Operating Systems

Xinu – Hardware and Runtime Environment

1

Processors and Systems

- The textbook describes two architectures – The ARM-based BeagleBone Black and the Intel Galileo
  - This covers both the ARM and x86 Instruction Set Architectures (ISA)
  - ARM is a Reduced Instruction Set Computing (RISC) ISA, whereas x86 is referred to as a Complex Instruction Set Computing (CISC) ISA
    - The previous edition of the book describes the Linksys E2100L home router which uses the MIPS ISA
- We will focus on the BeagleBone, which is based on a system on a chip (SoC) from TI, the AM335x Sitara
  - The Raspberry PI system is based on a Broadcom BCM2835 which includes an ARM1176JZF-S processor

2

Registers

- High speed memory used by the processor
- The compiler generates code to load (fetch) and store to memory, and operates on values in registers
- Some are general purpose, and some have a purpose (by convention only in some cases)
- Hold part of the state of the computation and thus must be saved and restored when switching processes

3

ARM Processor - Registers

- ARM registers are banked (or windowed) such that some registers are renamed when the mode changes (in e.g. interrupts)
  - r0 – r6 are visible across modes

Name	Use	Notes
r0-r3	General Purpose	Argument registers for 1st four args
r4-r12	General Purpose	r4-r11 are "callee-save"; r12 = ip
r13	Stack Pointer	by procedure calling convention
r14	Link Register	by procedure calling convention
r15	Program Counter	
cpsr	Current Program Status Register	spsr is the Saved Program Status Register

4

ARM Program Status Register (CPSR)

- The CPSR includes
  - Processor mode (0-4)
  - Interrupt disable (7)
  - Fast interrupt disable (6)
  - Condition code flags
    - **N**egative result from CPU logic (31)
    - **Z**ero result from CPU logic (30)
    - ALU operation **C**arried out (29)
    - ALU operation o**V**erflowed (28)

Intel Registers

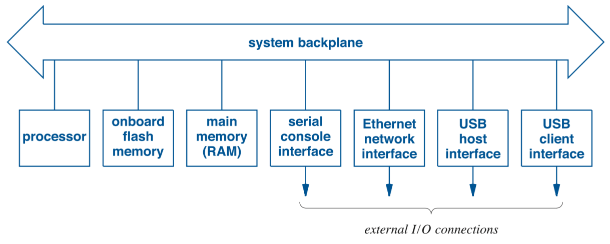
Name	Use
EAX	Accumulator
EBX	Base
ECX	Count
EDX	Data
ESI	Source Index
EDI	Destination Index
EBP	Base Pointer
ESP	Stack Pointer

Also a Program Counter (PC) register called the Instruction Pointer (RIP)

The Bus

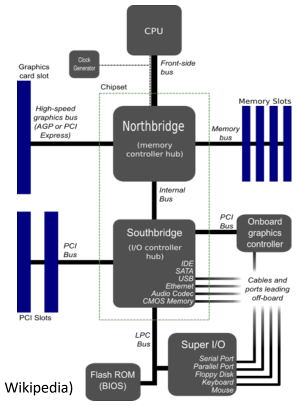
- The bus is for communication with peripherals and memory
  - Load (Fetch) / Store operation
- The CPU places an address on the bus, and signals ready
- The appropriate target (e.g. memory bank) retrieves the value from the requested address and puts it on the bus
- In memory-mapped I/O – devices appear as “memory” with particular addresses
  - reading and writing to a device uses fetch/store in the same way

System Overview



More Modern Architecture

- Device Controllers are connected or integrated in various locations
  - PCI-e
  - PCI
- Location determines the speed of access to memory

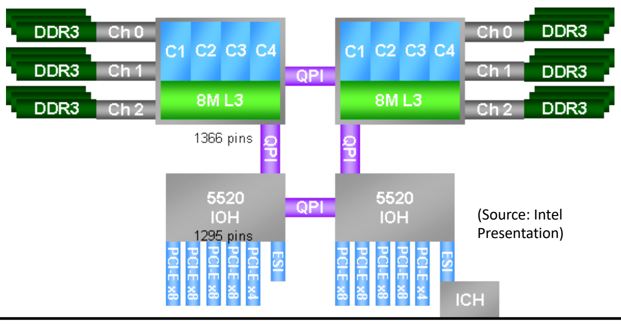


(Source: Wikipedia)

