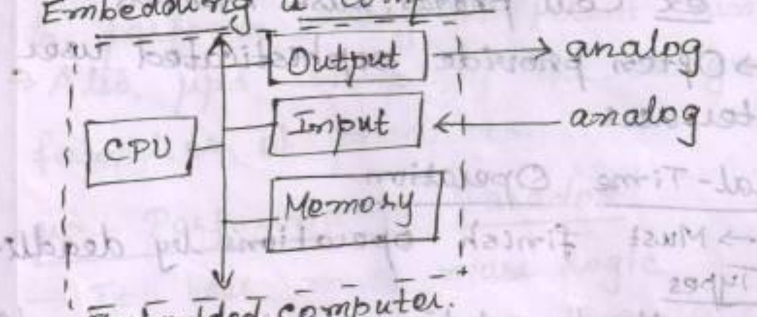
EC 6703 - Embedded & Real Time Systems.

Unit I - Introduction to Embedded Computing & ARM Processors

Embedded Computing System.

- Any device that includes a programmable computer but is not itself a general purpose computer.

Embedding a computer.



Examples for embedded System:

Cell phones, Printers, Automobile - engines, brakes. Airplane - engine, flight controls. Digital TVs, Household Appliances.

Microprocessor Varieties.

-> Microcontroller - includes I/o devices & On-board Memory.

-> Digital Signal Processor - Microprocessor optimized for digital Signal Processing.

Characteristics of embedded System.

→ sophisticated functionality.
→ Real Time Operation.
→ Low Manufacturing cost
→ Low Power

Functional Complexity

→ Often have to run sophisticated algorithms or multiple algorithms

ex Cell phones, Laser Printers.

→ Often provide sophisticated user interfaces.

Real-Time Operation

→ Must finish operations by deadlines.

Types

1. Hard real time - missing deadline causes failure.
2. Soft real time - missing deadline results in degraded performance.

→ Most of the systems are multi rate - must handle operations at widely varying rates.

Non functional requirements.

→ Many embedded systems are mass-market items that must have low manufacturing costs.

ex limited memory, microprocessor power etc.

→ Power consumption is critical

in battery-powered devices. i.e. excessive power consumption increases system cost even in wall-powered devices.

<u>Need for microprocessors</u>

→Alternatives - FPGAs, Custom Logic, etc.
→But μps are often very efficient. because they can use same logic to perform many different functions.
→ Also, μps simplify the design of families of products.

<u>μp - Performance Paradox</u>

→ It uses much more logic to implement a function than does custom logic.
→ But μps are often at least as fast : heavily pipelined, large design teams, Aggressive VLSI technology.

<u>Power</u>

→ Custom Logic uses less power, but CPUs have advantages:
- Modern μps offer features to help control power consumption.
- Software design techniques can help to reduce power consumption.

Embedded computing platform:
Hardware Architecture + Associated Software.

Physics of Software

→ Computing is physical act.
→ s/w doesn't do anything without hardware
→ Executing s/w Consumes energy & requires time
→ To understand the dynamics of s/w. first we need to characterize the platform on which the software runs

Performance means
→ In general purpose computing → may not be well-defined
→ In real time systems, → meeting deadlines
1. Missing the deadline, by even a little is bad
2. Finishing ahead of the deadline may not help.

To characterize Performance:
→ we need to analyze the system at different levels of abstraction,
· CPU, Platform, Program, Task, Multiprocessor.

Challenges in embedded System Design.
→ Hardware Requirement
↳ CPU, Memory.
→ Meeting Deadlines
↳ Faster h/w or clever s/w
→ Minimizing Power
↳ Turn off unnecessary Logic
↳ Reduce Memory access.

→ Is the specification correct?
→ Does the implementation meet the specification?
→ How do we test for real time characteristics?
→ How do we test on real data?
→ How do we work on the system?
  ↳ Observability.
  ↳ Controllability.
  ↳ What is our development platform?

Design Methodologies.

