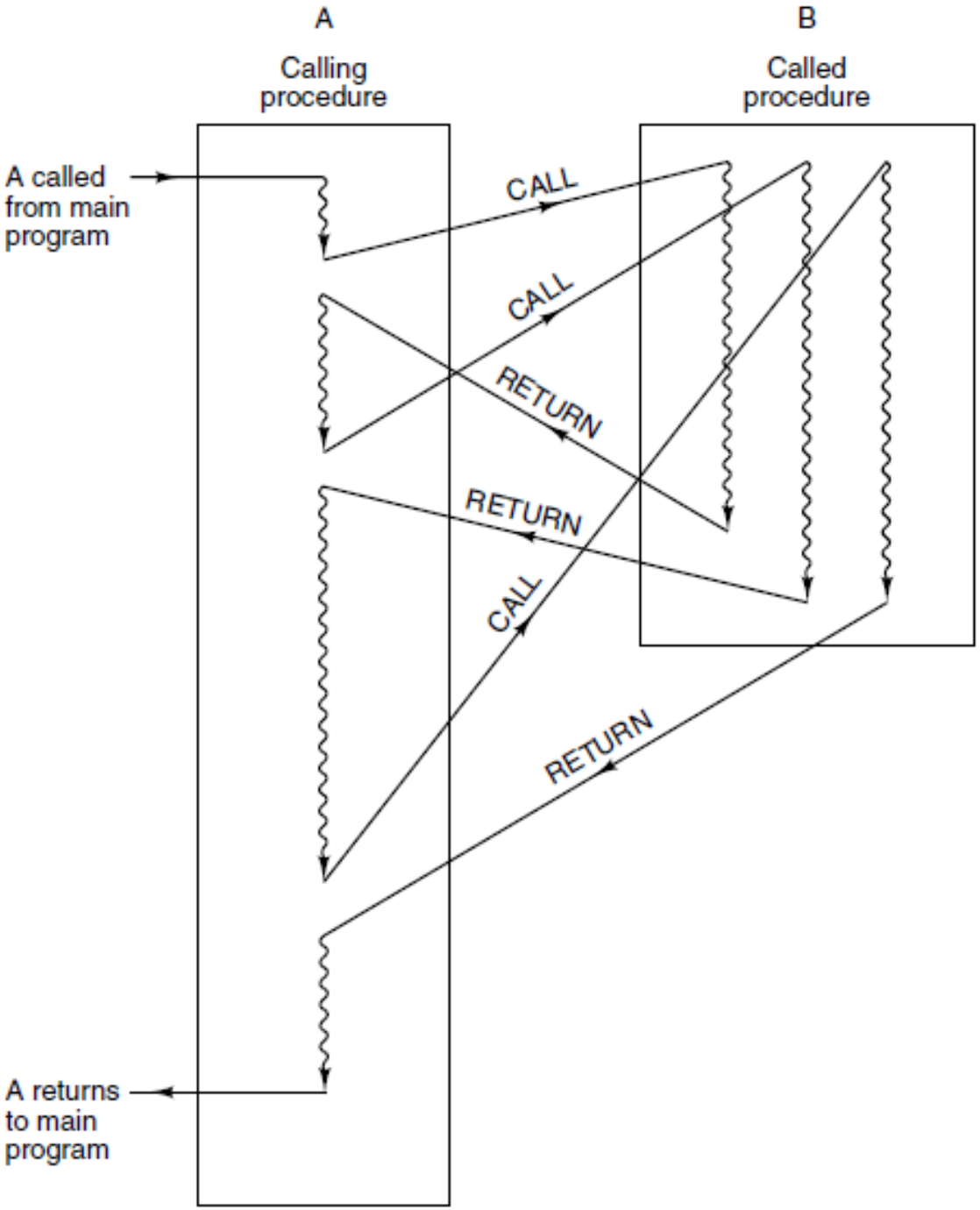
ARM 3-Stage Pipeline Processor Execution Cycle



