CPSC 2310 Notes

8/26

Major Concepts:

Data Structure Model:

Basic representation of data such as integers, logical values, and chars

Homogeneous data structure - structures containing data of the same type

A string is homogeneous with the char type

Heterogeneous data structures, such as floating point numbers and records

Floating point binary representation

1 bit - the sign (This is a logical flag)

8 bit - exponent (This is a binary int)

23 bit - mantissa (mantissa is a representation of significant digits.

Gets multiplied by the (base raised to the exponent) to give the

actual number

+__________

32 bits

Accessing data structures, involving pointers and values, field extraction

Addressing mode support for particular data structures

Machine model:

Processor, memory, and I/O

Processor data paths and register transfers

Instruction cycle of fetch/execute and interrupts

Allows for instructions to be interrupted or overridden in favor of other

procedures

Bus interconnections(physical connection)
Think USB - Universal Serial Bus
Relative speeds and capacities of components

Performance metrics, such as storage space, memory traffic, execution time
How everything works together, things like how frequently cache should
be accessed or how often virtual memory should be held

Major Concepts Continued

A programming language translation model:

- Compilation versus interpretation
  - Compilation - change source code to an executable - all at once
  - Interpreter - takes a HLL to machine language line by line, INTERPRETING each instruction
- Steps: macro processing, compilation, assembly, link, load, execute
  All “high-level” source code is changed to machine instructions
  The language it is changed to is called Assembly
  Assembler is a translator will take source and translate into machine language (binary numbers containing the instructions)
  Files must be linked in the case of multiple source files, and is done with a linker
  IE collect object files to include in an executable file
  Also links in libraries
- Code generation issues demonstrating the relationship between HLL and machine code.

A system model:

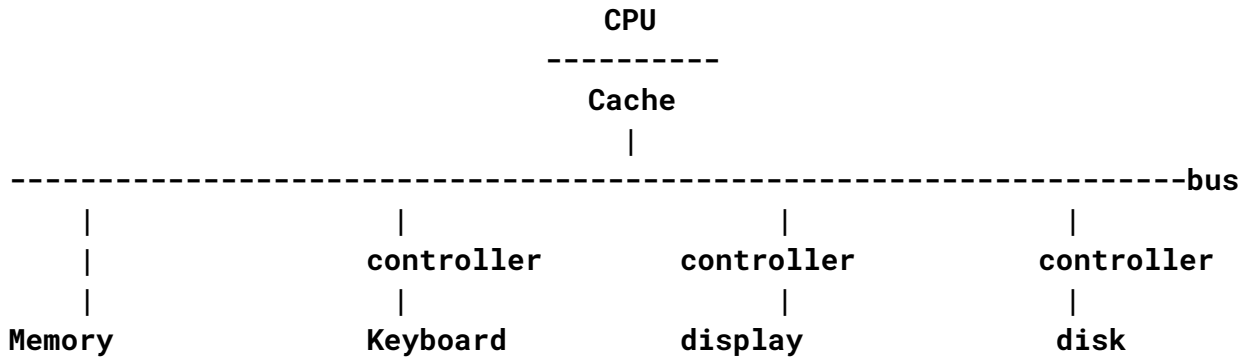
- run -time environment, including stack frame and heap support
  - Stack - automatically allocated memory
  - Heap - holds allocated data (malloc, calloc, free [c] / new and delete (c++))

- Procedure call support, including such issues as parameter passing and register conventions
- I/O support, such as level of intelligence in controllers and synchronization
- Operating system support, including memory protection and interrupts
- Sharing and reentrancy
- Virtual memory support

Computer Systems:

- Increasingly smaller
- Higher performance
    - 7-9x more powerful relative to early computers speed
- More memory
    - Ram measured in megabytes -> Gigabytes
- Lower power
    - Battery operable
- Embedded
    - A machine within another machine that serves a dedicated purpose
        - The microchips and computers in your car are embedded systems
- Everywhere
- Lower cost
- … but extremely complex
    - He mentioned transistor - it is like a logical gate, when allowed, the “gate” opens and allows power to flow into the circuit. Has a binary “open” or “close”, and the on/off state dictates whether power can flow past the transistor

Here is a diagram of a simple computer System: This diagram will be the one needed for exams!!!!!!!!!!!



REQUIRED TO MEMORIZE ^^^^

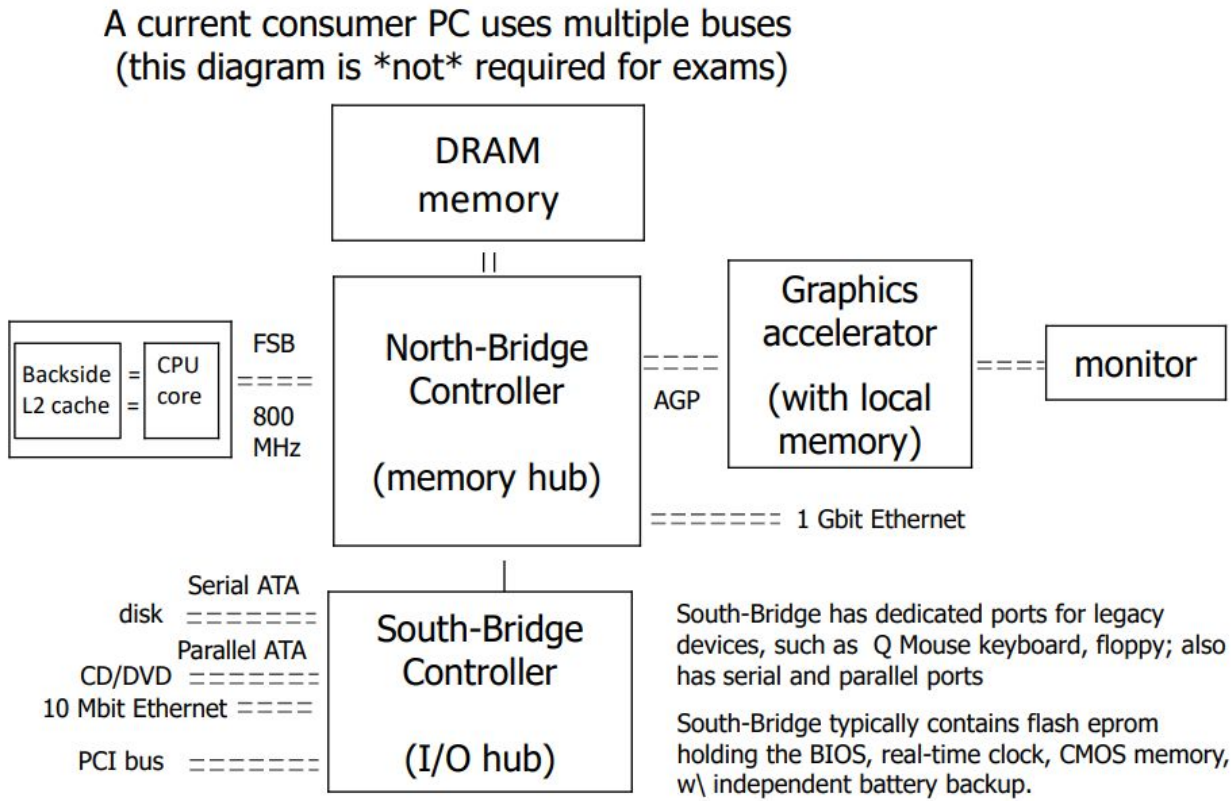
Bus notes

The cpu connects to the memory hub with the “Front Side Bus” and is often the highest

throughput bus

NOT REQUIRED TO MEMORIZE \I/

Computer Systems



If CPU cannot find what it's looking for it will check Level 2 Backside cach

Southbridge is usually lower speed devices and legacy devices (IE serial ports/floppy disk)

The big idea in the more complicated bus design is to pyramid from high speed busses to lower speed busses (IE one northbridge branches to many slower, less important busses)

Computer Components:

Input - keyboard, mouse, scanner(appliance, not java object), etc..
- Connected via a controller. CPU ---> Controller ---> Peripheral device

Output - display, printer, sound, etc..

Disks can be input or output, whether or not you're writing to, or reading from

CPU == central processing unit == processor composed of two parts:

Data Path - (temp memory called registers, and function units)

- Consists of small pieces of memory that hold operands for operations, as well as result of functions

Control logic - (sequencing of datapath actions)

IE Data path is what to do

Control logic is when/how to do it

Fetch an instruction from memory to register, read to decode instruction

Different instruction sets:

Intel IA32 (x86), Apple/IBM/Motorola PowerPC, Sun SParc, ARM…

https://www.amd.com/system/files/TechDocs/24592.pdf

^ AMD64 Architecture programmer's manual

Common instructions include add, subtract, jump, …

Emulator - a translator that changes instructions from source instruction set to the instruction set that the local machine will use (IE qemu will translate ARM instructions to IntelIA32 (x86) instructions

Memory:

multilevel hierarchy due to cost vs. speed tradeoffs

fastest and most expensive -->

- CPU registers
- cache (perhaps multiple levels)
- Recently accessed memory - fast access
- main memory - cpu uses to access mem for active progs
- This mem is still cpu adjustable CPU registers lie directly on the datapath

slowest and least expensive -> long-term storage

^- This is why computers have different RAM and storage.

Also why Hard drive cant be used for RAM (not fast enough)

cache and main memory are made of RAM - random access memory
RAM - memory that takes a uniform time to access regardless of location in memory

DRAM - dynamic RAM - most main memories
- One of two ways of physically managing RAM
- Cheaper to manufacture than SRAM, also higher density
- Allows for higher capacities
- Must be refreshed in regular interval, slows access time (10-20 nanoseconds)
- Drawback: Dynamic RAM data is stored as a charge on a capacitor. Uncharged capacitors are decharging. Transistors need to be refreshed in a rolling manner.

SRAM - static RAM - fast and expensive, used for caches (1-2 nanosec access time)
- Each memory cell has 2 logic gates with a feedback loop, in order to maintain a bit value. Lower density = lower capacity
  - The feedback loop means that writing/reading is *nearly* instantaneous
    No need to refresh capacitors like with Dynamic ram, which makes them much faster

WHEN SHUT DOWN, DRAM and SRAM loose their data

ROM - read-only memory (holds initial "bootstrap“ loader program and basic I/O programs)
- Cpu has no way to access the disk, thus memory is loaded into ROM in order for unchanging programs (IE bootstrap, operating system) to be accessed without power.
- Disk memory is non-volatile

long-term storage - so slow that it is treated as I/O:
(Processor has to talk to I/O controller, which handles transfer of data)
-floppy disk
-hard disk
-CD-ROM
-DVD

Prefixes for speed, time, and capacity

K (kilo-) = one thousand = 10^3 ~ 2^{10}	m (milli-) = 10^{-3}
M (mega-) = one million = 10^6 ~ 2^{20}	u (micro-) = 10^{-6}
G (giga-) = one billion = 10^9 ~ 2^{30}	n (nano-) = 10^{-9}
T (tera-) = one trillion = 10^{12} ~ 2^{40}	p (pico-) = 10^{-12}
P (peta-) = one quadrillon = 10^{15} ~ 2^{50}	f (femto-) = 10^{-15}
E (exa-) = one quintillion = 10^{18} ~ 2^{60}	a (atto-) = 10^{-18}

SMALL: a,f,p,n,u,m,K,M,G,T,P,E :BIG

Main memory is measured in powers of 2
Speed is measured in powers of 10

The capacity of most hard disks is measured in powers of 10

HLL		assembly lang		machine code(object file or executable)
A = B + C;	-->	load(B)	-->	0000 0010 0010 0100
		add(C)		0000 0001 0010 0101
		store(A)		0000 0011 0010 0011

High Level Language -> Assembly Language (one to one with binary approx) -> machine instructions (this is the binary)

Each step is one increment closer to the bit representation needed by hardware for execution

Compiler - Takes high level language to assembly code. Makes optimization

Interpreter - translates and executes AT THE SAME TIME (also no optimization)

Assembler - takes assembly to make into machine language

Assembly Language - a line of code is called an instruction
    Operation code (which operation)
    Operands (what is being operated on)

    Has symbolic names like “loop”, theyre called labels
    Each label must be unique

Assembler - have a 2 pass structure
    Instructions have forward or backward references into labels

    Because of forward references, most assemblers use 2 pass structure to prevent
    use before something is defined

8/28

Accumulator Machine
- Has one processor register - the accumulator
- All other operands are in memory, and expressions require a sequence of load/operate/store instructions
    - Load puts a value in memory in the register
    - Operate effects data in the register
    - Store puts a value back into memory

accumulator machine instruction set

All instructions except halt have an address field for the parameters

opcode	address	operation name	machine action
halt	----	halt	stop execution
div	addr	divide	acc = acc/memory[addr]
mul	addr	multiply	acc = acc*memory[addr]
sub	addr	subtract	acc = acc-memory[addr]
add	addr	add	acc = acc+memory[addr]
load	addr	load	acc = memory[addr]
store	addr	store	memory[addr] = acc
ba	addr	branch always	pc = addr
blt0	addr	branch on less than	if acc<0 then pc = addr
ble0	addr	branch on less than or equal	if acc<=0 then pc = addr
beq0	addr	branch on equal	if acc==0 then pc = addr
bne0	addr	branch on not equal	if acc/=0 then pc = addr
bge0	addr	branch on greater than or equal	if acc>=0 then pc = addr
bgt0	addr	branch on greater than	if acc>0 then pc = addr
print	addr	print	display contents of memory[addr]

Assembly language psuedo-ops(Assembly language just ignores them):

- Comment, End, Label, Word

assembly language pseudo-ops

name	arg1	arg2	meaning
comment	text		allows for comments within program
end	symbol		defines starting address of program (should appear once, as the last line of the program)
label	symbol		defines a symbolic address within the program
word	symbol	value	defines a symbol address and allocates a data word at that address with the given value

The end instruction (w corresponding symbol) will place the location of the first instruction and then the program will look at the last numeric value and use that for the address to jump to to begin operations. IE it tells the program (after it has gone through the code once to make machine instructions) where those instructions begin.

Accumulator machine program - example 1

```
comment(` first example accumulator machine program ')
comment(` ')
comment(` data section for program -- word(label,value) ')
    word(a, 23)
    word(b, 45)
    word(c, 17)
    word(d, 0)
comment(`code that implements the expression d = a + b - c;')
label(start)
    load(a)   comment(` ACC <- memory[a] ')
    add(b)    comment(` ACC <- ACC + memory[b] ')
    sub(c)    comment(` ACC <- ACC - memory[c] ')
    store(d)  comment(` memory[d] <- ACC ')
    halt
comment(` start execution at label start ')
    end(start)
```

- Executable is in numeric format-- example assembly (numbers are in decimal rather than binary)

- The assembler has a location counter variable called “loc” that it uses in the first pass to assign addresses

instruction table (built into assembler)

name	opcode	args	loc increment	
add	40	$1	2	<- add is defined as two words
sub	30	$1	2	* first word is opcode 40
load	50	$1	2	* second word is address of arg
store	60	$1	2	* loc should be incremented by 2
halt	00		1	

Loc Increment moves the memory location past the data used by the function call

Pass 1

first pass input --

assembly stmts	label	addr==loc	
word(a,23)	a	00	<- a is defined as location 00
word(b,45)	b	01	<- b is defined as location 01
word(c,17)	c	02	<- c is defined as location 02
word(d,0)	d	03	<- d is defined as location 03
label(start)	start	04	<- start defined as location 04

symbol table after first pass

...

<< no other statements define labels >>

...

Any instruction with a value in parentheses will increment the location counter by 2

Pass 2

second pass input		"executable" after second pass		
(loc:)	assembly language	addr:	machine code or data	
(00:)	word(a,23)	00:	23	<- value in location a
(01:)	word(b,45)	01:	45	<- value in location b
(02:)	word(c,17)	02:	17	<- value in location c
(03:)	word(d,0)	03:	0	<- value in location d
	label(start)			
(04:)	load(a)	04:	50	<- opcode for load
		05:	00	<- address of a
(06:)	add(b)	06:	40	<- opcode for add
		07:	01	<- address of b
(08:)	sub(c)	08:	30	<- opcode for sub
		09:	02	<- address of c
(10:)	store(d)	10:	60	<- opcode for store
		11:	03	<- address of d
(12:)	halt	12:	00	<- opcode for halt
(13:)	end(start)	13:	04	<- starting address

In memory the op code comes before the value to be used for operation

9/02

Control Structures in Assembly

- Uses branch construction
  - Avoid Back to back branching - its inefficient and repeats based on the same logical flags.