- It is a procedure for designing a system.
- Compilers, s/w engineering tools, CAD tools etc., can be used to,
  ↳ help automate methodology steps.
  ↳ Keep track of the methodology itself.

Design Goals.

→ Performance
  └ Overall speed, Deadlines
→ Functionality & User Interface.
→ Manufacturing cost.
→ Power Consumption.
→ Other requirements (Physical size, etc.,)

Levels of abstraction.

$$\begin{tikzcd}
\boxed{\text{requirements}} \arrow[r] & \boxed{\text{specification}} \arrow[r] & \boxed{\text{architecture}} \arrow[d] \\
& \boxed{\text{system integration}} & \boxed{\text{component design}} \arrow[l]
\end{tikzcd}$$

Top-down vs. bottom-up.

→ Top down design
  ↳ start from most abstract description
  ↳ Work to most detailed.

→ Bottom up design.
  ↳ Work from small components to big system.

→ Real design uses both techniques.

Stepwise refinement

→ At each level of abstraction, we must analyze the design to determine char. of the current state of the design.

→ refine the design to add detail.

Requirements.

→ Plain language description of the expected result.

Functional vs. Non-functional requirements.

→ Functional → op as a fn. of ip.

→ Non-Functional → 1. Time required to compute op. 2. size, weight 3. power consumption. 4. reliability.

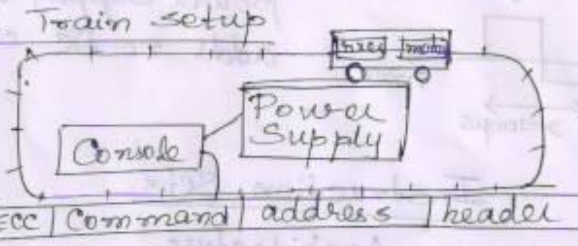
Requirement form.

Name, purpose, inputs, outputs, functions, performance, manufacturing cost, power, physical size/weight.

Model Train Controller.

Purpose -
To follow a design through several levels of abstraction.

Model Train setup



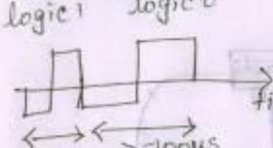
Requirements
- Console can control 8 trains on 1 track.
- Throttle has at lease 63 levels.
- Inertia control adjusts responsiveness with at least 8 levels.
- Emergency stop button.
- Error Detection scheme on schemes.

Requirements form

Name	: Model train controller.
Purpose	: control speed of <= 8 trains.
inputs	: throttle, inertia, emergency stop, train #
outputs	: Train control signals
Functions	: set engine speed, emergency stop.
Performance	: can update train speed at least 10 times/sec
Manufacturing cost	: $ 50
power	: wall powered
Physical size/weight	: console comfortable for 2 hands < 2
size/weight	: lbs.

Digital Command Control. - electrical Standard

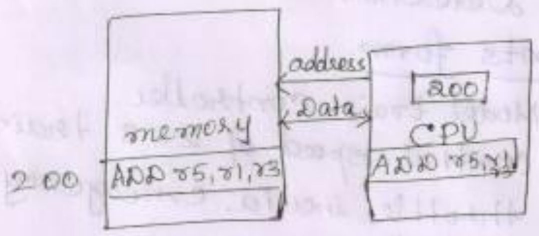
- Voltage moves around the power supply voltage; adds no DC component



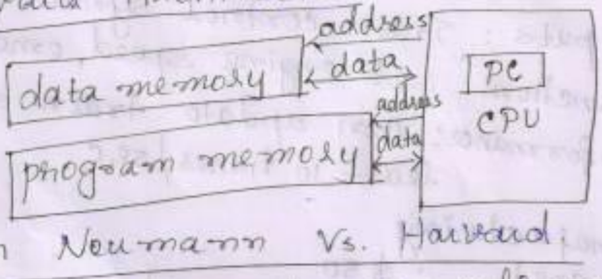
Instruction Sets.