Flow of Control

- Left: no branches; Right: branches / jumps

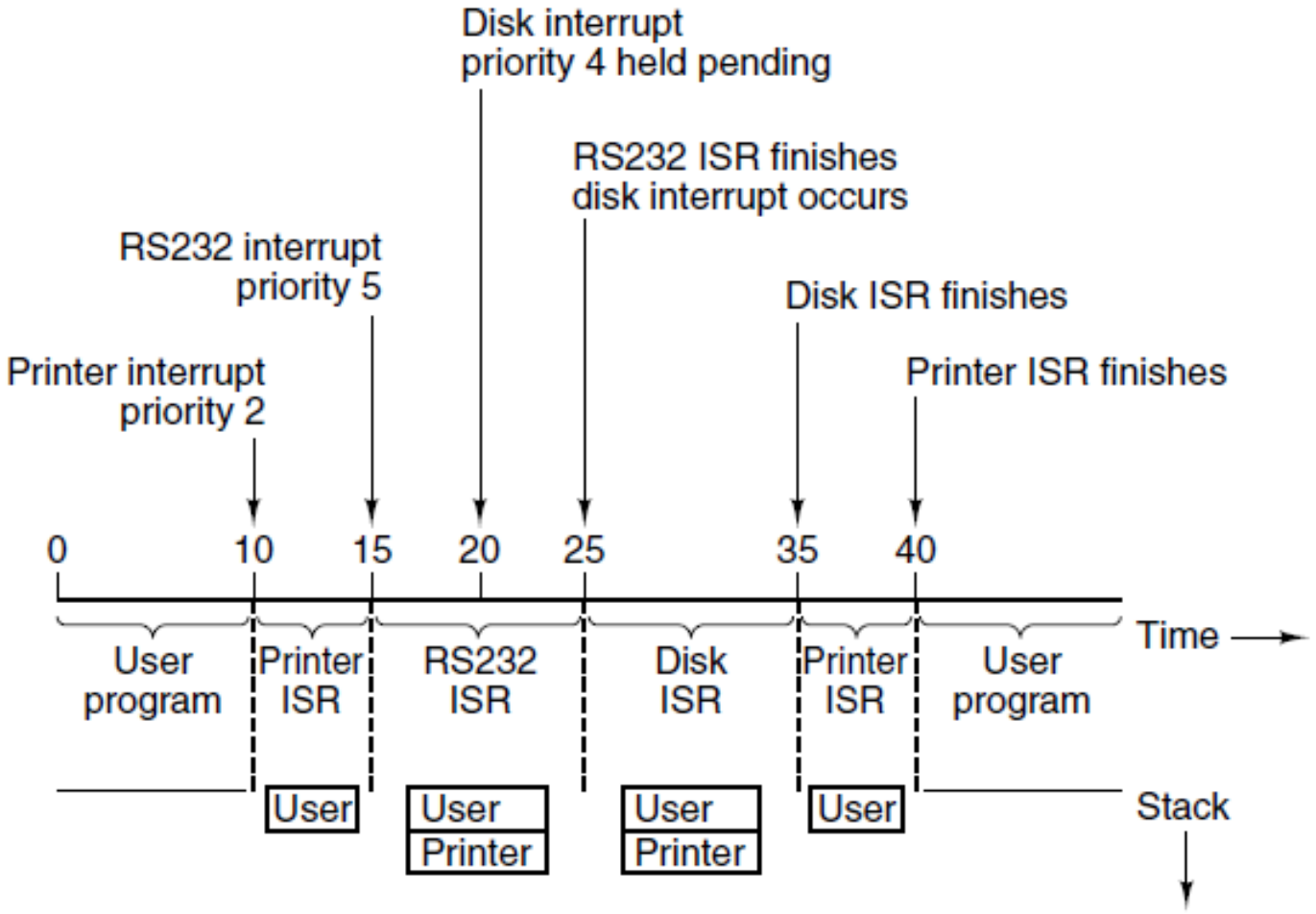



A
Calling procedure
B
Called procedure
A called from main program
CALL
CALL
RETURN
RETURN
CALL
RETURN
A returns to main program