- IE
  - load(x)
  - bne0 do_nothing (instead of beq0)
  - ^cmd, be not equal to 0
    - ^do_nothing is a label for a forward branch
      - ^using beq0 would require all loop instructions To be listed below the “do_nothing” label

- Opposite relational operators

Logical opposites

Operator	Logical Opposite
==	!=
!=	==
<	>=
<=	>
>	<=
>=	<

if/else conditions

If one branch is not taken, it must take the other branch

IE

```
        beq0(then_part)
label(else_part)
        Load…
        Sub..
        ba(done)
label(then_part)
        Load…
        Add…
label(done)
        store...
```

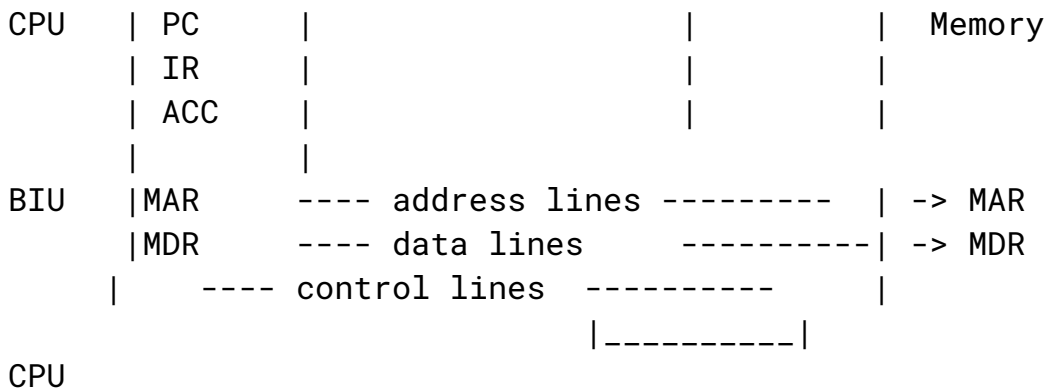
If you have a forward branch evaluates true, have it go through code sequentially
If you have a forward branch evaluate false, then it may skip to the forward branch

Backward Branch -
A branch that will cause a loop to complete

```
load(x)
sub(10)
bgt0(done) //this condition, if true, will skip forward to done label
label(loop)
sub(ten)
bgt0(done) // check condition to exit loop
add(y)
store(x)
ba(loop) //if loop not exited, repeat loop
label(done)
… //more code
```

For loop in Assembly

```
label(loop)          |
sub(y)               |
bgt0(done)           | test condition
load(z)              |
mul(x)                    |
store(z)             | loop body
load(x)                   |
add(one)                  |
Store x                   |
ba(loop)                  |update loop control variable
label(done)
                          ------------
```



CPU

- PC(program counter) - points to next instruction
- IR (instruction register) - holds current instruction

BIU (bus interface unit)

MAR - (Memory Address Register) - a CPU register that stores the memory addresses from which data will be fetched to the CPU or the address to which data will be sent and stored

MDR (memory data register) - a CPU register that contains the data to operate on.

Machine And Von-Neumann Architecture

Prior to von neumann - there was separate, specialized memory for data or program instructions
Von neumann is credited with the idea of the “stored program” computers, where the program is stored in the same memory as the data

Von Neumann Machines Characteristics

- Random-access, one-dimensional memory(vs. Sequential memories)
  Sequential memory Ie disk/tape

- Sequential processing (NO PIPELINING)
  Instructions start only after the previous instruction is finished

- Stored program, no hardware distinction between instructions and data vs
  Harvard architecture - separate data/instruction memory

- Binary, parallel by word, two's complement (vs. decimal, serial-by-digit,
  sign-magnitude)
  A word is essentially the size of data in number of bits that an op works
  with
  Hardware uses 2s compliment to represent if an int is pos or neg
  Signed integers can be pos or neg
- Instruction fetch/execute cycle, branch by explicit change of PC (vs. following a
  link address from one instruction to the next

- Three register arithmetic -- ACC, MQ, MBR
  - Accumulator
  - Multiplication/Quotient register (mult/divis register)

Von Neumann Characteristics

Memory contains series of bits grouped into addressable units

Note: binary devices are relatively easy to build - positive feedback drives the device to one of two extreme states and keeps it there, so the decimal storage devices of some old machines were usually four binary devices for which only ten of

the sixteen states were used - Called “binary coded decimal” or BCD

Data is accessed in memory by naming memory addresses
    Its like an array kinda

Addresses are consecutive binary integer values (unsigned)
    This is convenient for addressing arrays and sequentially stepping through a prog

Bit strings have no inherent data type - can be integer, char string, machine instruction, or just garbage => HARDWARE DOESN'T KNOW DATA TYPE
        The program requesting the information knows how to interpret it for use
        Its also what lets you use integer 1 as true in C/C++

Tags - a few old computers tried to type fields with tags added to memory words in order to distinguish between data and instruction types

Type checking - compiler checks that all data types are agreeable and prog can run
Notes:
    Strongly typed languages dont allow implicit casting
    In strong and statically typed language, mismatch is a compiler error
    In a strong/dynamically typed language, the interpreter or other runtime system declares a run time error when types dont match
    Because C is generous in using implicit conversions, it is considered
        weakly-typed with static (compile-time) type checking

CONSIDER

```
Int main(){
Int i ;
Char c = ‘a’;
printf(“%c 0x%x %d\n”,c,c,c);
i = c + 1;
printf(“%c 0x%x %d\n” i,i,i);
Return 0;}
```

Output:
a 0x61 or 97
b 0x62 or 98

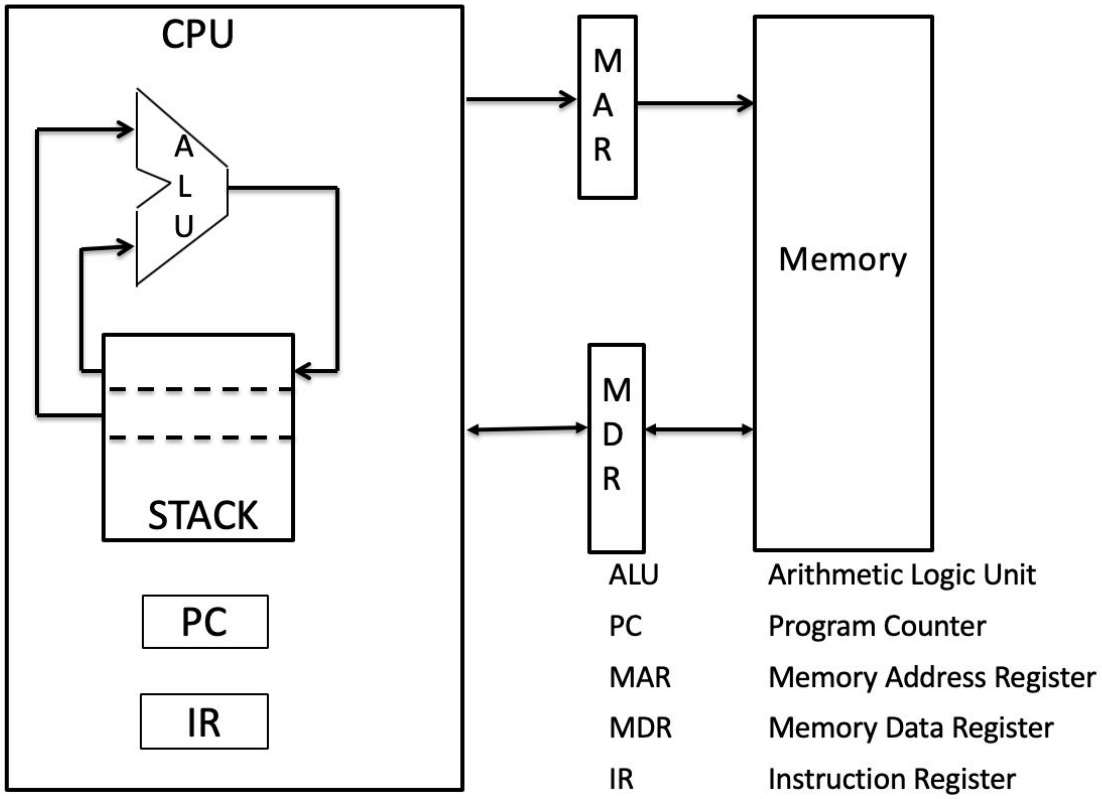
A compiler typically type-checks all the variables it uses so that all operations are legal
NOTES:

- A strongly-typed language allows few, if any, implicit type conversions

Varieties of processors

1 - accumulator
2 - stack machine - operrands on a stack and push/pop from memory
3 - load/store machine - computational instructions operate on general registers
and load/store instructions are the only ones that access memory

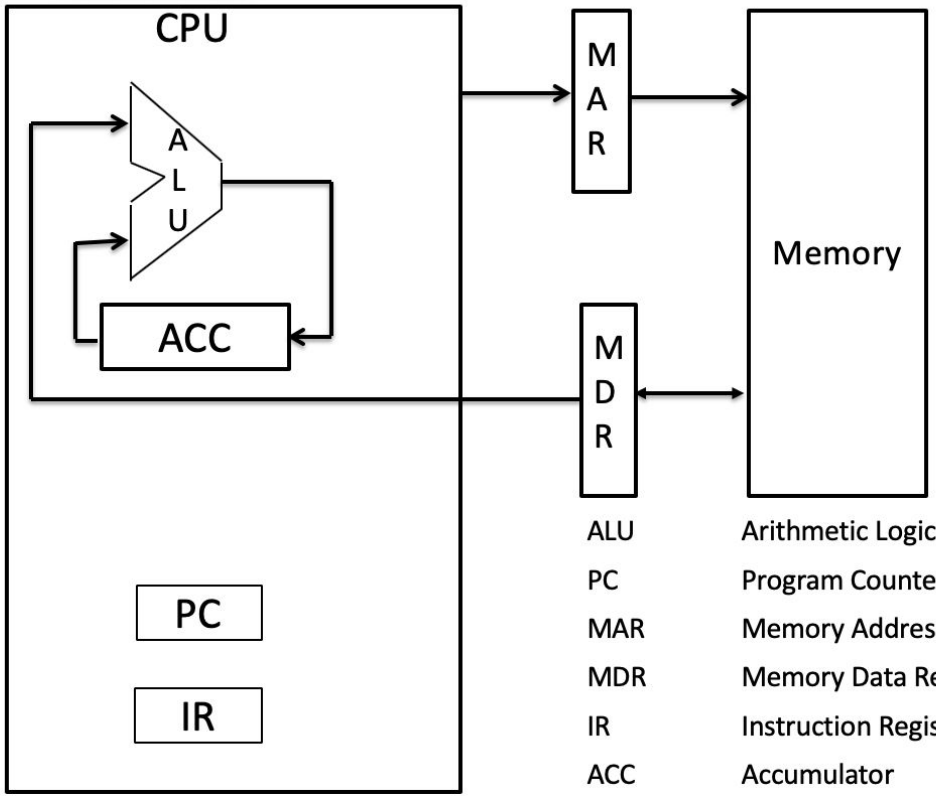
Stack Machine



ALU	Arithmetic Logic Unit
PC	Program Counter
MAR	Memory Address Register
MDR	Memory Data Register
IR	Instruction Register

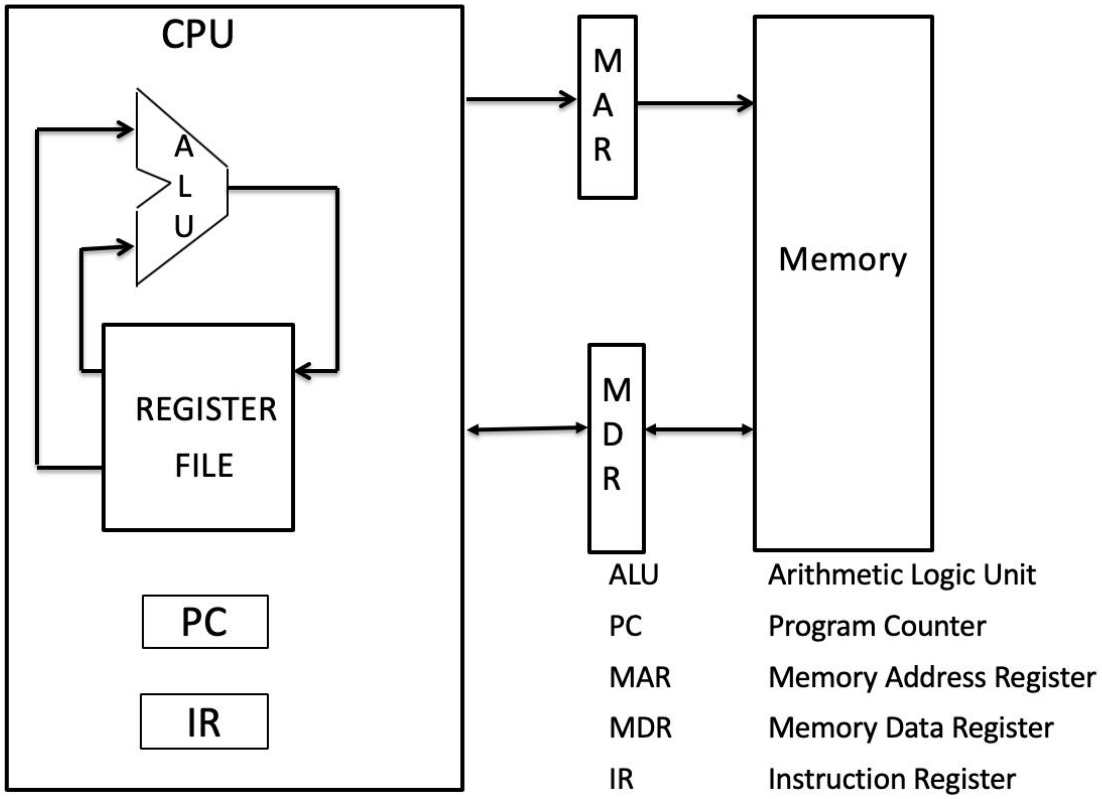
Accumulator Machine

Draw a Text Box



ALU	Arithmetic Logic Unit
PC	Program Counter
MAR	Memory Address Register
MDR	Memory Data Register
IR	Instruction Register
ACC	Accumulator

Load/Store Machine



ALU	Arithmetic Logic Unit
PC	Program Counter
MAR	Memory Address Register
MDR	Memory Data Register
IR	Instruction Register

9/04

Varieties of Processors

Arm assembly uses the load/store architecture

Common parts(3) across CPU types

Data path - general purpose data registers

Bus Interface Units

Control unit - PC and IR

Data Path

- Data registers such as ACC or set of general registers
- Address registers such as SP (stack pointer) and Xn (index registers)
- ALU (arithmetic logic unit) executes operations
- Internal buses (transfer paths) Internal Busses connect everything together a.k.a datapath

Bus Interface Units (BIU or memory bus interface)

- MAR - mem address register - a CPU reg that either stores mem addresses from which data will be fetched to the cpu or the address to which data will be sent and stored
- MDR (memory data register -- perhaps better called memory_buffer_reg) - a CPU register that contains the data to be stored in memory or the data after a fetch from memory.
  - Essentially the intermediate point where data is registered