Von Neumann Architecture

-> Memory holds data, instructions.
-> CPU fetches instructions from memory
↳ separate CPU & memory distinguishes programmable computer
~~CPU~~ CPU + Memory.



Harvard Architecture



Von Neumann Vs. Harvard.

-> Harvard can't use self-modifying code.
-> Harward allows 2 simultaneous memory fetches.
-> Most DSPs use Hardward architecture for streaming data

RISC Vs. CISC

→Complex instruction set computer
- ↳ many addressing modes
- ↳ many operations.

→Reduced instruction set computer
- ↳ load / store
- ↳ pipelinable instructions.

Instruction Set Characteristics

→ Fixed Vs Variable length
→ Addressing Modes
→ Number of operands
→ Types of operands.

Programming Model

↳ registers visible to the programmer
→ some registers are not visible.

Multiple Implementations

- ↳ Vary clock speeds
- ↳ different bus widths.
- ↳ different cache sizes
- ↳

Assembly language. - features.

- ↳ 1 instruction / line
- ↳ Labels provide names for addresses (I Column)
- ↳ Instructions often start in last column
- ↳ Columns run to end of line.

ARM Instruction Set

ARM data types

↳ word is 32 bits long.
→ Word can be divided into four 8-bit bytes.

→ ARM addresses can be 32 bits long.
→ can be configured at power-up as either little- or- big-endian mode

Endianness

① little-endian

bit 31 ... bit 0

byte 3	byte 2	byte 1	byte 0

② big-endian

bit 0 ... bit 31

byte 0	byte 1	byte 2	byte 3

ARM data instructions

- Basic format $\boxed{\text{ADD r0, r1, r2}}$
  computes r1 + r2, stores in r0.
- Immediate Operand, $\boxed{\text{ADD r0, r1, \#2}}$
  computes r1+2, stores in r0.
- ADD, ADC → add with carry
- SUB, SBC → Subtract with carry.
- RSB, RSC → Reverse subtract with carry.
- MUL, MLA → multiply and accumulate.
- AND, ORR
- BIC → bit Clear
- LSL, LSR → logical shift left / right
- ASL, ASR → Arithmetic shift left/right
- ROR → Rotate right
- RRX → Rotate right extended with C.

Data Operation Varieties.

① Logical shift - fills with zeroes.

② Arithmetic shift - fills with ones.

ARM comparison Instructions

→ CMP : Compare

→ CMN : Negated compare

→ TST : bit wise test

→ TEQ : bit wise negated test.

ARM move instructions.

- MOV ex MOV r0, r1 → sets r0 to r1.

- MVN → move (negated)

ARM load/store instructions.

- LDR → Load

- LDRH → load (half-word)

- LDRB → Load (byte)

- STR → Store

- STRH → Store (half-word)

- STRB → Store (byte)

Addressing modes.

↳ register indirect ex LDR r0, [r1]

↳ with second register ex LDR r0, [r1, -r2]

↳ with constant ex LDR r0, [r1, #4]

Example 1

C assignment → $\boxed{x = (a+b) - c ;}$

Assembler

ADR r4, a → get address for a

LDR r0, [r4] → get value of a

```
ADR r4, b -> get address for b
LDR r1, [r4] -> get value of b
ADD r3, r0, r1 -> Compute a+b
ADD r4, c -> get address for c
LDR r2, [r4] -> get value of c.
SUB r3, r3, r2 -> r3 = r3-r2.
ADR r4, x -> get address for x
STR r3, [r4] -> store value of x.
```

Example 2

C assignment $\rightarrow$ $\boxed{z = (a << 2) | (b \& 15);}$

Assembler

```
ADR r4, a -> get address for a
LDR r0, [r4] -> get value of a
MOV r0, r0, LSL 2 -> perform shift
ADR r4, b -> get address for b
LDR r1, [r4] -> get value of b
AND r1, r1, #15 -> perform AND
ORR r1, r0, r1 -> perform OR.
```