9

More Modern Architecture (NUMA)

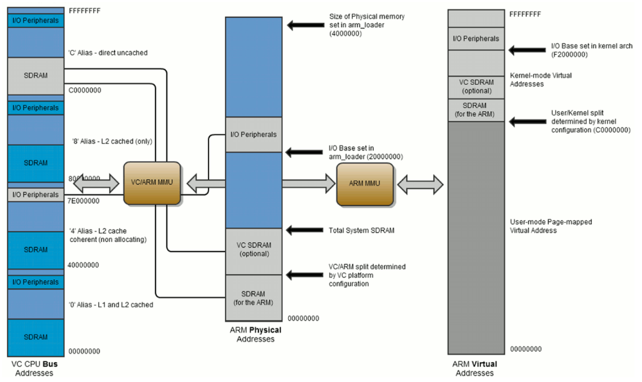
- No longer a single memory controller
- Symmetric Multiprocessor with Non-uniform Memory Access (NUMA)



(Source: Intel Presentation)

10

BCM2835 (Raspberry PI)


FFFFFFFF
I/O Peripherals
'C' Alias - direct uncached
SDRAM
C0000000
I/O Peripherals
'8' Alias - L2 cached (only)
SDRAM
80000000
I/O Peripherals
7E000000
'4' Alias - L2 cache coherent (non allocating)
SDRAM
40000000
I/O Peripherals
'0' Alias - L1 and L2 cached
SDRAM
00000000
VC CPU Bus Addresses
VC/ARM MMU
I/O Peripherals
VC SDRAM (optional)
SDRAM (for the ARM)
00000000
ARM Physical Addresses
Size of Physical memory set in arm_loader (4000000)
I/O Base set in arm_loader (20000000)
Total System SDRAM
VC/ARM split determined by VC platform configuration
ARM MMU
FFFFFFFF
I/O Peripherals
I/O Base set in kernel arch (F2000000)
VC SDRAM (optional)
Kernel-mode Virtual Addresses
SDRAM (for the ARM)
User/Kernel split determined by kernel configuration (C0000000)
User-mode Page-mapped Virtual Address
00000000
ARM Virtual Addresses

11

ISA Implications for the OS

- The compiler generates machine code from C language source
- There are a few places that we need to consider the system details
  - Hardware Abstraction Layer (HAL) in Windows
- Mostly in startup, context switching and interrupt handling as we will see

12

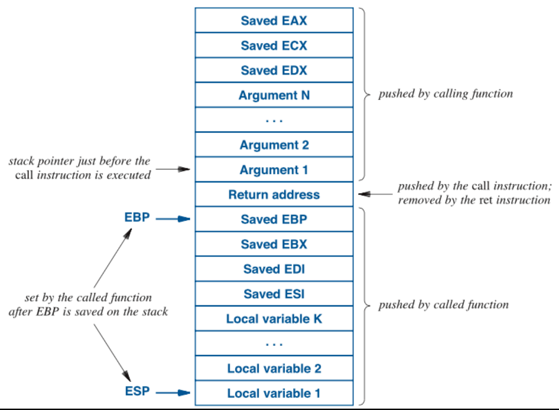
Stack: Function Calling Conventions

- Function invocation is central to OS operation
- Calling conventions refers to what happens when a function is called and when it returns
  - Conventions in that it is really decided by the compiler, and it simply needs to be consistent
- The run-time stack stores state associated with a function call
  - with addresses growing downward (generally)
- There is space in this stack frame (or activation record) that includes
  - local variables
  - return address
  - arguments
  - return values