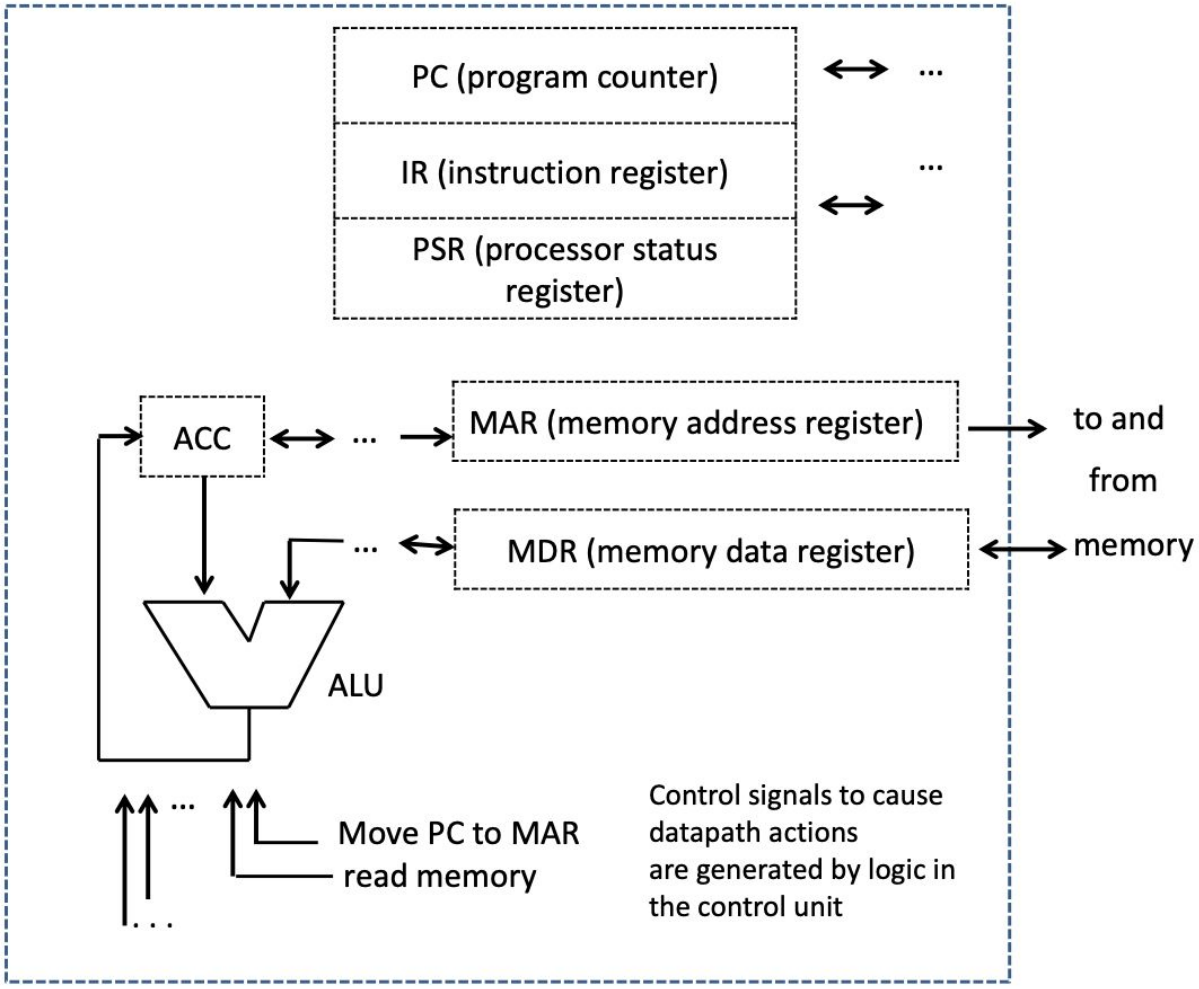
Control Unit - Program Counter and Instruction Register

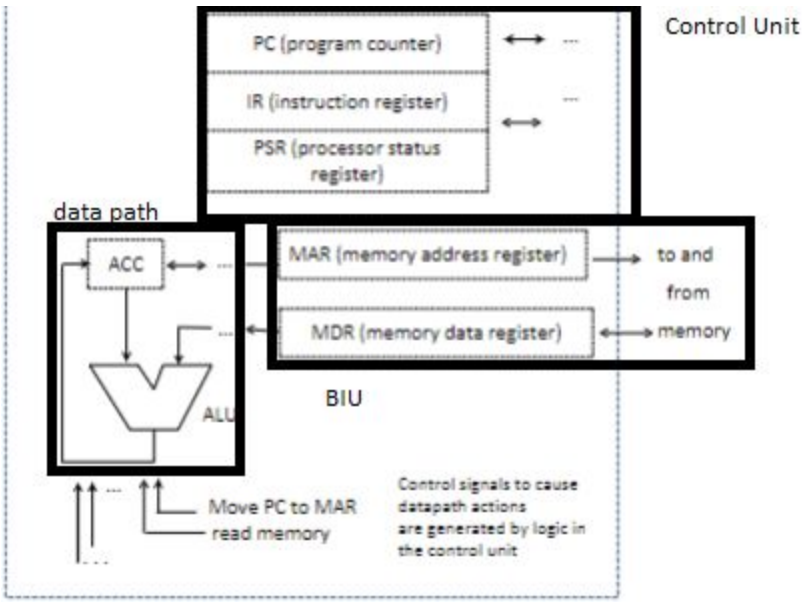
- PC(program counter) - points to next instruction(x86 calls it IP == instruction pointer) - determines when a loc in mem holds instr
- IR (instruction register) - holds current instruction - holds while being executed
- PSR (processor status register) - indicates results of previous operation (called condition codes or flags) - without this, logical branching and control flow are able

to be used and manipulated - holds all condition codes or flags

Kernel mode - a mode which allows more control, typically allowed only for the operating systems

Closer view of CPU - central processing unit





Fetch - execute cycle

- Each instr gets a sequence of control signals
  - Instr fetch is the same for each instr
- The control signals determine the register transfers in the data path
- Timing - how long should a control signal be active?
  - Data signals must travel from a register, across the bus, through the ALU, and finally to a register => minimum clock cycle time

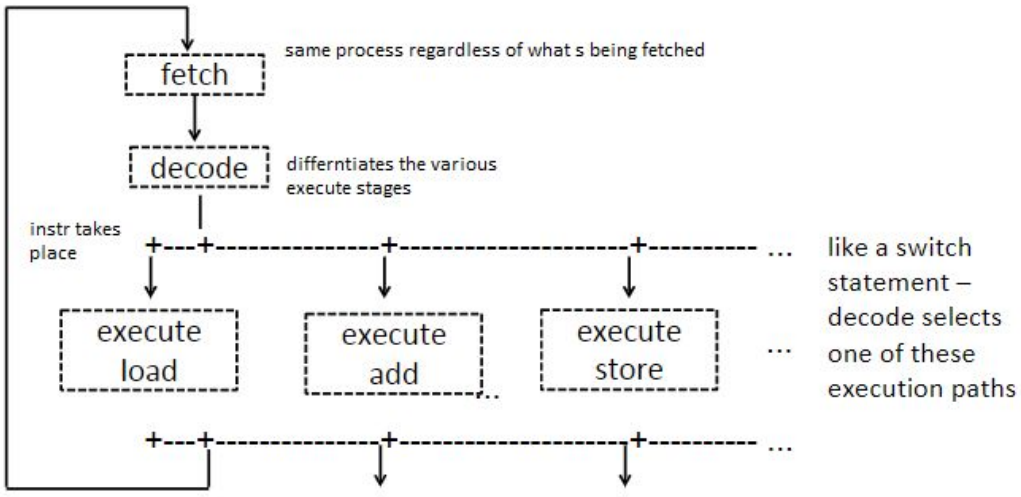
IE
(each variable is a time)

TravelFromRegister + timeacrossbus + timethruALU + intoregister >= minimum clock cycle time

IE if all of these happen as fast as possible, the MINIMUM time it will take should be equal to the minimum clock cycle time

If the clock cycle time was shorter than the time for data to travel,
Data just wont make it to storage and will get destroyed

fetch-execute flowchart

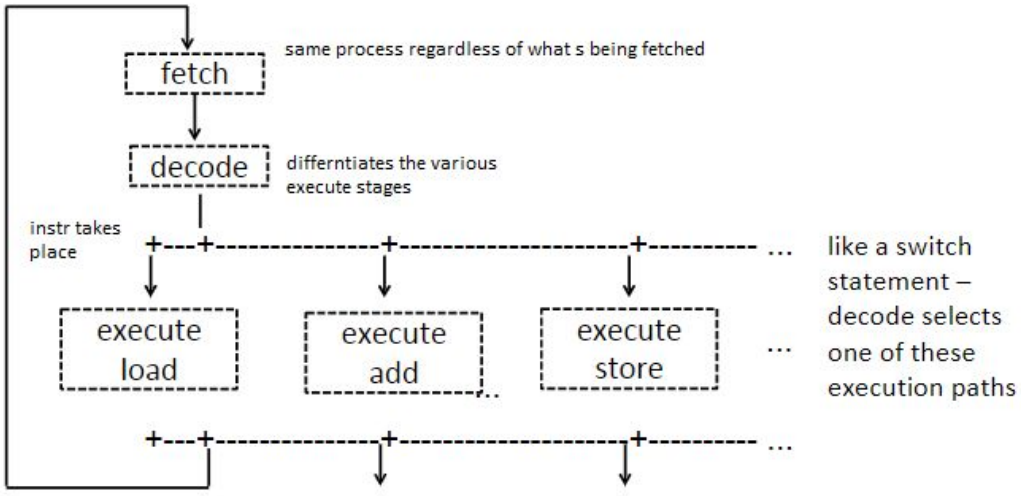



```
for A := B + C ;
assembly code / action on computer
                ;
load(B)         ; ACC <- memory[B]          register is loaded with a copy
                ;                           of the value in memory[B]
add(C)          ; ACC <- ACC + memory[C]    value in ACC register is added with
                ;                           a copy of the value in memory[C],
                ;                           and sum is placed in ACC register
store(A)        ; memory[A] <- ACC          memory[A] gets a copy of the value
                ;                           in ACC register
```

3 Distinct stages

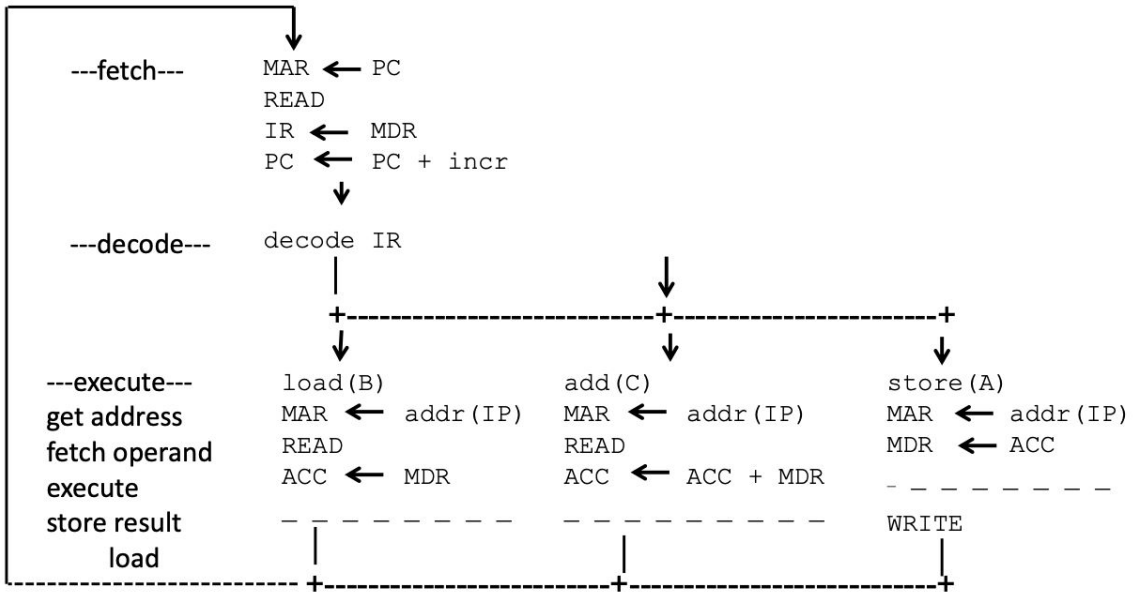
- Instruction fetch stage
- Decode stage - control signals generated
- Execute stage - set of actions actually take place here

fetch-execute flowchart




```
for A := B + C ;
assembly code / action on computer
                ;
load(B)         ; ACC <- memory[B]          register is loaded with a copy
                ;                           of the value in memory[B]
add(C)          ; ACC <- ACC + memory[C]    value in ACC register is added with
                ;                           a copy of the value in memory[C],
                ;                           and sum is placed in ACC register
store(A)        ; memory[A] <- ACC          memory[A] gets a copy of the value
                ;                           in ACC register
```

egister transfers (datapath actions) for these three instructions



can check for external signals (interrupts) between instructions

This is a more detailed fetch-execute flow chart, shows steps that take place in each stage
This figure would continue right for all opcode and processes. Has to stop in image because of size constraints (load, add, store, …)

Reiterating steps for register :

1. Fetch
2. Decode
3. Execute

PC -> MAR ->WAIT ->MDR->IR->PC->PC+INCR

Java Virtual Machine (JVM)

Load pushes from variables
Store pops from stack to local variables
Swap swaps the top words on the stack

Dup - takes value on top and makes a copy, pushes val on top of stack

add/sub/mul/div/rem - the two values on top of stack are popped.
    If stack looks like this
    A
    B

    Then the values will be taken in order

    B <operation> A = result

Increment -

6 Relationship Ops: - pop top value off the stack

    1.Ifeq - if value is 0, branch t0 <label>
    2.Ifge - if value is greater than or equal to 0, brach to instruction at <label>
    3. Ifgt - if value is more than 0, branch to <label>
    4. Ifle - if less than or equal to 0, branch to <label>
    5.Iflt - if value is less than 0, brach to instruction at <label>
    6.Ifne - if value is not zero, go to <label>

Invokevirtual - method call specialized for print with a local var as an argument

Return instruction - same as halt from assembly

9/9/2019

Instruction Set design

- One goal of instruction set design is to minimize instruction length
- Many instructions were designed with compilers in mind.
- Determining how operands are addressed is a key component of instruction set design

Instruction Format

- Defines the layout of bits in an instruction
- Includes opcode and has implicit or explicit operands

- Usually several instr formats in an instr set
- Huge variety of instruction formats have been designed; they vary widely from processor to processor

    - Most modern instruction sets have many instruction formats
    - RISC - reduced instruction set computing - set up for the processor to execute as fast as possible
    - CISC - complex instruction set computing - instructions cant run as fast, but will take fewer instructions to complete

    RISC and CISC basically have a trade off in a LOT of fast instructions or a few slower instructions (respectively)

Instruction Length

- The most basic issue
- Affected by and affects
    - Memory Size

- Memory organization
- Bus structure
    - Ideally, one instruction fetch per cycle
        (no wasted bus transfers)
        IE bus width >= instruction size
            ^ this ensures that instructions arent truncated and at
            least one makes it through when needed.
- CPU complexity
    - CISC focuses on this (do more each instr at the expense of speed)
- CPU Speed
    - RISC focuses on this (do less each instr to get them done faster)
- Trade off between powerful instruction repertoire and saving space with shorter instructions

IE

- Large instruction set -> small programs
- Small instruction set -> larger programs
- Large memory => longer instructions
- Fixed length instructions same size or multiple of bus width -> fast fetch (just fetch the same amount of mem every time)
- Variable length instructions may need extra bus cycles
    (case where instruction size doesnt match with bus transfer rate will take multiple cycles to get all instructions)

Instruction Format Trade-offs

Processor may execute faster than fetch

- Use cache memory or use shorter instructions
- NOTE: complex relationship between word size, character size, instruction size, and bus transfer width

- Most all modern computers use multiples of 8 related by powers of 2 for instruction size. (IE 32 bit, 64 bit, 128 bit)
  - Smallest addressable unit in mem is the 8 bit chunk, a “byte”
  - Powers of 2 is a constraint so these sizes are easily related

