Basics & Instruction

- CA: Instruction, modes, data format, CPU design
- Opcode necessary in each instruction (instruction: binary combination)
- No PC in 4-address instructions
- Variable length Instruction: Fixed Length Opcode
- Fixed length Instruction: Variable Length Opcode
- Autoinc: Postinc, AutoDec: Predec
- PC relative & Base reg mode supports relocation without any change in code
- PC relative mode offset: Negative for backward jumping
- In execution phase of branch instruction value of PC updated with appropriate address

CPU & Control Unit

- Cycle time = 1/ clock rate
- 1 instruction execution time = CPI * cycle time
- Program execution time = n * CPI * cycle time
- MIPS = $\frac{Number\ of\ instructions}{Execution\ time * 10^6} = \frac{Clock\ rate}{CPI * 10^6}$
- 2 CPU having same instruction set can have different CPI and clock rate
- In vertical microprogrammed one signal can be enabled from one group
- In vertical microprogrammed maximum signals can be enabled at once = number of groups
- Speed up = $\frac{Slower\ Technique\ time}{Faster\ Technique\ time}$
- Throughput: Number of operations per unit time
- Bandwidth: Data transferred per unit time

RISC vs CISC

Seq. No.	RISC	CISC
1.	Simple Instruction	Complex Instruction
2.	Fixed Length Instruction	Variable Length Instruction
3.	Simple and limited Addressing Modes	Complex addressing modes
4.	Less Number of Instructions	More Number of Instructions
5.	Easy to implement using hardwired control unit	Difficult to implement using hardwired control unit
6.	One cycle per Instruction	Multiple cycle per instruction
7.	Register-to-Register arithmetic operation only	Register-to-Memory & Memory-to-Register arithmetic operations possible
8.	More Number of Registers	Less Number of Registers

IO Organization

- Efficiency of asynchronous line = $\frac{Char\ bits}{Total\ bits\ sent\ per\ char}$
- Time in programmed IO = time to check status + time to transfer data
- Time in Interrupt IO = interrupt overhead + time to service interrupt
- CPU sends 2 information to DMAC before transfer: starting address & Data count
- DMAC can generate address and can send control signals for memory
- CPU wait for more time in burst mode as compared to cycle stealing mode
- No CPU waiting or blocking in interleaving mode of DMA
- % of time CPU blocked (burst mode)= $\frac{transfer\ time\ to\ memory}{prepatation\ time + transfer\ to\ memory\ time} * 100\%$
- % of time CPU blocked (cycle stealing) = $\frac{transfer\ time}{prepatation\ time} * 100\%$
- DMA is faster mode for transferring data between IO & memory
- Max data transferred using DMA without CPUs intervention = $2^x - 1$, x = bits in data count
- At a time only one of DMAC and CPU can use the system buses
- During instruction execution DMA transfer can be done but not the interrupt service

Memory Mapped IO	IO Mapped IO
1. Memory wastage	1. No Memory wastage
2. All Memory access instructions used for IO access also	2. IO access and memory access instructions are different
3. No separate address space for IO	3. IO have their own separate address space
4. More Instructions for IO access	4. Less Instructions for IO access
5. More addressing modes for IO access	5. addressing modes for IO Access
6. More IO devices connected	6. Less IO devices connected

Memory Organization:

- Byte addressable memory = 128KB = $128K \times 1$ B = $128K \times 8$ bits
- Memory access rate = $\frac{1}{\text{memory access (cycle) time}}$
- Memory access decoder = $a \times b$, a = address size, b = number of cells
- Multiplication table for 2 n-bit unsigned number = $2^{2n} \times 2n\ bits$
- Addition table for 2 n-bit unsigned number = $2^{2n} \times (n+1)\ bits$
- In multiple chip memory 1 decode output selects one entire horizontal arrangement
- If required addresses more => Vertical Arrangement
- If required data more => Horizontal Arrangement
- If required addresses & data more => Hybrid Arrangement
- Default storage unit: bits
- CPU can initiate the memory request only when memory is ready
- Associative memory is faster than SRAM (costlier too)

Static	Dynamic
1. Implemented using flip-flops	1. Implemented using capacitors
2. No refresh required	2. Periodic refresh is required
3. Faster Read/Write	3. Slow Read/Write
4. Used for Cache	4. Used for main memory
5. Low Idle power consumption	5. High Idle power consumption
6. High operational power consumption	6. Low operational power consumption