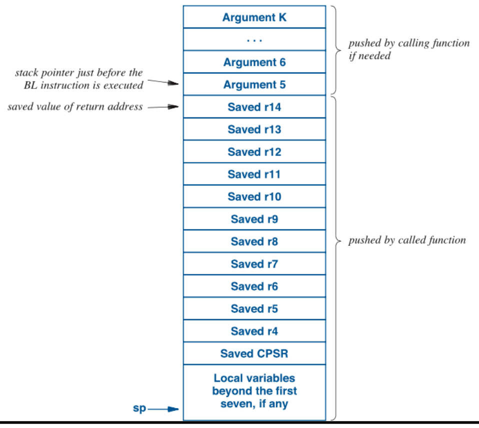
Stack: Function Calling

- Arguments / Parameters
  - Fixed number of arguments are passed in registers (r0-r3) and the remainder are passed via memory on the stack
- The compiler computes the size needed for the stack frame
  - This is why you need a function prototype accessible when the calling code is compiled
- Argument save area in stack frame for function X is used by function Y

x86 Stack Layout


Saved EAX
Saved ECX
Saved EDX
Argument N
. . .
Argument 2
Argument 1
Return address
Saved EBP
Saved EBX
Saved EDI
Saved ESI
Local variable K
. . .
Local variable 2
Local variable 1
pushed by calling function
pushed by the call instruction; removed by the ret instruction
pushed by called function
stack pointer just before the call instruction is executed
EBP
set by the called function after EBP is saved on the stack
ESP

ARM Stack


Argument K
. . .
Argument 6
Argument 5
Saved r14
Saved r13
Saved r12
Saved r11
Saved r10
Saved r9
Saved r8
Saved r7
Saved r6
Saved r5
Saved r4
Saved CPSR
Local variables beyond the first seven, if any
pushed by calling function if needed
pushed by called function
stack pointer just before the BL instruction is executed
saved value of return address
sp

Interrupts

- As discussed, I/O devices interrupt the system
  - Similar handling for other *exceptions* like divide by zero or page fault
- I/O interrupts are generated by placing a signal on the bus (control lines)
  - Monitored by the processor or a co-processor
- When an interrupt occurs
  - Record the address of the next instruction
  - Jump to memory location 0x18
  - Set a control bit in cpsr to disable further interrupts

Interrupt Processing

- The interrupt controller has registers for interrupt masks
  - Controlling what interrupts can and cannot occur
  - There is also a bit to disable all of them
  - Xinu uses this when manipulating internal structures
  - Functions disable() and restore() to save and restore the mask
- At startup, an instruction must be stored at 0x18 (on ARM) for servicing interrupts

Timer Hardware

- Timer device works similarly to I/O
  - When the timer expires, it generates an interrupt
  - Once the timer has been set, the OS must be ready to handle interrupts
- The timer is programmable with two registers
  - Counter – initial value
  - Limit – the value at which to interrupt (the length of time)
- An alternative is a real-time clock that interrupts with a fixed period (60x / second)

Direct Memory Access - DMA

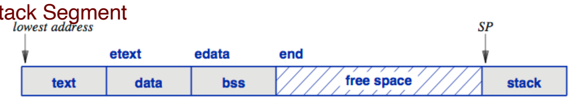
- Hardware devices communicating directly with memory
  - Rather than the processor reading from the bus, then writing to the bus…
  - The I/O device can perform a series of operations while the processor runs code
- Network device example
  - The OS allocates a list of packet buffers and starts the network device
  - Packet arrives, device transfers the complete packet into memory, then interrupts

Serial Device

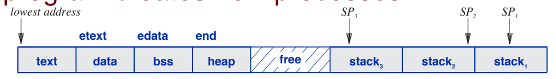
- Serial devices are the simplest I/O device
- Universal Asynchronous Receiver Transmitter (UART)
- Handles both input and output
  - Called transmit (TX) and receive (RX)
- When an interrupt occurs, the system must examine a device register
  - Byte(s) received
  - Byte(s) transmission complete

Memory Layout

- When a program is compiled the compiler generates partitions of memory segments
  - Text segment
  - Data segment
  - BSS segment
  - Stack Segment



- As the program creates new processes:


lowest address
SP$_3$
SP$_2$
SP$_1$
etext
edata
end
text
data
bss
heap
free
stack$_3$
stack$_2$
stack$_1$