Example 3 Conditional instruction

C assignment ~~-> switch (test) {~~

$\boxed{\text{switch (test) \{ case 0; ... break; case 1: ... \}}}$

Assembler

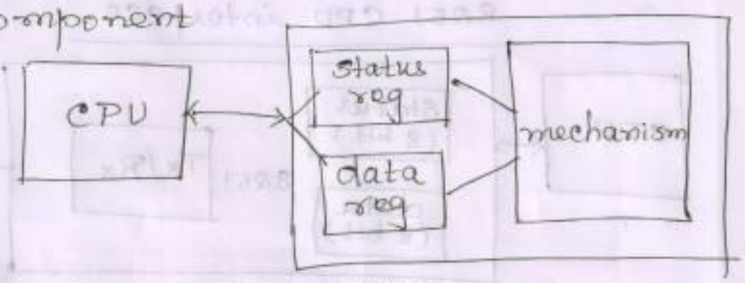
```
ADR r2, test ; get address for test
LDR r0, [r2]; load value for test
ADR r1, switchtab; load address for switch
                         table
LDR r1, [r1, r0, LSL #2] ; index switch table
```

switchtab DCD case0
    DCD case1
    ....

CPU

I/O Devices

→ Includes some nondigital component



Application : 8251 UART

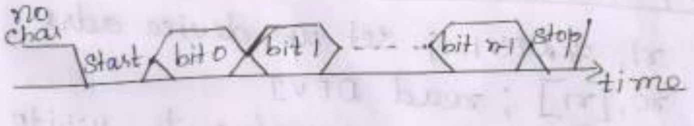
UART – Universal asynchronous receiver transmitter → provides serial communication.

→ 8251 fns are integrated into standard PC interface chip.

→ Allows many communication parameters to be programmed.

Serial Communication.

→ Characters are transmitted separately.

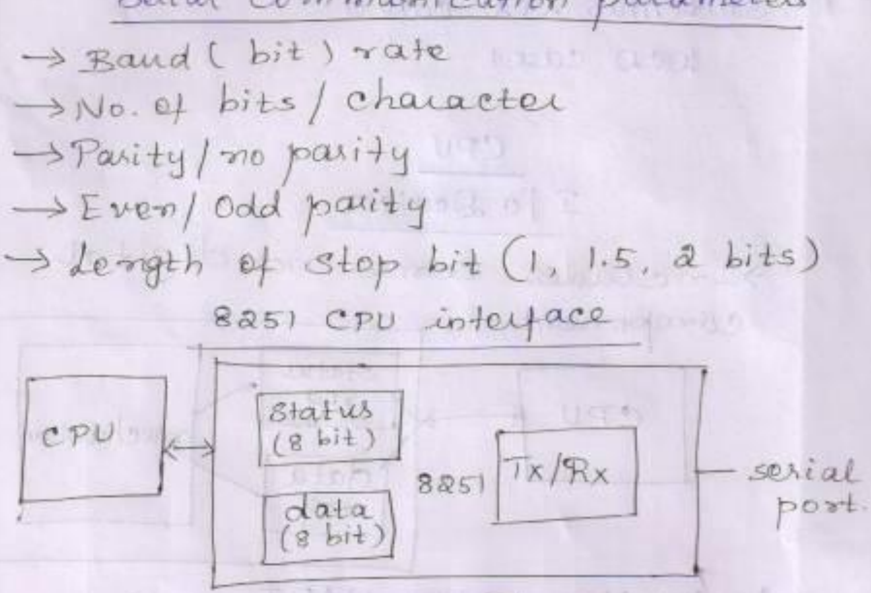


~~Ser~~

Serial Communication parameters

→ Baud (bit) rate
→ No. of bits / character
→ Parity / no parity
→ Even / Odd parity
→ Length of stop bit (1, 1.5, 2 bits)

8251 CPU interface



Programming I/O

→ 2 types of instructions can support I/O: ① special purpose I/O Instr. ② memory mapped load/store instr.
→ Intel x86 provides in, out instr.
→ Most other CPUs use memory-mapped I/O.

ARM memory mapped I/O.

Define location for device:
DEVI EQU 0x1000.