Allocation of Bits

Determines important factors

Number of addressing modes

- Implicit operands dont need bits (specified by the operation)
- X86 uses 2 bit mode field to specify interpretation of 3 bit operand fields

Number of operands

- 3 operand formats are rare (most take 1-2 operands for input)
  - 1 operand ex- negation
  - 2 operand ex- add
- For two operand instructions we can use one or two operand mode indicators
- X86 uses only one 2 bit indicator

Allocation of bits

- Register vs memory addressing
  - Tradeoff between # of registers and program size
  - Studies suggest optimal number between 8 and 32
    - Over 32 uses half the instr for addressing mode
  - Most newer architectures have 32 bits or more
  - X86 architecture allows some computation in memory

- Number of register sets
  - RISC archs tend to have larger sets of uniform registers
  - Small register sets require fewer opcode bits
  - Specialized register sets can reduce opcode bits further by implicit reference

Determines several important factors (cont’d)

- Address range
  - Large address space requires large instructions for direct addressing
  - Many architectures some restricted or short forms of displacement addressing
    - EX: x86 short jumps and loops,
      powerPC 16 bit displacement addressing
- Address granularity
  - Size of object addressed
  - Typically 8,16,32, 64 instruction variants
    ^ 8 bit (1 byte) is the minimum address granularity

Data types are in this form so any smaller data type aligns with respect to the size of larger data type

MUCH like how char can be expressed as an int

Addressing Modes

- For a given instr set architecture, addressing modes define how machine language instructions identify the operand(s) of each instruction
- An addressing mode specifies how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within a machine instruction

- COMMON ADDRESSING MODES
    - Immediate addressing (No memory Accesses)
        - Holds value for operation
        - Operand = address field
        - The value is stored in memory immediately after the instruction opcode in memory
        - Similar to using a constant in a high level language
        - EX, add_immediate(5)
            - Add 5 to contents of accumulator
            - 5 is the operand (5 sits in the address field)
            - Other EX - B = A + 10, 10 is immediately addressed

    - Advantage
        - - fast, since the value is included in the instr
        - No memory ref to fetch data
    - Disadvantage
        - Inflexible since operand values are fixed at compile time
        - Can have limited range in machines with fixed length instruction
    - INSTRUCTION
      Opcode| Operand
- Memory Direct addressing (direct) [memory access for alias]
    - Holds mem address for a variable
    - Advantage
        - Single mem reference to access data
        - No additional calculations to determine effective address
        - More flexible than immediate addressing
    - Disadvantages
        - Limited address space
- Indirect (Memory Indirect addressing) [memory access for alias, then ptr]
    - Address field of instruction is the address of a pointer that points to operand (IE pointer memory addressing)
    - IE the memory held in a variable is a memory address. The memory at that held address would be indirectly accessed
    - Advantages
        - Large Address Space
        - 2^n where n = word length (this defines address space)
        - May be nested, multilevel, cascaded
            - Nesting greatly increases addressing time
    - Disadvantages
        - Multiple em access, hence slower
- Register

    - Operand is held in register named in address field
    - EA = R - r means register
    - Dominant addressing mode for operations in a load/store machine
      IN C you can specify register variables with keyword register
        But it isn't a guarantee - only advisory to compiler
    - Advantages
        - No memory accesses because on data path
        - VERY fast
        - Very Small address field needed
            - Shorter instr (3-5 bits usually)
            - Faster instr fetch
        - Multiple registers improves performance
    - Disadvantages
        - Limited number of registers
- Register Indirect
    - Register holds pointer to operand in memory
    - EA =(R)
    - One fewer mem access than indirect addressing
    - Large address space (2^n)
    - Advantages
        -
    - Disadvantages
        -
- Displacement
    - Uses pointer and a second value for an offset
        - Like how arrays have a start(ptr)+index(how many places forward to move)
    - EA = (R) + A
        - Address field holds 2 values
        - Address field holds the base address (R)
        - Address field holds offset (A)
    - Useful for elements in an array
    - Has “Subcases”
        - Relative Addressing Mode

        - Uses an implicitly addressed register for base ptr
            - Then add offset to address held there
        - EA = (PC) + A
        - R = Program counter
        - IE get operand from A cells from current loc pointed to by PC
    - Base Register Addressing
        - R holds ptr to base address
        - R may be implicit or explicit
        - A Holds displacement
    - Indexed Addressing
        - A = base
        - R = displacement
        - EA = A +(R)
        - Good for accessing arrays
            - EA = A + (R)
            - R++
        - Pre-Index
            - R has a base address
            - Address is found immediately
        - Post-Index
            - R iterates through memory
            - EA = (R)
            - (R) += A
- Advantages
    -
- Disadvantages
    -
- Implied (stack)
    - Operand is implicitly on top of stack
    - IE the iadd from JVM
        - Does Not need address as it knows to get 2 values from the stack

ARM ASSEMBLY

ARM Registers

Register - interal CPU hardware that stores binary data. Can be accessed more rapidly than RAM

ARM HAS

- 13 general purpose registers R0-R12
- 1 stack pointer - R13
- 1 Link Register (LR) R14 - holds the callers return address
  - Need this when entering a fxn to be ready to get the return val
- 1 Program Counter (PC) - R15
  - If a result gets placed here, a seg fault is going to happen probs
- 1 current program status register (CPSR) R16
  - (NON ADDRESSABLE)
  - Doesnt hold data
  - Contains condition flags set by arithmetic and logical CPU instr and used for conditional execution
  - Gen condition code flags
  - N - pos 31 - negative/lessthan
  - Z - zero - pos 30
  - C - Carry Out - pos 29
  - O - Overflow - pos 28

Wednesday the 18th - First Exam
Up through addressing modes

Review Session Tuesday

9/16/2019

- Review session tomorrow 8pm-10pm
- Test on
  - Everything up until addressing modes

Processor Status register

- Register 16 in ARM
  - Non-addressable
  - Has logical flags for conditional execution

  - 4 leftmost bits - condition code flags

  - Bit position
  - 28 - overflow, called V
  - 29 - CarryOut, called C
  - 30 - Zero, called Z
  - 31 - Negative, called N

Processor Status Register (PSR)

- The N, Z, C, and V bits are the condition code flags
- Flags are set by arithmetic and logical CPU instr, for use in conditional execution
  - Usually only with comparison instr

ARM instructions

ARM instr are written as an opcode followed by zero or more operands

Operands can be constants, registers, or mem references

Mem references use register direct addressing

ARM being a load/store machine, these are the only instr that use mem references

Simplified instruction syntax

Opcode{cond}{flags} Rd, Rn, operand2

Where

Cond - optional condition of most recent comparison

Flags - an optional flag (EG- S)

Rd - the destination register

Rn is the first source register

Operand2 is a flexible second operand

Syntax is rigid

1 operator, 3 operands

Why 3 operands, most ops are binary

You want to save your result,

Therefore you have 3

write in the result into destination

Preserve the value of other operands

Move operation, takes a value and assigns it to a register

```
          Mov r1, <value>
               // this is a unary operation, in that you store
only one parameter
               // unary operations will allow the only parm to
be an immediately
               // addressed constant value
     Note:
          Operand 2 is a flexible second operand to most instr
          It is passed through the barrel shifter (a functional
unit that can rotate and
               shift values)
               Also Note: Barrel shifter only sees last 8 bits
          It takes one of 4 forms
               Immediate value: an *8Bit* number rotated right
          by an even num of
                    Places
                    8 bits, allows up to 255
               Register
                    Can use operand 2 the same way you pass any
               other
                         register
               Register shifted by a value: IE a 5 bit unsigned
               int shift
                    After the register provided for the last
               operand
                         There is a “sub instruction” for the
               barrel shifter
                              Can either take an immediate value
                                   IE 5 bit unsigned int shift
                    This is a constant shift
               Register shifted by a register: the bottom 8 bits
               of a register
                    This will shift by a variable shift
                         IE found in the register
```

Examply calls with 3 operands

```
     Immediate values
          Add ro, r1, #3
          Mov r0, #15
          Mov r1, #0x12 //moves hex 12 into r1
     Register shifted by value
          Mov r0, r1, lsl #4 //lsl #4 is the barrel shifter sub
          instr
                         This left shift by 4
          Orr r0, r0, r1, lsr#10
                         This is shift right by 10
                              Preceding L means logical shift
          right
                              A preceding A means Arithmetic
          shift right
     Register Shifted by a register
          Cmp r1, r2, lsl r0
                         This is a shift left by r0
          Add r1, r5, r3, ror r0
                         This is a rotate by r0



     When to use shifts
          Suppose you want to mul by 2
               You can do this with an int by shifting left by 1
position


          So suppose you want to multiply a value by 16 before
storing it
               You use a left shift by 4 bits

          Suppose you want to DIVIDE a value by 16
               You use a right shift by 4 bits?


ARM comments
```

Use
// - same as c/c++
@ will block out a line as well as //
/* comment */ works as well

Labels in ARM
For our emulator, a label in arm is designated as the following
Label_Name:

As long as it isnt an opcode, it will work

Immidiates
Numerical constants
They appear a lot so there are ways to indicate the existence

An immediate is specified by the # symbol

Move instructions
The one and only SOURCE operand is designated to behave as operand2
This allow you to put constants/immediates in registers

Ex:
```
Mov r0, #15    //use an immediate. R0 = 15
Mov r0, r1     // ref another register, r0 = r1
```

Mvn - move negated - this command will take a val from
The second operand. This is bitwise negated, which means it will
then take a bitwise all 1 and compare with value, and negate any
like bits
Immediate Values in ARM
These must be an 8 bit value

After such values are exhausted, they can be shifted to allow larger
numbers
This is limited to only binary-accessible numbers
Example only: 1024 may work
1025 will not work

Ldr

The ldr psuedo instruction loads a register with either
A 32 bit const value
An address

General Format
LDR{Condition} register,=[expression|label-expression]

Use of LDR
Generate literal const when an immediate value cannot be moved into a
register using a mov instr because it is too large

To load a program-relative address or an external address into a register

Compare Instructions

Cmp = compare
Flags set to result of (Rn - op2)
Cmn - compare negative
Flags set to result of (Rn + op2)
Tst - bitwise test
Rd := Rn or Operand2
Teq - test for equivalence
Rd:=Rn and not operand 2

Comparison produces no result, it just sets condition codes
If you want to save the result, use the S bit (appended to end of instr)
The S bit will set condition codes from result for an expression

The S is implicit on pre-defined compare calls

Logical Instructions

And - logical and	Rd <- Rn and Operand 2
Eor - exclusive or	Rd <- Rn eor Op2
Orr - Logical Or	Rd <- Rn orr Operand 2
Bic - bitwise clear	Rd <- Rn and not op2

Arithmetic instructions

- Add Rd <- Rn + Op2
- Adc Rd <- Rn + operand +carry
- Sub Rd <- Rn-Op2
- Sbc Rd <- Rn - Op2 - not(carry)
- Rsb Rd <- op2 - Rn
- Rsc Rd <- op2 - Rn - not(carry)

Conditional Control Structures in ARM

- Eq - equal - z =1
- ne - not equal - z

Suffix	Description	Flags tested
eq	Equal	**Z = 1**
ne	Not equal	**Z = 0**
cs / hs	Unsigned higher or same	**C = 1**
cc / lo	Unsigned lower	**C = 0**
mi	Minus	**N = 1**
pl	Positive or Zero	**N = 0**
vs	Overflow	**V = 1**
vc	No overflow	**V = 0**
hi	Unsigned higher	**C = 1 & Z = 0**
ls	Unsigned lower or same	**C = 0 or Z = 1**
ge	Greater than or equal	**N = V**
lt	Less than	**N != V**
gt	Greater than	**Z = 0 & N = V**
le	Less than or equal	**Z = 1 or N = !V**
al	Always	

Last trivial condition is never

Append these to comparison instructions

9/23/2019
Binary digit - called a bit
    Binary devices are relatively easy to build
        Take a simple “is it on or off”

How many binary patterns exist in n digits?
    2^n

Unsigned int representation is all the way up to
    2^n - 1

    IE for 8 bit integer the max is 254
        Why cant we use 255? Is it saved for overflow to let program know to loop
            Back to 0 to prevent breaking?

Memory unit terminology
    Word - unit of memory access and/or size of data registers
        IE 16/32/64 bit words

    Byte - unit for character representation, 8 bits
    Nibble - unit for binary coded decimal (BCD) digit, 4 bits

Conversions
    Between number bases

    Base 8 - octal   - 0, 1, 2, 3, 4, 5, 6, 7
    Base 16 - hexadecimal - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f

    Easy to take binary to octal or hex

Hex, octal both overlap with binary
    This means that each corresponding group of positions relate to a hex/octal digit

IE for
Bin - hex
Organize stream of binary digits into groups of 4
Find the hex value for each group of four bits

1001 0010 1110 0001 1010
9 2 E 1 A
Hex val = 92E1A

Hex - bin
9 2 E 1 A
1001 0010 1110 0001 1010

Find the binary representation of each digit in the hex number

Note for the above
You start from the RIGHT
Suppose you have binary
110011
Put 0’s on the left side UNTIL you get a number of digits div by 4

Alternatively, just start at the right and assign each hex digit four binary digits
When the last one has fewer than 4 then just dont worry, and go on to
Convert

Binary to octal works similarly, but will only use 3 binary digits

IE

10	010	010	111	000	011	010
2	2	2	7	0	3	2

Magic Numbers
- Arbitrary value, nonarbitrary meaning

- IE
- Numbers that read in english as hex
  - IE
  - 0xdeadbeef
  - 0xbaadf00d
- -idetify file type 0xcafebabe in java class file

- -typically chosen to be a large, negative, odd ineger
  - And thus increasingly unlikely to be used

2s compliment is used to represent a negative number
- A 2s compliment is a ones compliment + 1 at the end

Binary Arithmetic
    Half adder and full adder
        One bit adders
            Use one bit adders in series to build a full adder

Full Adder is made by the carry output going to the carry input of the next adder, with a defined number of bits

OR, you can use the same adder, save the carry bit, and use it as the I/O on the same adder

Overflow - The full adders have a carry bit that cant go to the next adder

Signed binary addition/subtraction
    Digit compliment- inverse of each bit for the number
    2s compliment - ones compliment, then add 1

Compliment - a set of digits up to the same significance in which the digit is BASE-1
    Ie 9s compliment of (100) is
                -(999)
            =   (-899)

Radix Compliment - Twos compliment in binary
            Tens compliment in dec

IE invert all bits in binary, add one

All overflow will occur when a number cannot be represented with the number of bits present. Overflow occurs at (2^n) - 1

Signed overflow gives one fewer bit, thus overflow occurs at(2^n-1) - 1

Wraparound - present when overflow occurs,
    It goes back to the min value, and starts adding in again
    THUS if you see overflow from addition of 2 positives
        Then you will have encountered overflow

Binary Arithmetic

Multiplication

Shift left then add

1 - if LSB = 1, write down multiplicand and shift one place left

2 - If the LSB is 0, then write down 0s and shift left once

3 - for each bit, repeat

4 - add all together

IE

1011 x 1101 =

```
               1011   multiplicand
               1101 multiplier
               ------
               1011
                   0000
                 1011
               1011
```

Prod = sum = 10001111

Division

A = b/c

B = dividend

C = divisor

A = quotient

Division

Same as multiplication as far as the three registers

Steps of multiplication in reverse

(shift left and add becomes sub then shift left)

Double length dividend divided by single length divisor yields single length

Quotient and single length remainder

Initially

Carry = 0

Acc - high bits of dividend - remainder

Mq - low bits of dividend - quotient

Mdr - divisor

Steps:

Shift left

Subtract

Set LSB to 1 if successful

Start again

GIVEN code for the binary arithmetic

More Binary Arithmetic - Division

```
if( mdr == 0   ) { raise( DIVIDE_BY_ZERO__SIGNAL   ); }
    if( mdr <= accumulator ){ raise( QUOTIENT_OVERFLOW_SIGNAL ); }

    for( i = 1; i <= n; i++ ) {
        /* step i */
        shift (carry:accumulator:mq) left one place
        (carry:accumulator) ⟵ (carry:accumulator) - mdr    /* subtract */
      if( (carry:accumulator) is negative )  {
            mq_bit[0] ⟵ 0
            (carry:accumulator) ⟵ (carry:accumulator) + mdr   /* restore */
      }
      else  {
          mq_bit[0] ⟵ 1
      }
    }
    results:
      quotient in mq
      remainder in accumulator
```