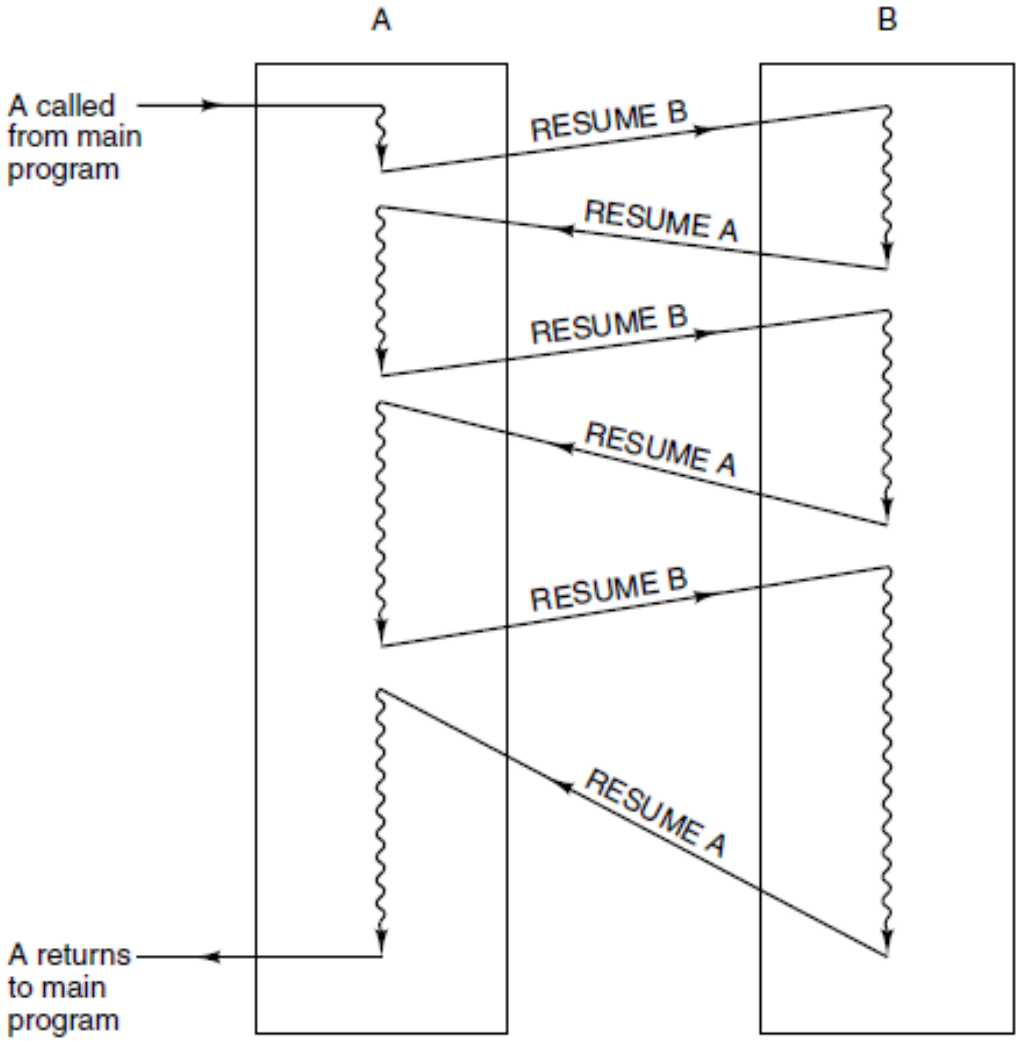
Interrupt Service Routines (ISR)