Read/Write Code.

```
LDR r1, #DEV1 ; set up device adrs
LDR r0,[r1] ; read DEV1
LDR r0, #8 ; set up value to write
STR r0,[r1] ; write value to device.
```

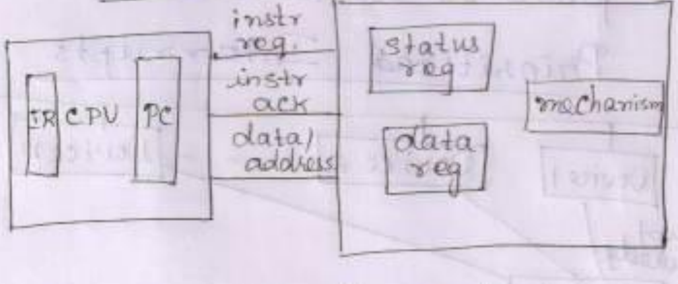
Need for interrupt I/O.

→ CPU can't do other work while testing device

→ Hard to do simultaneous I/O.

→ Interrupts allow a device to change the flow of control in the CPU.

Interrupt interface



Interrupt behavior

→ Based on subroutine call mechanism

→ Interrupt forces next instruction to be a subroutine call to a predetermined location.

Interrupt physical interface

→ CPU & device are connected by CPU bus

→ CPU & device handshake :

↳ device asserts interrupt request

↳ CPU asserts interrupt acknowledge when it can handle the interrupt

Priorities & Vectors.

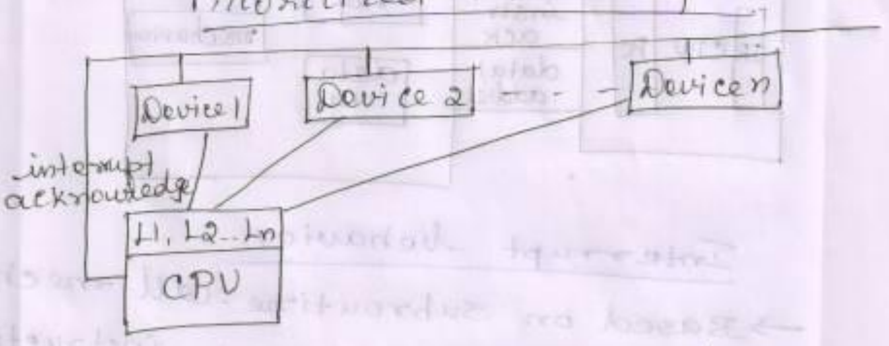
→ 2 mechanisms allow us to make interrupts more specific:

↳ Priorities determine what interrupt gets CPU first.

↳ Vectors determine what code is called for each type of interrupt.

→ Mechanisms are orthogonal: most CPUs provide both.

Prioritized Interrupts.



Interrupt prioritization.

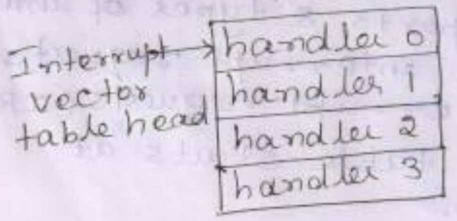
→ Masking - interrupt with priority lower than current priority is not recongnized until pending interrupt is complete

→ Non-maskable interrupt (NMI)
- highest priority, never masked.
- often used for power-down.