This is restoring division, since c:acc must be restored to prev val if sub yields neg
When dividend is double-length spec check to make sure quotient wont overflow
Written as psuedocode

IF not div by 0
    If divisor less than value on accumulator, you will encounter overflow

    For all n bits
        Shift left one place (carry:acc:mq)
        Subtract the concatenated carry:accumulator (5 bits)

        If carry:acc (result storage) are neg, youre done
            Set mq LSB to 0
            Restore of c:acc
        If nonneg
            Set mq LSB to 1

Recall steps for decimal division

```
Bool loop = true;
while(loop)
{
     Dividend -= divisor;
     Quotient++
     if(dividend < 0)
     {
          Dividend += divisor
          Loop = false;
          Quotient --;
     }
}
```

Division is repeated subtraction while your dividend is greater than 0

This process will be applicable for all bases

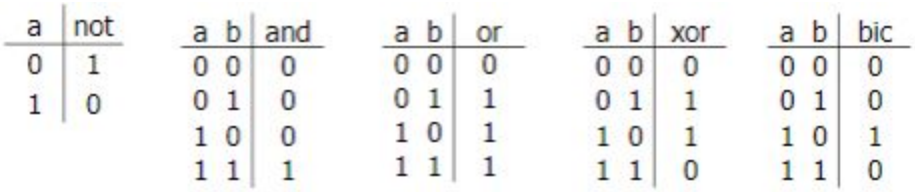
#steps = #bits in binary division

Binary Logic review

- Not - invert bit
- And - both true returns true
- Or - one or both true returns true

Xor - only one true bit returns true (two returns false)
bic - gives (a && (!b))

NOT AND OR XOR BIC



a	not
0	1
1	0

a	b	and
0	0	0
0	1	0
1	0	0
1	1	1

a	b	or
0	0	0
0	1	1
1	0	1
1	1	1

a	b	xor
0	0	0
0	1	1
1	0	1
1	1	0

a	b	bic
0	0	0
0	1	0
1	0	1
1	1	0

c/c++ logic operators
|| - or
&& - and
! - not
Zero word = false
Nonzero word = true

c/c++ bitwise operators
| - or
~ - not
^ - xor
Each bit in the word is independent

Uses for logical operators
ANDing bitwise will “preserve” the bits that are both true
OR, if the AND value is one, you copy, else you get 0

IE
1110 0101 1101 1111 <- value to extract from
0011 1111 1100 0000 <- Bitmask

--10 0101 11-- ---- <- Extracted bits
Note: - is placeholder for 0

```
Oring a bit with 1 produces 1 at output
Oring a bit with 0 produces original bit
     This can force certain digits
     IE
     Operand        mask
     0x12345678 |0x0000FFFF = 0x1234FFFF

     0011       | 1010      = 1011
     1010


And clears with 0
Or sets with 1
Bic will clear with 1


BIC w/ a 1 will reset the bit to 0
Bic w/ a 0 produces original bit
     Remember does (a && !b)

Xor w/ a 1 flips the bit
Xor w/ a 0 produces original bit
```

As we have already discussed, Arm provides the following Boolean (logical) instructions:

Instruction	Description
and{cond}{S} rd, rn, op2	Performs logical AND of rn with op2.
orr{cond}{S} rd, rn, op2	Performs logical OR of rn with op2
eor{cond}{S} rd, rn, op2	Performs logical exclusive or operation of rn with op2
mvn{cond}{S} rd, op2	Performs a bitwise logical NOT operation on the value of op2
bic{cond}{S} rd, rn, op2	Performs an AND operation on the bits in Rn with the complements of the corresponding bits in the value of Op2

Barrel Shifter

The barrel shifter is a functional unit, provides f5 operations which shift and rotate

Shifts perform op spec if done on integers

- Lsl - shift left, makes digits MORE significant
  - IE mul by 2
- Lsr - shift right, makes digits LESS significant
  - IE unsigned div by 2
    - If you shift a 1 off the right then you Have div by 2 and a remainder of 1
  - Gcc generates lsr only when using unsigned integers
- Asr - Arithmetic shift right
  - Made to work with signed int
  - New bits = sign bit (0 if pos, 1 if neg)
  - This does signed int div
- Ror - rotate right

THESE ARE NOT STANDALONE OPERATIONS IN ARM, BUT ARE SUB-OPERATIONS

Only operand 2 goes through barrel shifter

This means though that you can shift and perform ops All in the same line such as

```
Mv r1,#2  //hold output
Mv r0, #4 // value to op on
Mul r1, lsl r0, #3

This makes r1 = r1 * r0*8
           R1 = 2  * 32
           R1 = 64
```

Logical shifts

Bits that get moved off - they get lost

Bits that get moved on - they are 0

Bitwise rotation

Circular shift

Unlike arithmetic shift, rotate does not preserve sign, but takes the digits moved off one side and moves them on the other side

A shift left has no choice as to what to do with sign bit - it was already in the register, which is determined by the shift amount; zeros always come into lsb on right

ror – rotates right by #n bits; valid range for #n: 1 - 31

Note to rotate left you just rotate right n - 1 times, where n is the number of bits

FIELD EXTR WITH BITMASK

((1 << m) - 1)<< n

Without bitmask

Left shift and right shift to isolate desired field

Mul by small constants

To use shifts and addition

Convert mul by powers of 2, do the left shifts

Same for div by powers of 2

10/9/2019

Feild extraction using a mask

To create a mask - set the certain bits in the mask you want to 1, make the others 0
And the two operands, it will preserve only the masked bits

To do this make the bit one PAST the largest bit you want to be 1, then subtract 1.

((1 << m) - 1 << n)

Alternatively You can do this with shifting
IE

- start by moving the most significant bit to extract to the most significant bit in the register
- Shift such that the least significant bit in the feild is in bit place 0

Feild Insertion with a bitmask
First, clear the spots to insert (use an extraction mask and a bitclear, it will not the mask and make 0's in the pos you wish to clear)

Now shift the value to desired position
Insert new value to the position by making a new "insertion mask", with the desired value in the correct position. OR this with the value to be inserted into

This is important in GRAPHICS - IE if you want to manipulate just the RED field, youll need to make a bitmask that will then operate on only bits that are in the red range of significance4

Review

Shifts

Recall shifting LEFT will double vals in binary
Shifting RIGHT will halve vals in binary

In cases where x * ((2^N) - 1) it is easier to just
Just shift and subtract once - its faster

IE

X * 7

= x(2^3) - x(2^0)

If you have a binary with more 1 bits than 0 bits,
It will be faster to subtract out the 0 bits
(use with a reverse subtract)

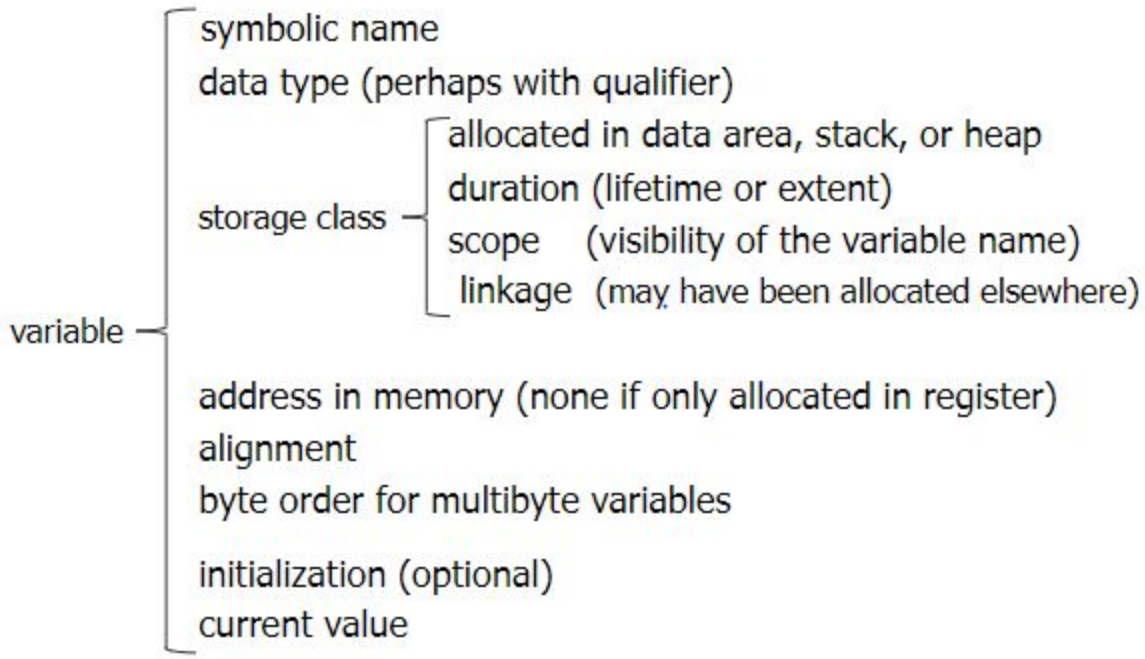
IE x* 7 becomes

```
Rsb r1,r0, r0, lsl #3
```

Recall - when doing unsigned div
Use lsr
When doing signed div
Use asr (arithmetic shift right)

Data In Memory

Variables have multiple attributes



Some of these

Common to hll and LLL

Variables have symbolic names (things like calling a variable “foo”)

Low Level - (data type based on instructions performed)

High Level Language - (data type declared)

Storage Class

- Where the mem is stored (stack/ heap)
- Duration (IE when var “dies”
- Scope - where the variable may be refered to
- Linkage - linking between source files

Address in mem (is this where rvalues come into play?)

Alignment - denotes aligned/unaligned
- Aligned IFF Mem address / sizeof(variable) gives no remainder

Byte for multibyte variables
- Big endian - most sig bit on LEFT (how you read)
- Small endian - most sig bit on RIGHT
  - Chars/strings dont get interpreted as big/little endian

Initialization (optional) (does this mean the init value is preserved through entire life)
Current value

Qualifiers -
- Signed - default
- Unsigned
- Long
- Short
- Value

Data Sizes

Byte	-	8 bits
Halfword	-	16 bits or 2 bytes
Word	-	32 bits or 4 bytes
Double Word	-	64 or 8 bytes

Alignment in byte- addressable memory
(note all sizes are multiples and powers of 2)

Mem access is typically word by word access

Alignment restrictions on load/stor instr prevent multiple byte units from spanning across two words. THUS… (view the slide) allows individual load/store instr are more efficient (

Unaligned access

- May be illegal (cause bus error
- May cause a trap in operating system
- Requires extra hardware for 2 mem access and insertion of Bytes

Alignment rules

- REVIEW REST IN FRee TIME

10/21/2019

Data Alignment in Memory WRE Structures

Program demonstrating alignment

Pack.c

Each struct in Pack.c will have a varying number of bytes needed to allocate the struct in memory. Assuming they are built on a 32bit machine, they must “Align” on a multiple of 32. This means that if we 32 bytes of other data, and one char (1 byte) then the struct will be passed as 64 bits, etc.

Alignment rules

Bytes start anywhere

Halfwords start on halfword boundary (address %2 = 0)

Words start on a word boundary (address % 4 = 0)

The size of a struct must be divisible by the largest member

Remember

Char - 1 byte

Short int - 2 bytes

Int - 4 bytes

Double - 8 bytes

Most efficient way of ordering values in a struct is in descending size order

Organization of Programs in memory at the largest Scale

At the largest scale the program gets broken into logical parts

- Stack - automatically allocated variables (local vars, subdivided into stack frames - one per procedure invocation)
  - Also holds everything else that can reside in a stack frame resides
    - IE return addresses, etc.
    - Function calls add to stack when called
      - Removed when returned
- Heap - dynamically allocated variables
- Initialized data (called data segment) - global/static vars that are initialized by programmer. Includes read only data
  (read only data = ROData)
    - Gets allocated at start, de-alloc at prog end
  - There are 3 sections of the data segment, listed additionally below
    - Variables - data segment
    - Read only data - rodata
    - Bss segment (block started by symbol)

  - Uninitialized data (bss, acronym for Block Started by Symbol) - global and static vars that are init to 0 or dont have explicit initialization.

- Text - program code/instructions
  - Allocated at load time
    - De-alloc at termination

Run-Time Environment of a program

Text segment
DAta Segment (vars decl once per program)
HEAP (grows down)
|
…
|
STACK (grows up) - space for saved procedure info
(stack pointer points to top of stack)

Memory Stack

Stack matches the last in first out of nested procedure calls

Storage for automatic vars , that is, for local vars from subroutine calls (get discarded at end of call)

The collection of info related to a specific instance Of calling a subroutine is called a stack frame

- A generic stack frame has:
  - Parameters
  - Return address
  - Registers to save
  - Local variables
  - Old frame pointer
- Stack pointer
  - Points to top of stack

Stack is located in high address in mem, growing towards low addresses, so creating space for local vars decrements the stack pointer

Subroutines must make room for local variables

ARM has following load and store instr

Ldr - load word (4 bytes)
Ldrh - load halfword (2 bytes)
Ldrb - load byte (1 byte)
Ldm - load multiple words
Str - store word (4 bytes)
Strh - store halfword (2 bytes)
Strb - store byte (1 byte)
Stm - store multiple words

Typical form of load for accessing a local var

```
Ldr rn, [sp, #4]
     ^ load from
          ^ operands get brackets around first or both ops
               Sp = stack pointer
          ^ first op must be a register
               ^ second op can be immediately addressed
```

(displacement)

Typical form of store for accessing a local var:

Str rs, [sp, #4]

^ takes val in register

^ stores it at stack pointer + second op

^ second op determines displacement

Pointers

Address of an object in memory

A pointer typically is 64 bits on current machines

A ptr can be in a register or mem word

Parameter Passing Techniques

PAssing data to/from subroutine calls can be done through parms and return value of a function

- You can pass by:
  - Call by Value:
    - Input parms declared with INN in Ada
    - Default method for parms in C, C++, and Pascal
    - Parameters of primitive types in java
  - Call by - result:
    - Output parameters (Declared with OUT) in Ada
      - IE up to function to assign value to these parms
  - Call by value result
    - in/out parms declared with “IN OUT” in ada
  - Call by reference (pref with large memory size parameters)
    - Large array parameters in ADA
    - Array parameters in C and C++
      - In C you use the pointer value for “reference”
    - Reference parms declared with & in c++
    - Reference parms declared with VAR in pascal
    - Object parms in java
    - Default method for parms in ForTran

10/23/2019

Project 3 - a program to calculate the byte offset in a 2d array of integers in row-major order
Fxn will read the row/col index

Call by value
- Copy values of actual parms into mem locs of formal parms before executing the body of the subroutine; do nothing on return

Call by result
- Do nothing prior to executing the body of the subroutine; copy the final values of the formal parms into mem locs of actual parms on return
  - useful in return values (IE the value returned is named in parm list)

Call by value/result commonalities
Seperate variable for formal parameter

Call by value-result
You copy parms in
You write the result back into parms to pass out

How is it different from call by reference?
You copy in/ copy out
The things that happen intermediately do not get “recorded” in the actual parameter’s memory location

Call by reference
Pass addresses of parameters
Copy these addresses of actual parms
On each ref to a formal parm in the body of the subroutine, perform an indirect reference to the corresponding actual parm
- The formal parm is an alias of the actual parm, thus both refer to the same obj
- Changes made using the formal parm are executed as if the actually parm was used

Call by value
Advantages

Loads and stores op on formal parms

Disadvantage

Copying overhead

Call by ref

No copying, allows subr to change vals of actual parms

Indirect ref thru addresses in formal parms to actual parms

Optimizations

pass parms in registers (not using stack)