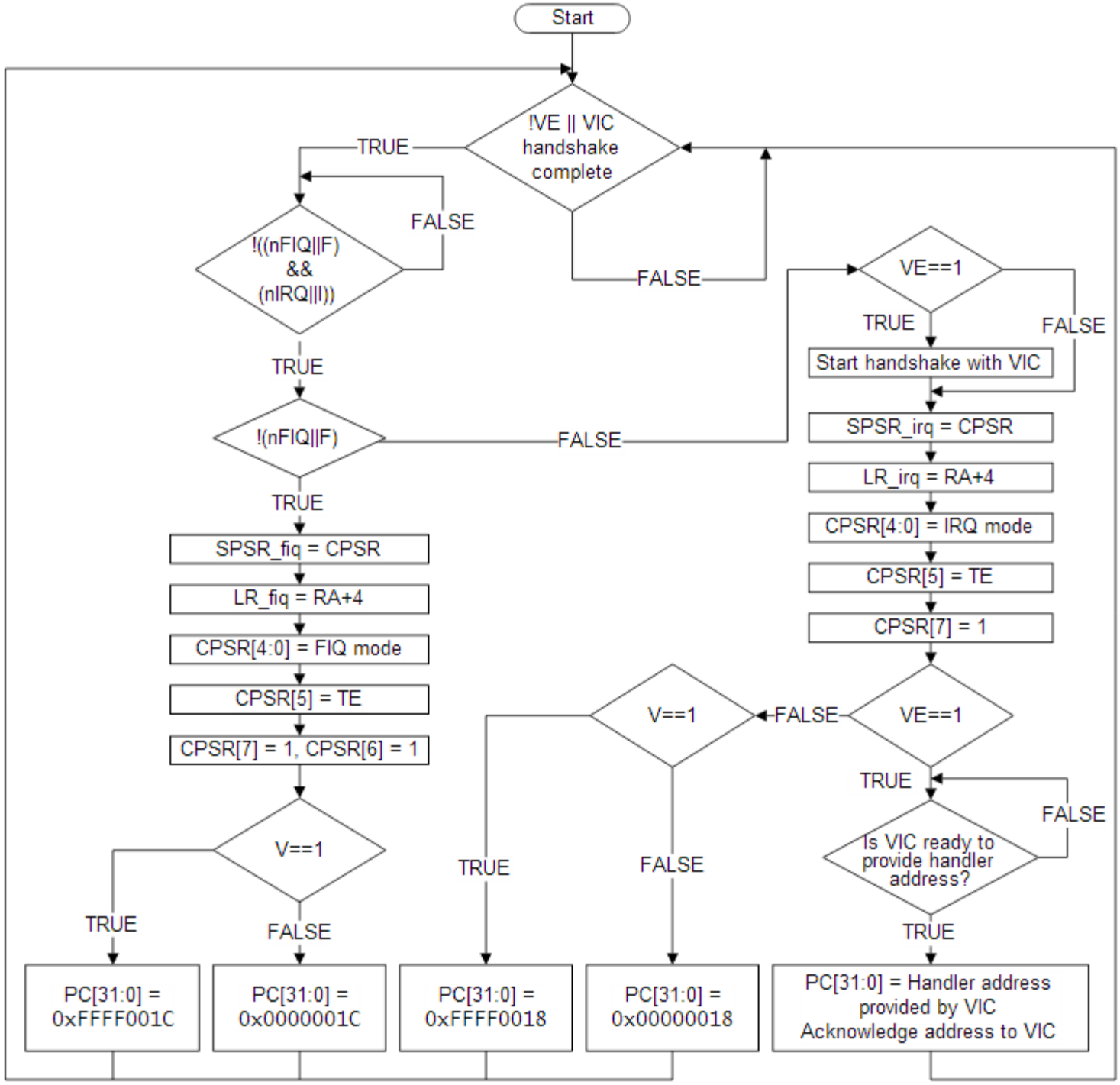
- Coroutine: Procedure that resumes execution at previous point
- Special routines that may interrupt other routines
  - Interrupt: stop execution of procedure A and start execution of procedure B
  - Useful for I/O: suppose we have data coming from the keyboard
  - Priorities: if we have several ISRs, how do we decide which gets executed now?
- Interrupt handler (or ISR): PC gets loaded with address of ISR
- ISR specified in an interrupt vector table

ISR Timeline



ISRs and Coroutines




Start
!VE || VIC handshake complete
TRUE
FALSE
!((nFIQ||F) && (nIRQ|||))
FALSE
TRUE
!(nFIQ||F)
FALSE
TRUE
SPSR_fiq = CPSR
LR_fiq = RA+4
CPSR[4:0] = FIQ mode
CPSR[5] = TE
CPSR[7] = 1, CPSR[6] = 1
V==1
TRUE
FALSE
PC[31:0] = 0xFFFF001C
PC[31:0] = 0x0000001C
VE==1
TRUE
FALSE
Start handshake with VIC
SPSR_irq = CPSR
LR_irq = RA+4
CPSR[4:0] = IRQ mode
CPSR[5] = TE
CPSR[7] = 1
VE==1
FALSE
V==1
TRUE
FALSE
PC[31:0] = 0xFFFF0018
PC[31:0] = 0x00000018
TRUE
FALSE
Is VIC ready to provide handler address?
TRUE
PC[31:0] = Handler address provided by VIC
Acknowledge address to VIC

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception (trap)
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, divide by zero, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

Traps / Exception

- Automatic procedure call initiated by some exception condition caused by program
- Examples:
  - Overflow (arithmetic, stack)
  - Divide by zero
  - Undefined opcodes
- Trap handler: procedure that takes care of exception condition
- If exception detected, PC loaded with exception handler address

Summary

- Input/Output
  - Memory-mapped I/O: I/O devices have registers, and they're mapped to the CPU accessible memory
  - UART input
  - UART output
- Interrupts and Exceptions