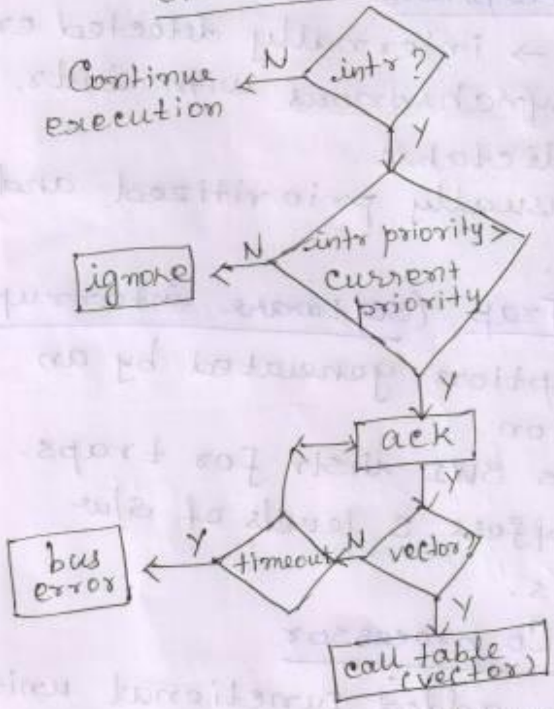
Interrupt vectors

→ Allow different devices to be handled by different code.

→ Interrupt vector table:



Generic Interrupt Mechanism.



Interrupt Sequence.

→ CPU acknowledges request

→ Device sends vector.

→ CPU calls handler
→ Software processes request
→ CPU restores state to foreground program.

ARM interrupts

→ ARM7 supports 2 types of interrupts
↳ Fast interrupt requests (FIQs)
↳ Interrupt requests (IRQs)

→ Interrupt table starts at location 0.

Exception

Exception → internally detected error
→ It is synchronous with instr. but unpredictable.
→ It is usually prioritized and vectorized.

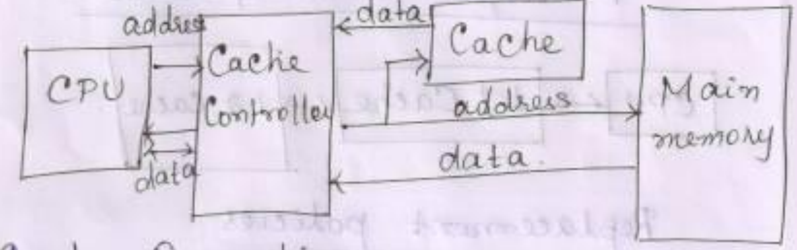
Trap (software interrupt)

- An exception generated by an instruction.
- ARM uses SWI instr for traps.
- SHARC offers 3 levels of s/w interrupts.

Co-processor

- It is a added functional unit that is called by instruction.
↳ Floating point units are often structured as co-processors.

Caches and CPUs



Cache Operation.

-> Many main memory locations are mapped onto one cache entry.

-> We may have caches for,

1. instructions
2. data
3. data + instructions.

Terms Used

-> Cache hit required location is in cache.

-> Cache miss required location is not in cache.

-> Working set set of locations used by program in a time interval.

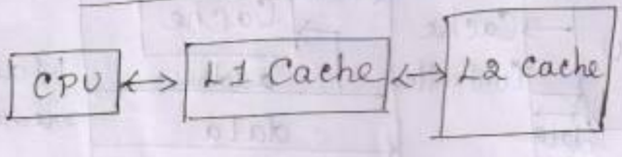
Types of misses.

-> Compulsory (cold) - location has never been accessed

-> Capacity - working set is too large

-> Conflict - multiple locations in working set map to same cache entry.

Multiple levels of cache.



Replacement policies

→ strategies for choosing which cache entry to throw out to make room for a new memory location.

→ 2 popular strategies → random, least - recently used.

Cache organizations

→ Fully associative - any memory location can be stored anywhere in the cache.

→ Direct mapped - each memory location maps onto exactly one cache entry.

→ N way set associative - each memory location can go into one of n sets.

Scratch pad Memories

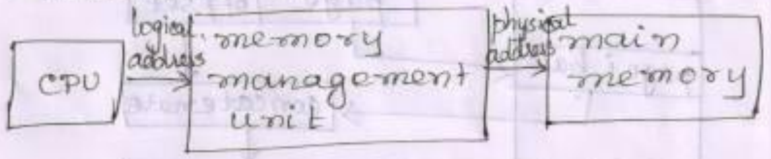
→ Alternative to cache.

→ s/w determines what is stored in scratch pad.