Can sometimes not save/restore registers when executing a leaf Routine

Call by value result

Requires copys in and out

Lower level Subroutines

Subroutines - interchangable with function

Reason for subroutine - so you dont have to repeat code, or similar code with slightly different parms

Hide design decisions/complexity

Partition off code susceptible to change

Provide for seperate compilation

Provide for re-use

Open subroutine

MACRO

Resolved before assembly by textual substitution

Body of routine is placed in-line at call site with parm Substitution

Do not use these to call other open subroutines

Never use for recursive fxns

Reason for these is because it MAY be more efficient

Because they can maintain sequential execution

Closed subroutine -> standard notation
branch/execute/return at run time
One copy of body which acessesses its formal parms

Three main concepts involved WRT subroutines
Transfering control from calling prog to a subroutine and back
call/return control flow
Call - transfer control to subr
Return - transfer control to parent prog
Passing parm vals to a subroutine and results back from subr
Writing subroutine code that is independant of the calling prog

In addition, sometimes a subroutine must allocate space for local Vars

Transferring control from calling program, to subroutine, and back
Necessary to save the return address, then branch to subroutine

This can be accomplished in ARM using the branch and link Instruction (bl)

```
bl subr_name
    //this will keep the program counter for us
```

To transfer control back to the calling program, can use the branch and exchange instruction (bx)

```
bx lr
   ^^ - this is important, it holds the program counter
```

This branch instruction does not use a label target
It uses a register as a branch target address

You can avoid using this, given you push lr to the Stack. Then pop directly to the pc

Placing a return addressing a link reg works so long as no nested subroutines

For nested subroutines, it is necessary to save return address to stack

Subroutine calls and returns are LIFO. and older processors save the return address by pushing onto stack
Older processors typically pass parameters by pushing them onto same mem stack that holds the return address

PArameters
- Actual parameters - vals or addresses passed
- Formal parms - the variables named as parms, used in implementation of a subroutine

Register usage
- Up to 4 registers for parameters (r0-r3)
  - 5th onward use stack

- Temp register variables, must be preserved
  - R4 - r11
  - Stack base (r9)
  - Stack limit if software stack checking selected (r10)

- r12 - spare register

- Stack pointer - r13/sp
  - Should be 8-byte (2 word) aligned
- Link register - r14/lr
  - Can be used as temporary val once stacked
- Program counter r15/pc

Command line args in Assembly
- Load / Store instr
- Working with Arrays in Assembly

Command line arguents
- Arguments to main are passed by the command interpreter
  - Argc / argv
  - A count of args and a “vector” of ags

  - Each ARMGUMENT is formatted as a separate char string by placing \0 after each token

Argv[0] is the command name

By convention, a leading “-” means a program option, usually followed by a single letter.

On unix systems, the environment vars follow the command line arguments

Command line args are ALWAYS strings - even numeric arguments

Stdlib fxn to convert from char to numeric
IE use atoi

Put your string constants in the

.section “.rodata” //section
Fmt1: “<STRING LITERAL>”

Load/store instructions
Arm is a load/store architecture
Only load and store ops can access mem
All other ops must have values in registers
Does not support memory to memory data processing
Operations
Must move data to registers before making use of them

IE 3 ways to move data values
- Single register data transfer (LDR, STR)
  - Copy single data transfer between memory and register
- Block Data transfer (LDM, STM)
  - Copy MULTIPLE data segments and place them into registers or consecutive sequence in memory
- Single data swap (SWP)
  - Works both ways at once, takes value in register and swaps it with value in the specified memory address

Basics for load/store

Signed only has special consideration with loads

Store will not concern itself with signed/unsigned

This is bc store will only store part of the data,

Ignoring frivilous “padding”

Bc load needs to know to preserve/not preserve sign

Store can store as is, the load will make it correct

LDR STR - Word

LDRB STRB - byte

LDRH STRH - Halfword

LDRSB - Signed byte load

LDRSH - Signed halfword

Memory system must support all access sizes

Syntax

- LDR {<cond>}{<size>} Rd, <address>

IE this results in calls like

LDREQB

Load if equal, one byte

To transfer a word of data you need to specify

- The register to place the operand in
- Memory address (more difficult to find)
  - Think of mem as a single 1d array
    - You address it simply by supplying a pointer to address
    - There are times when you want to offset from the ptr, so you can add directly to it (pointer arithmetic)

(this offset is like your index value, unsigned int 0 -> n-1 elements)

IE arguments in order

- Destination (load) / source (Store)
- Address of loc in mem
- offset /displacement (unsigned int)

Note [r1] means take the address in r1 and dereference it

When there is an offset, bracket placement will determine how bracket placement is used

- Pre - index addressing - the address generated is used immediately
- Post- index addressing - the address generated later replaces the base register

Bracket placement

- Immediate offset (ie pre - index addressing)
  Ldr r2, [r1, #12] //load value 12 bits from r1 into r2

- Register offset (still pre - index addressing)
  Ldr r2, [r1, r2] //offset address in r1 by values in r2
  Then place the value found in
  *(r1+r2) into r2
- Scaled register offset (pre indexing still)
  Ldr r0, [r1, r2, lsl#2] //allows shifts on the 3rd operand
  Helpful for arrays like really

****Pre indexing discards value for indexing into memory after the load or store****

Pre indexing with update - will update the register containing the address to access with the address plus the value of the offset arg
(can be done with all of the above offset systems as well)
To do this, append a “!” after the final bracket

If you want to place the bracket after the second argument (address register), you will change it to POST INDEX addressing
It will access the address in the address argument
THEN it will generate the offest address and update the address
Variable

Displacement values can be NEGATIVE or POSITIVE
Atypical to be using negative offsets

LOAD INSTRUCTION REFERENCE ON CANVAS

Arrays

In HLL, there are many types of data structures

At assembly level, these all map to single dimensional arrays
To emulate HLL, you have to map it to the HLL representation

Simplest data structure - 1d array
Associated with each arr is the base address (first elements storage loc)

In Assembly, you have to allocate space for the array

IE

```
.data
.align 2 //keep the data aligned on 2 byte structure
A:    .skip 400
```

The above will make 100 integers, all 2 byte spaced

To reference any element in an array, you need starting address and the index of the element

With arrays, it helps to access using pre-indexing without update to

1 - immediately address the desired data
2 - preserve the base pointer

11/06/2019

Programming examples for CLA and data transfer, and intro to arrays

Passing arrays to functions
- In the example he showed (calling printarray)
  - You put the base address of array in r0
  - You put size of array in r1

Discerning PAss by value and pass by reference
- Loads in pass by value are done in the CALLING CODE

- Loads in pass by reference are done in the CALLED CODE
  - Keep in mind instead of moves into and from the register
  - Use loads and stores to place or pull val from register

Pointers in C vs. ARM, multiple entry points
- We are not restricted to just one label per function
  - If you want to use default arguments, its a good place to use multiple entry points

- Multiple exit points are also possible with many bl instructions
  - or pops into pc

Pointers in C vs ARM

- Pointers in C are addresses of variables
- ^ common to C and assembly

- Pointers in C are *TYPED*
- IE a char* is different from an int*
  - Note you can still cast pointers, and this creates no extra Work

In machine instructions - assembler dgaf about data types or pointer types since pointers are JUST addresses to memory

Recall

sizeof(int) = 4

sizeof(ptr) = 4

sizeof(char) = 1

sizeof(charptr) = 4

In C

But in ARM

All addresses are 32 bits (4 bytes), so an int ptr is the same as a char pointer in LITERALLY every way

Addressing into memory

In ARM you must use BYTE offset instead of index offset

IE if i want element 1

ARM - ptr + 4

C - ptr[1] = *(ptr + 4) at low level

C (ptr arithmetic) - ptr+1 moves it sizeof(type)

Separate assembly

Allows program to be built from modules rather than single source File

Multiple files need to be assembled to machine code(object files) and then linked into a single executable file

Even use of things like printf or scanf require linking

Separate source files are nice for same reasons as HLL

Program Testing

Bottom up dev

Write and test lowest level modules first

Needs written test drivers

Top down dev

Write High level programs first, and test logic

Requires effort of “stub” routines that are placeholders for lower level subr that will be developed later

Useful also to test that a call occurred

Linking Objects

By defualt, compiler/assembler makes .o or object files

Uses -c for armc production

Linker yeilds an executable file (a.out)

(got named ld)

-o option to name output executable is used by linker

Type codes in .o files

Upper case - global

Lower case - local

Note local symbols will change address wrt to object file

Linker resolves external references between .o, .a, and .so files

The linker performs storage management to assign regions within the executable file

CPSC 2310

Exam is on Monday the 18th

Linking objects

Each object file may have one or more unresolved symbols

It makes sense that these unresolved symbols dont have Address

The Linkers job is to resolved these unresolved symbols

And create an executable

Dynamic linking

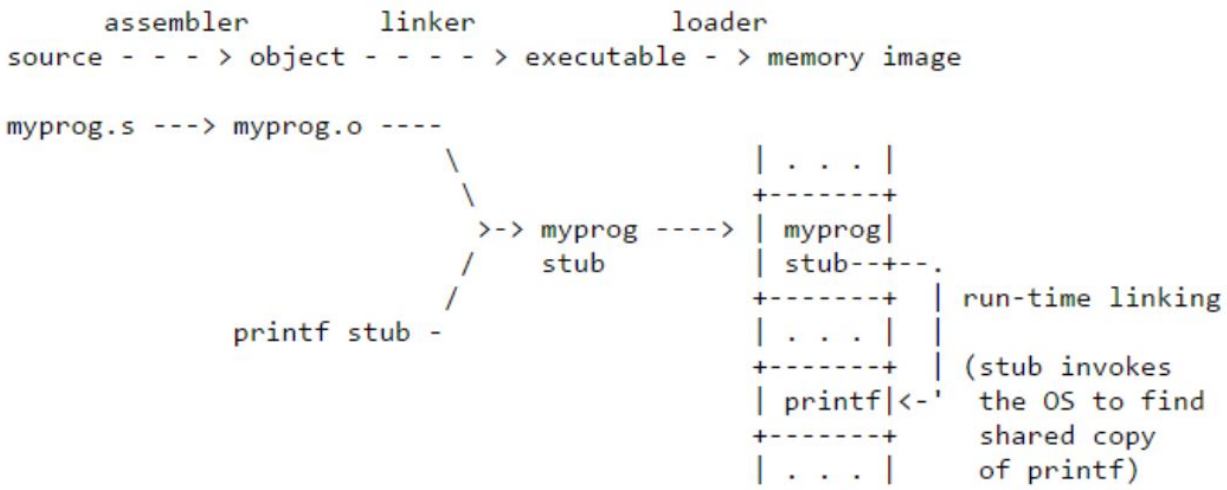
The linker will resolve dynamic symbols to refer to a special subroutine which then finds the shared object (library function)

Static linking

The linker will copy code from library archive to the executable

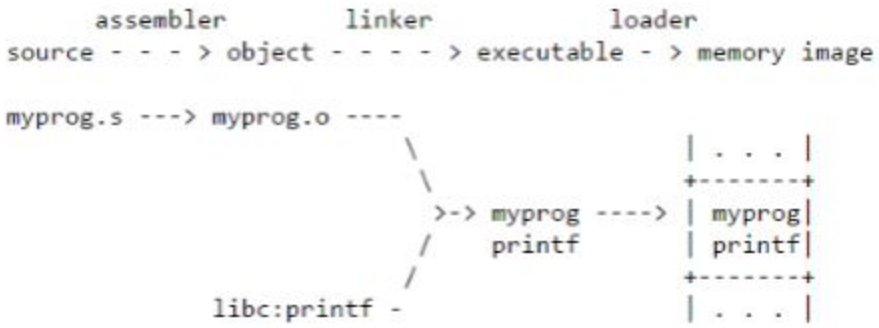
Simple obj files and libraries/archives are combined using static linking to make a self contained executable. However, for common library routines such as printf, this requires too much disk space for every executable that uses the common routine to store its own copy

Shared Objects will use dynamic linking to save disk space (since program doesnt need to keep a copy of the shared obj in the executable) And memory space.



As seen, this is the general path of how compilation goes with DYNAMIC LINKING (note how myprog does NOT contain printf, the stub subroutine will take us there so we can do what is needed.)

If there was Static linking - these objects would all be placed in the executable. This leads to MASSIVE executable(as seen below)



Dynamic Linking

- Originally all programs were statically linked
- All references fully resolved
- Each program's executable was self - complete
  - (you could run the program, given same architecture, in a vacuum)

Since the 1980s, most systems support shared libraries and Dynamic Linking
- For common libraries, only a single copy kept to be shared
- Dont know where lib is loaded until runtime, and must resolve references dynamically as program runs

Static linking advantages
- Executable is self contained
- No runtime overhead

Dynamic Linking Advantages
- Reduced disk space for executable
- Only one copy of shared routine needs to be in memory
- Thus reduced mem space across several currently executing programs
- Will get Latest version of shared object

Running a program
- Actually loading into memory
- Branching to the entry point address

Loader usually performs address relocation as words containing absolute addresses are loaded into mem, relocation is required in both the linker and loader so the program will run with correct addresses.

Summary

	Assembler	Linker	Loading
PC-relative offset			
* caller and subroutine in same source file	bound	--	--
* caller and subroutine in different files	--	bound	--
absolute address			
* definitions and use in same source file	bound	may need relocation	may need relocation
* definition and use in different files	--	bound	may need relocation

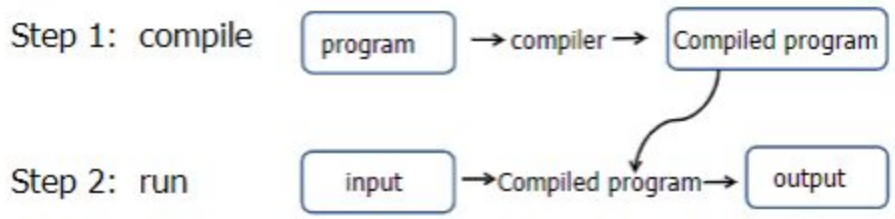
The linker will bind any global symbols

It will need to re-find the addresses for local symbols

IE they are first given “relative” mem addresses
Relative being relative to the read only data segment, the text segment, etc

There is then relocation needed when its used in main memory

Compilation Vs. Interpretation
Compilation diagram

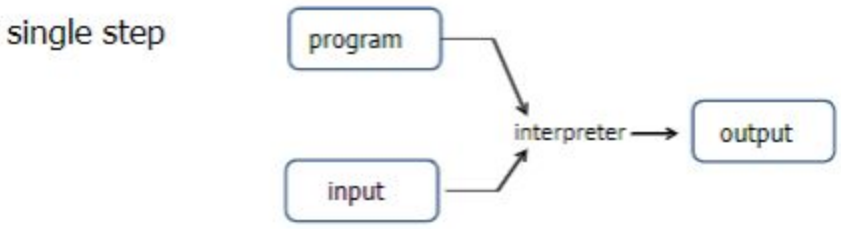


Compilation - translation from one language to another - typically easier to execute. A pure compiler will translate source code to machine executable instructions

Comiplation allows one translation and multiple execution of executable. Thus a fairly large amt of time gets spent by compiler to make optimizations once, in order to make a better executable

A compiled program runs fast but is harder to debug
Compiler example - gcc

Interpretation diagram



The source files with interpreter directly runs program
The interpreter will translate lines of code in source files

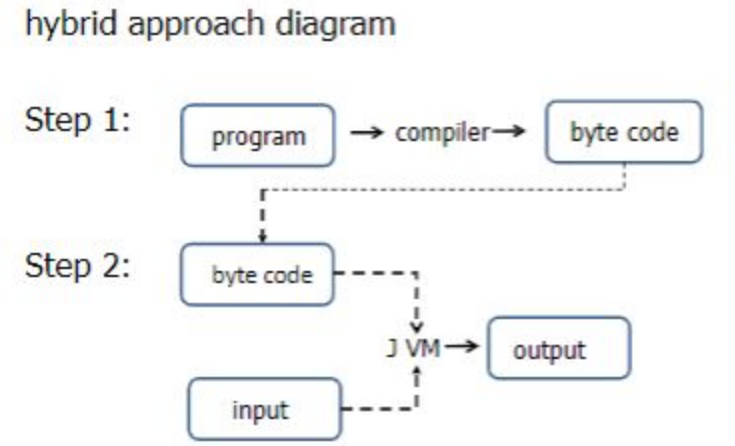
The interpreter will skip making the translation to another language
It combines the translation and execution

Interpretation starts from source code each time you want to run program. It performs same analysis as compiler but on source line by source line basis

A pure interpreter keeps NO results from this analysis even when encountering the same source line (even within a loop)

Interpreted programs are EASIER to debug
Interpreted languages easily support dynamic typing and dynamic scoping of variables
Interpreter examples, shells, m4, or python on the cmd line
Also formatted I/O (printf) relies on interpretation

These approaches are NOT mutually exclusive
    Hybrid approaches exist



The program gets compiled.
    The java byte code gets interpreted.