→ provides predictable behavior at the cost of s/w control.

Memory management Units

- Translate addresses



Memory management Tasks.

→ Allows programs to move in physical memory during execution.

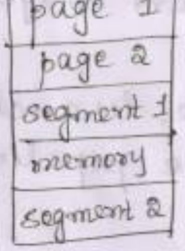
→ Allows virtual memory.
- → memory images kept in secondary storage
- → images returned to main memory on demand during execution.

Address Translation

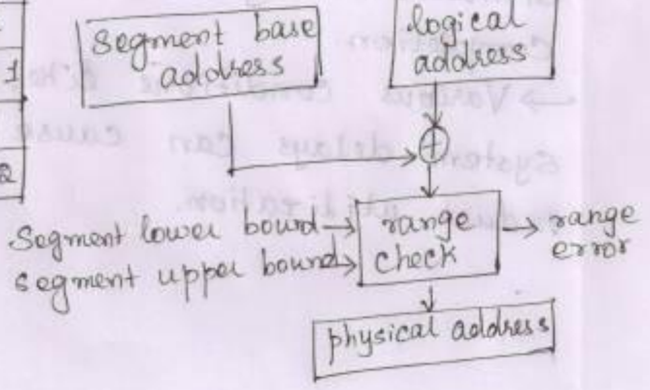
→ Requires some sort of register/table to allow arbitrary mappings of logical to physical addresses.

→ 2 basic schemes < segmented / paged

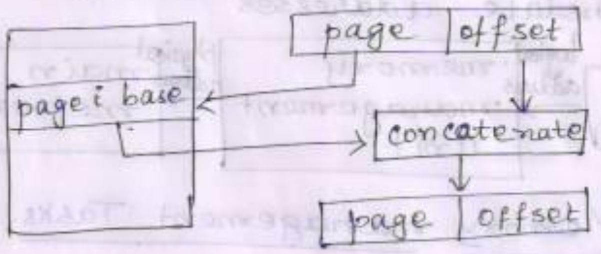
Segments and pages



Segment address translation



Page address translation



ARM memory management

→ Memory region types
- → section : 1 Mbyte block
- → large page : 64 kbytes
- → small page : 4 kbytes

→ An address is marked as section-mapped or page-mapped.

Elements of CPU performance

→ cycle time
→ CPU pipeline
→ Memory system

Pipelining

→ Several instructions are executed simultaneously at different stages of completion.
→ Various conditions like, branches, memory system delays can cause pipeline that reduce utilization.

Performance Measures.

→ Latency - time it takes for an instr. to get through the pipeline

→ Throughput - No. of instructions executed per time period.

Pipelining increases throughput without reducing latency.

ARM7 pipeline

→ ARM7 has 3 stage pipe

- ↳ fetch instr. from memory
- ↳ decode opcode and operands
- ↳ execute.

Delayed branch.

→ To increase pipeline efficiency, delayed branch mechanism requires n instructions after branch always executed whether branch is executed or not.

Memory system performance

→ Caches introduce indeterminacy in execution time, depends on order of execution.

→ Cache miss penalty → added time due to a cache miss.

Types of Cache misses

→Compulsory miss → location has not been referenced before.

→Conflict miss → 2 locations are fighting for the same block.

→ capacity miss → working set is too large.

CPU power consumption.

→ Most modern CPUs are designed with power consumption in mind to some degree.

→ Power vs. energy
- → heat depends on power consumption.
- → battery life depends on energy "

CMOS power consumption.

→ Voltage drops → power consumption ∝ V^2

→ Toggling → more activity means more power.

→ Leakage → basic circuit characteristics; Can be eliminated by disconnecting power.

CPU power saving Strategies

→ reduce power supply voltage
→ Run at lower clock frequency
→ Disable fn units with control signals when not in use
→ Disconnect parts from power supply when not in use.

Power management Styles

→ Static power management
- doesn't depend on CPU activity.

ex user-activated power down mode.

→ Dynamic power management
- based on CPU activity

ex disabling off- fn units.