Java compiler and JVM interpreted - hybridization

    Javac produces byte code (easy to interpret)
    Java interprets byte code

Provides for a portability of byte code files across systems
    Perl also uses hybrid translation model
    Others do too

Other hybrid translation model is the JIT compiler
    JIT compiler - compile functions/procedures on the first call

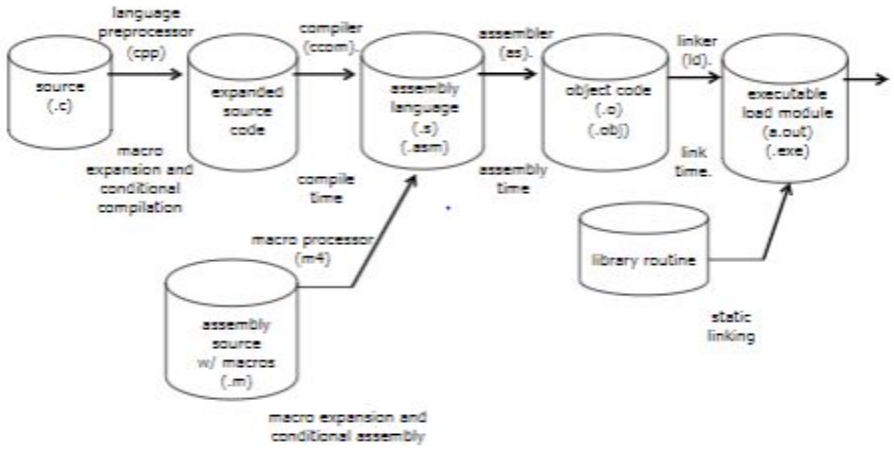
    Terminology - source code that needs to be compiled is typically
    Called a program, while code that is interpreted may be called a
    script.

Major translators in compilation model

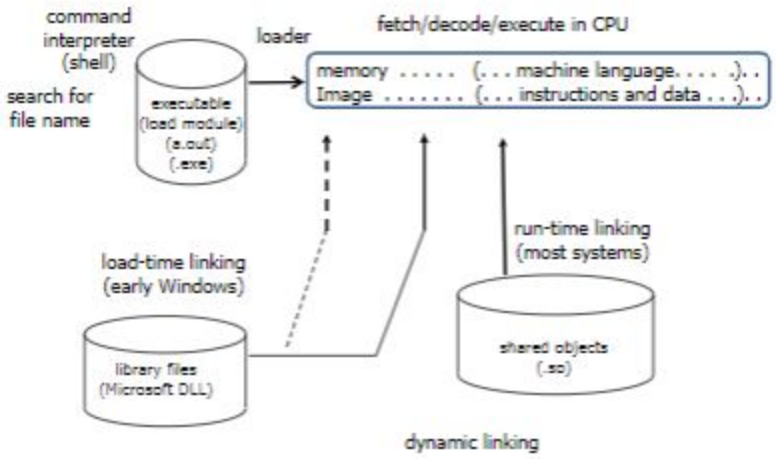
1 Language preprocessor
    Text substitution and conditional compilation
    (direct execution of special statements)
    (things like #include, etc)
2 Compiler
    Lexical analysis, parsing, conditional assembly
    (all code is now compiler syntax format)
    Here it goes from HLL -> machine instrs
3 Macro processor
    Textual substitution and conditional assembly
    (this is like the language preprocessor for machine code)
4 Assembly
    Translates symbols into addresses and machine code
5 Linker
    External symbol resolution plus relocation
    Makes executable
6 Loader (part of run step)
    Relocation according to load addresses, produces mem image

NOTE: many compilers generate obj code directly, without calling
    a seperate assembler

COMPILER STEPS GRAPHIC



LOAD/RUN STEPS GRAPHIC



Dynamic linking - at broadest sense occurs during run stage

True dynamic linking happens at runtime

11/20/2019

Files and Buffers

Device driver

The computer can only access those on the BUS

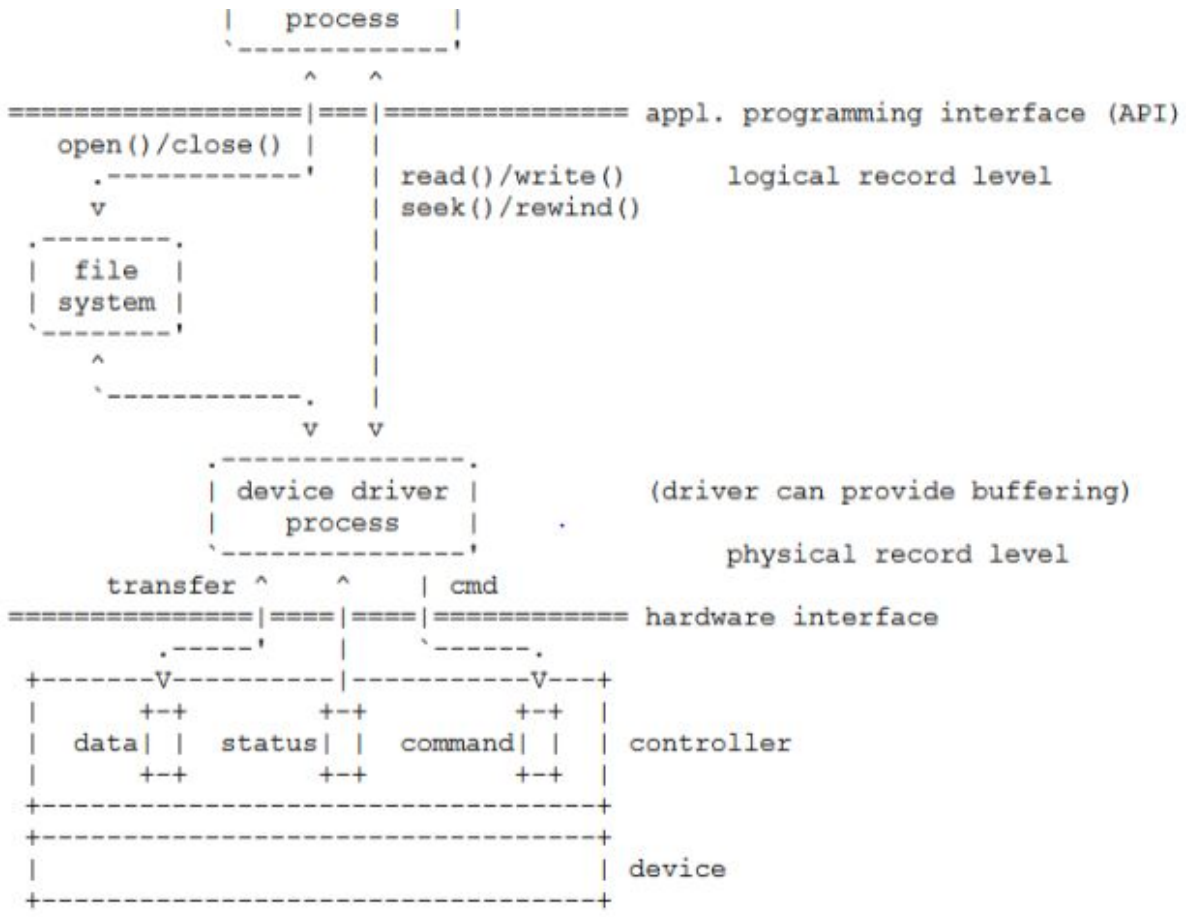
Thus it does not talk directly with the peripheral

But instead talks with the controller

Remember - the disk is a peripheral device

Process within an OS that interacts with I/O controller

Application (this part got cut off from below pic)



The API provides functions to use the High level operations on the files (open/ close)

Also includes read, write, seek, rewind (reverse seek)

The libraries providing the HLOps on files use Lower level libraries
    To interact with files and the file system. They also handle Buffering.

The libraries for the API
    The actual logic used on controller determines hardware interface

Circular Buffer
    A fixed size array as if front connected to back
        (ie index goes 0-> size-1 -> 0)

    Natural to use a modulo - but it is slow bc it needs to
        Do a division
    ^ Instead use a bitwise and
        IE val % 8 = val & 7
        This is only applicable when the mod value is a power of 2
            In which case, the mod value - 1 can be anded with the
                Variable value in the preceding expression
    Youll have two indexes - the in index and out index
        In index points to the place to insert
        Out index points to the place to remove FROM
        In == Out -> Buffer is empty
        Out == (in + 1)%size -> buffer is full

    IE if you wish to put (push/ enqueue) a value
        You have to check that the buffer is not full
            If it is shorcut out of the placement
        If it is full
            You may place the value in the place specified by
            index in, increment index in

    Get works the opposite way

Buffering using File I/O

Want to reduce/eliminate delay caused by IO device

timeline

Read data into buffer
Process data in buffer
Read data into buffer
Process data in buffer
…

Overlap I/O into one buffer with processing of data in separate Buffer

This is called double buffering (below)

IE you have buffer 1 and buffer 2

While buffer 1 is reading, buffer 2 is processing
While buffer 2 is readong, buffer 1 is processing

Cost: memory inefficiency

Files:

Request operations called

By calls to buffered functions from c STDIO library or

By calls to functions from Low level Unix I/O library
(this is a case of un-buffered IO)

Directly invoke OS by intentional interrupts
(this is the application talking with controller Directly, but is usually is only possible in kernel mode.)

Buffered file operations

All done with file pointers (file pointer tells lower level OS library IO info pertaining to file descriptor, the buffer, pos of indexes for buffer, etc.)

Fopen - pass name, check permissions, return file ptr
Fclose- release file (release from exclusive write access)
Fgetc - read a single char
Fputc - write single char
Fgets - read string
Fputc - write string (there are more…)

Low level file operations

Open - pass a symbolic name, checks permissions, returns file DESCRIPTOR

Close - release the file

Read - pass file descriptor, buffer addr, and byte count
Write - pass file descriptor, buffer addr, and byte count

Standard files in Unix (already open)

Stdin (file descriptor 0)
stdout(file descriptor 1)
stderr(file descriptor 2)

File structure

Struct: (named FILE)

Has a number of chars in buffer (int)
The ptr to next character from/to in buffer (char *)
Has base address of buffer (char*)
Has the state of stream (char flag)
Has file descriptor (unsigned char)
Other options (unsigned char)

There is a heirarchy of wrappers - but all lead to invoking OS

IE there is the following heirarchy

```
scanf()
    fscanf(stdin, )
        read(fd, buffer, buff_length)
            (here is where assembly level occurs)
```

Directory - map filename to disk locations of data

Also stores file protection info

Blocking

- The transfer of multiple logical records as a single physical record. Thus, the transfer overhead, such as seek nd rotational latency on a disk transfer is only required once for the single physical record rather than for each logical record

IE
Logical record 1 \
Logical record 2 -> transferred as single physical record
Logical record 3 /

Logical record is apllications unit of transfer
Physical record is IO unit of transfer

Real Numbers

Range - need to deal with a large range, not just 2^n-1 to (2^n-1) - 1

You need extremely large nums and small fractional nums

3

Fixed point numbers

The programer would move the point and essentially “scale” the Number

Fixed point = Integer part <assumed binary point> fraction part

Use positional representation to convert a binary fraction to a Decimal value

Using the fixed point representation - it is easy to go from Decimal to binary with the exception of powers of 2 (ie you must accommodate for a special case of integer part Being one bit more than the previous value)

When converting fixed binary point numbers

Right of decimal, place counting starts at 0, goes left

Left of decimal, place counting starts at -1, goes right

Converting a decimal fraction to a bin fraction

Repeatedly mulitply decimal fraction by two, taking the integer part 0 or 1 as bits of the fraction from left to right, stop when fraction is 0

IE

```
0.375 x 2 = 0.75
             ^
0.75 * 2 = 1.5
           ^
0.5 * 2 = 1.0
         ^
```

Therefore the binary fraction is 0.011

Remember

To convert integer part

Repeated division until only 0s get placed

To convert fraction part
Repeated multiplication until only 0s get placed

Floating Point Numbers
Automatic scaling by hardware
Like Scientific notation, automatic scaling by hardware requires an exponent field (IE scale factor) and mantissa (ie 1 . fraction)
^
Note the 1 is implied
In something called normal Form, and is omitted for another fractional bit

- One sign bit
- Range governed by number of bits in exponent
- Precision is governed by the number of bits in mantissa

Mantissa -> This is the part to be scaled by the scale factor

Scale Factor -> base ^ exponent

Floating point

S| exp | Fractional

Value = (-1^s) x (1.fraction)(2^exponent)
^sign bit

Exponent uses bias notation so that negative exponents have leading 0s
IE you subtract your bias from the exponent)
Historical reason is that integer bitwise ops work with this form of floating point representation

Scale factor and step size increase geometrically
Both are proportional as well
(ie 0.125, then 0.25, then 0.5, then 1.00)

Normal Form -
The integer part of the fractional portion is a 1

General formula for step size/scale factor

Precision - # of digits to represent a number
Accuracy - closeness to the value you wish to represent
Rounding - choosing nearest representable neighbor

Note - anything not representable as a sum of powers of 2
    Cannot be represented in floating point representation

Normal form - gauranteed upper bound for relative error
  For nonzero representations in normal form
    2^ -(num fraction bits + 1)

For a max number that gets truncated or rounded to zero. The max relative error can reach a factor of 1 if 0.1 is represented as 0

Denormal numbers are used for gradual underflow rather than just truncating values less than 2^min exp to 0

In the prev ex for 0.1, use 00 exp for smallest exp value 2^-1 but with no hidden bit, thus assign 0 00 01 = 0.125

Five bit floating point representation
  Note there is a distinction between 0 and negative 0
    You at least retain whether you round a positive vs neg num

Relative error bound of 1 / 2^(bits in fraction + 1) holds down to the smallest normalized number (2^min_exp)

Zero must be represented as de-normal (there is no nonzero digit that will make a 0 floating point value)

Floating point addition

The positions of each bit must line up

The fraction bits will change “effective positions” as they scale, so you need to scale one of the values so bits match significance

(note you cannot shift normal form left, since it would make it de-normal, so you shift the smaller value to the right and increase its exponent. Note you get a limited number of fraction bits, so you loose precision in your fraction bit when you are increasing the smaller fp number)

If when shifting, if you have a value in the 2s place,
In the mantissa, then you shift right to fix it
Note you may end up adding a normal and denormal floating point numbers

Subtraction is similar with 1 exception

When you subract 2 approx equal values, straddling 2 different scaling exponents, you get catastrophic cancellation.

Catastrophic cancellation - the leading bits cancel out and all that is left are the trailing bits, which are most affected by representation

Ie

```
     2.0
-    1.75
-----------
     0.25  but with floating point 5 bit representation = 0.5
```

Guard bit - Prevents catastrophic cancellation problem
The guard bit will catch the least significant fraction bit

Round bit - This provides for rounding, to increase rounding accuracy

Sticky bit -

If guard bit is 0 - round down (truncate)

There are many rule, they are on the slides.
    Keep in mind you ROUND TO 0
        Therefore if the LSB in frac = 0

IN SUMMARY (of above)
    0xx - round down = do nothing (0 for gaurd bit)

    100 - this is a tie, round up if matissa LSB is 1, else round down

Multiplication
    Multiply the 1.fraction parts
    Add the exponent parts

Also note
    There are single and double precision floating point

    There are also
        Extended precision - 80 bit format
        Quad precision   - 128 bit format

Special codes
    NaN - not a number (propagates through any operation)
    Inf - (propagates through most operations)
    Denormal numbers

IO devices and Interrupts

Bus connects CPU with communicable devices
- Memory and other controllers

Take keyboard for example
- Each key press calls an interrupt and sends a scan code
- Each key release also causes an interrupt and sends a scan code
- Keyboard ISR must keep track of scan codes and translate to ASCII
  - (typing A takes 4 interrupts, 2 for shift and 2 for the 'a' key)
  - ISR = interrupt service routine
- Auto repeat function requires a timer to be set at each depress and special processing should it go off prior to key release
- Raw mode - all chars sent to buffer and onto program (IE your program gets the backspace char too)
  (programs responsibility to process data received)
- Cooked mode - special characters processed in buffer before sending to program.

- Output devices
  - Monitor
    - Memory mapped display buffer
    - Monitor adapter refreshes display from display buffer
    - Will often use special dual ported mem chips (VRAM)
      - Dual ported - gets written to and read from Simultaneously

Disk
- A track - the concentric ring containing data
- Disk sector - a region denoted by the area spanned between outer arc and center (pie Slice)
- Track Sector - the intersection of a track and disk sector

Latency - the disk head being repositioned on tracks
Rotational latenct - how long it takes for the right sector to appear after seeking to correct track

- Rotating platters covered with magnetic oxide coating
- Collection of read/write heads, one per surface, mounted on an arm, which moves in and out
- Concentric circles by where r/w heads can be positioned are called tracks. The set of parallel tracks defined by one arm position is called a cylinder (ie tracks found without Needing a seek)
- Simple disks have a fixed number of sectors per track and a small num of byters per sector (IE 512)
- Disk acces needs
  - Seek - time to move head to track
  - Rotational latency - time for the proper sector to Rotate under the read/write head
  - Transfer - time to read/write the sector depends on sector size, but usuall frac of millisecond

Bus
Address lines
Data Lines
Control Lines (read, write, interrupt request,...)

RAID storage
Multiple disks operate in parallel to work as a faster disk with larger capacity

Raid = redundant array of inexpensive disks

Striping - spreading data thats being written to disk being written to all disks simultaneously

Raid5 and raid6 are similar, but raid 6 is more redundant and can deal with 2 disk failures before a failure (but with cost of needing 2 more disks)

Level 0 - no redundancy but parrallel access

Level 1 - disk mirroring/shadowing

2n disks
Multiple simultaneous IOs allowed
Each write goes to 2 disks
Read from either

Level 2 - bit interleaved array - similar to Error Correction Code

N+k disks
Dedicated ECC disks
Expensive and uncommon

Level 5

N+1 disks
…

First few slides of interrupts (READ THESE SLIDES)

Controllers have the

data , status, and command registers

Data reg - holds data going to/from device
Status register

Memory Mapped IO

ONe address space div into 2 parts
Some addr refer to phys mem loc
Other addr reference peripherals
…
To send data to device, cpu write to the right addresses

Programmed I/O

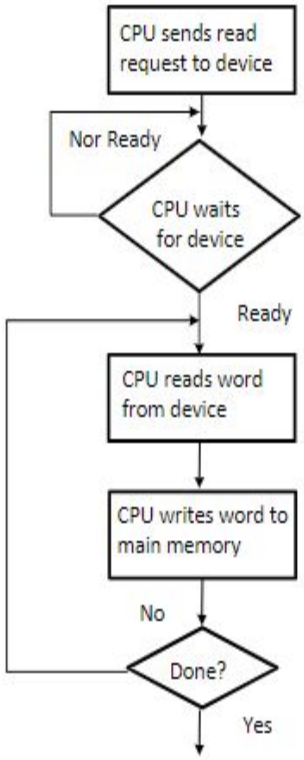
An ordinary subroutine will check if device is ready, and request an op from status register, perform, and then interact with data Register

Innefficient because there is a busy wait loop while device becomes ready as well as involves CPU through whole process

First CPU makes request and waits for device by checking status register repeatedly

A lot of CPU time is needed, if device is slow, the CPU will wait for a while

Programmed I/O



I/O - input/output

```
//  write bytes from memory buffer to device
  /***
   pseudo-code
    |
    |  int  count = N;
    |  char *addr = memory_buffer;
    |  char byte;
    |
    |  do{ byte = *addr;
    |
    |     while( io_device_status != READY ) /* busy wait */ ;
    |
    |     io_device_data    = byte;
    |     io_device_command = WRITE;
    |
    |     addr++;
    |     count--;
    |
    |  }while( count > 0 );

*/
```

This is a bad form of IO, since cpu spends 99.9% time waiting

I/O - input/output

programmed I/O (cont'd)

```
/***
 read bytes from device to memory buffer

    pseudo-code
    |
    |  int  count = N;
    |  char *addr = memory_buffer;
    |  char byte;
    |
    |  do{ io_device_command = READ;
    |
    |     while( io_device_status != READY ) /* busy wait */ ;
    |
    |     byte = io_device_data;
    |     *addr = byte;
    |
    |     addr++;
    |     count--;
    |
    |  }while( count > 0 );
  */
```

This is for reading bytes from mem as well

Interrupt Driven I/O

Interrupt: automatic transfer of software execution to a hardware event that is asynchronous with the current software execution

Hardware event is called trigger

When ready, signal gets sent to CPU and CPU responds with acknowledgement

Instead of repeatedly checking status register, the program moves on until the status register generates an interrupt saying “Im ready”

This interrupt is actually an interrupt service routine.

Relies on an external interrupt from controller to signal device is ready

This causes the executing prog to stop and the processor to enter the OS and start executing the Interrupt service routine

There are also internal interrupts (sometimes called exceptions) for various bad events (div by 0, unaligned mem access, segfault)

Interrupts can be given at ANY point so we must save

- Return address
- Processor state
- Then the prog resumes later

Usually are interrupt control bits in the controller’s cmd register, and interrupt enable bits (either priority level or bitmask) in the PSR - the processor typically disables interrupts whenever an ISR starts

The entry point address to the interrupt service routines is typically provided by a table of such addresses low in mem; for the I/O the entry is chosen according to the interrupt code placed on bus by controller

Interrupt vector table - basically array of mem addresses
Interrupt code indexes the interrupt vector table

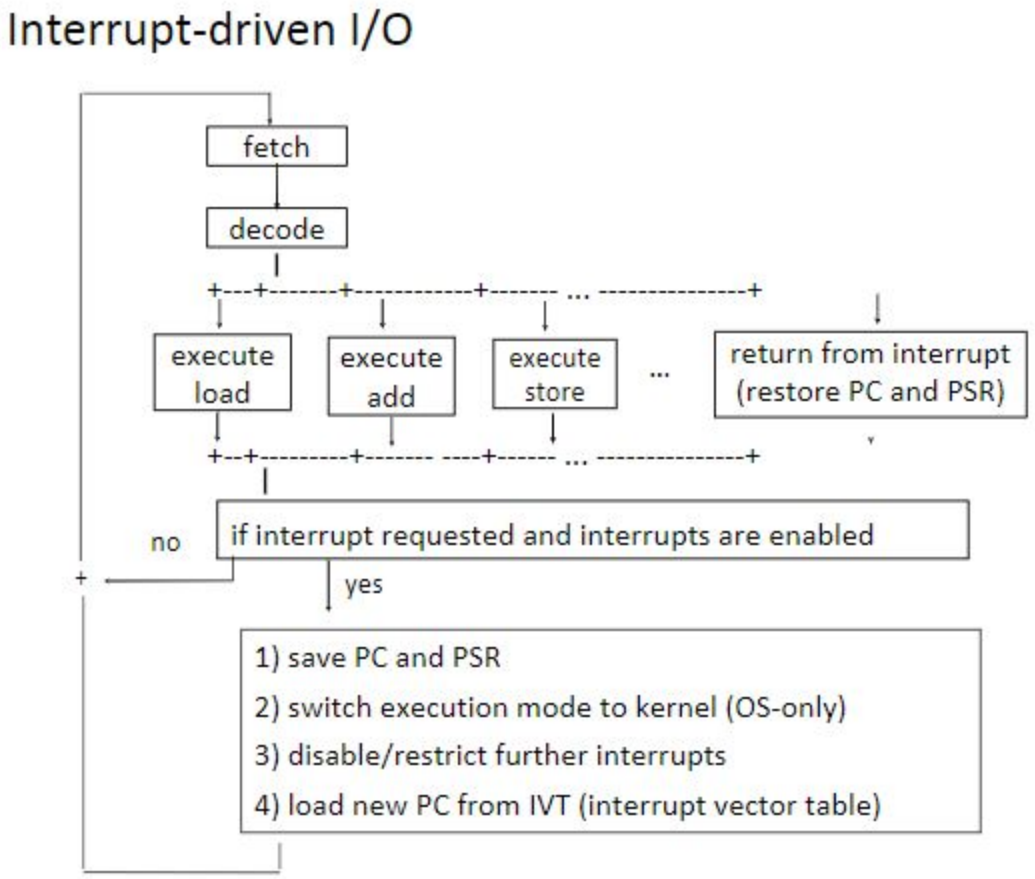
A special return from interrupt instr at the end of ISR switches back to prev processor state and restores saved PC

The fetch execute gets extended to check for interrupts after each instruction - the hardware response to an interrupt acts like a procedure call if interrupt requested by device and CPU has interrupts enabled

Note the ISR is a routine that gets fetched, decoded, executed, just like any other program.

PC - prog counter (address of next instr)
PSR - processor status register (kernel/user)
- Holds condition codes from compares
- Interrupt enable/permission (can be a single bit, mask, or priority code)
- Condition codes

IVT - interrupt vector table, contains entry point address of ISR
ISR - interrupt service routine

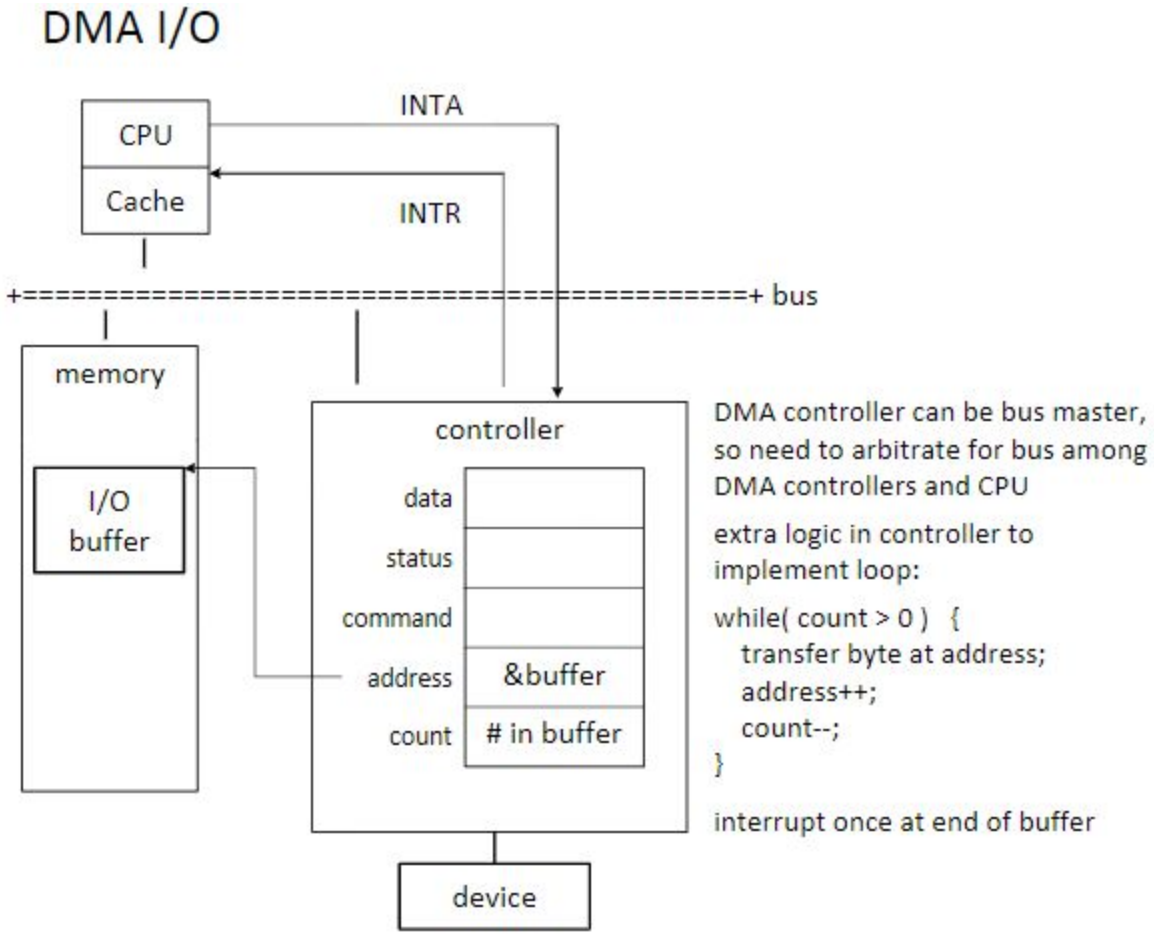


DMA I/O

Direct memory access - extra registers and logic in the controller allow it to transfer a whole block of bytes without CPU involvement, interrupts the CPU after completion or after an error

<still has the basic 3 registers

- Address register -address fo buffer in main mem, controller increments
- Count register - length of block, controller decrements to represent how much data is left to be transferred



Can transfer a set of bytes/words in a single block with a single Request

This uses interrupts but only works with DMA controller
Has 2 additional registers to ADDRESS the mem
And
Count how many words to send

CPU must support interrupts and IO controller must have 2 additional registers

Summary of I/O

I/O methods	CPU involvement	#interrupts
programmed I/O	completely dedicated to the transfer	none
interrupt-driven I/O	transfers each byte	after each byte
DMA I/O	initially loads the address and count registers; gives transfer command	one, at end of each block

With DMA and multiple blocks, #interrupts = # blocks

Channel IO

A sequence of blocks will be transferred at once, independent of CPU

Still needs the same info as DMA

Each channel command word makes a linked list to tell you what to transfer based on channel command word

CCW (channel Command worD) - can provide scatter/gather ops

Scatter - read data from phys block on an IO device and send to multiple noncontiguous IO buffers in mem

Gather - read from non contiguous memory into a single physical block to the device

Memory management
- Memory heirarchy
  - Programs exhibit locality of reference - non uniform reference patterns
- Temporal Locality
  - A prog taht references a memory loc once is likely to re-reference it in the future.
- Spatial Locality
  - A program that referrences a mem location once is likely to access a nearby mem loc in the future

Memory heirarchy

registers	on-chip	32-128 registers	250 picoseconds
L1 cache	on-chip	16 KB - 64 KB	1 nanosecond
L2 cache	SRAM	1 MB - 8 MB	5 nanoseconds
main memory	DRAM	64 MB - 512 MB	100 nanoseconds

Key idea - only keep active pieces of prog in memory and in caches
- IE
- OS control over main mem/virtual mem transfers
- Hardware control over cache/ main mem transfers
- Compiler control over register/main mem transfers

Called
- Hit - if obj to access is on current level
- Miss - go to a lower level

If hit rates in top level are high, then avg mem access time corresponds to speed of cache

Mem management policies
- Fetch policy - when to bring to higher level
  - Demand fetch - bring in when necessary
  - Prefetch
- Placement policy - where to put into higher level
- Replacement policy - when full, what to evict from higher level

Write policy - when to update lower level
    Write through - update lower level with all changes to the higher level
    Write back - update lower level only when evicted from higher level

VIEW PPT “IN_MEM_MGMT.pdf” on canvas
    Ignore segmentation exam
        View the
            Locality
            Cache performance and locality