Cache Organization:

- Cache is implemented based on locality of reference
- Every time there is a read miss, a block is brought from mm to cm
- Every time there is a write miss, a block does not come from mm to cm (no write allocate)
- Simultaneous access $Tavg = H * tcm + (1 - H) * tmm$
- Hierarchical access $Tavg = tcm + (1 - H) * tmm$
- $Tblock$= Block size $* Tmm$
- $tavg$= tcm if H =100%
- Write Through:

  $Tavg\ write$ = Tmm

  $Tavg$ = % of read $*\ Tread + \%\ of\ write\ * Tavg\ write$

  $Effective\ hit\ rate = read\ hit\ ratio\ * \%\ of\ read$
- Write Back:

  Simultaneous $Tavg = H * tcm + (1 - H) * (tblock + x * tblock)$

  Hierarchical $Tavg = H * tcm + (1 - H) * (tcm + tblock + x * tblock)$

  x = % of dirty blocks

- Only 1 data sent to mm for write in write through cache
- In write though cache the block is replaced from cache directly
- In write back cache, the dirty blocks are only written back to mm
- CPU always generated mm address (even to access cache too)
- Tag identifies among all mm blocks which maps to one index, which one is present in cache
- Cm block number = $(mm\ block\ number\ )\%\ number\ of\ blocks\ in\ cache$
- MM address in direct mapping

Tag	Cm block number	Byte offset

- Index in direct mapping = cm block number
- Tag in direct mapping = mm address $- \log 2\ cache\ size$
- Byte offset is not used to check hit/miss in any of the mappings
- Tag directory size (all mappings) = $Number\ of\ blocks\ in\ cache * (tag + extra\ bits)$
- For a given cache size, block size and mm size: Tag is same (for byte and word addressable memory both)
- Cm set number = $(mm\ block\ number\ )\%\ number\ of\ sets\ in\ cache$
- MM address in set associative mapping

Tag	Set offset	Byte offset

- Index in set associative mapping = Set offset
- Tag in set associative mapping = mm address $- \log 2\ cache\ size + log2k$
- MM address in fully associative mapping

Tag	Byte offset

- Index in fully associative mapping = 0-bits
- Tag in fully associative = mm address $- \log 2\ \ block\ size$
- Size of tag maximum in fully associative and minimum in direct mapping
- Size of index minimum in fully associative and maximum in direct mapping
- In fully associative mapping, tag = mm block number
- Direct mapping hardware:
  - Number of MUX for tag selection= Tag-bits
  - Size of MUX for tag selection= Number of blocks : 1
  - Number of comparators = 1
  - Size of comparator = Tag-bits
- k-way Set associative mapping hardware:
  - Number of MUX for tag selection= k * Tag-bits
  - Size of MUX for tag selection= Number of set : 1
  - Number of comparators = k
  - Size of comparator = Tag-bits
  - OR-gate = 1 (k-input OR gate)
- Fully associative mapping hardware:
  - Number of comparators = Number of blocks in cache
  - Size of comparator = Tag-bits
  - OR-gate = 1 (number of blocks-input OR gate)
- Hit latency time:
  - Direct mapping = MUX delay + comparator delay
  - Set associative mapping = MUX delay + comparator delay + OR-gate delay
  - Fully associative mapping = comparator delay + OR-gate delay
- No any replacement policy required for direct mapping
- No conflict miss in fully associative mapping
- To reduce conflict miss: increase associativity
- To reduce cold miss: increase block size
- To reduce capacity miss: increase cache size
- Total cold miss = number of blocks in mm
- Simultaneous access $Tavg\ =\ H1\ * t1 +\ (1 - H1) * [H2 * t2 + (1 - h2) * tmm]$
- Hierarchical access $Tavg\ =\ t1 +\ (1 - H1) * [t2 + (1 - h2) * tmm]$

Disk

- Disk capacity = 2 * no. of platters * tracks per surface * sectors per track * sector capacity
- Sector is smallest addressable unit of disk which can be read or written at once
- Disk access time = Seek Time + Rotational Latency + 1 sector Transfer Time
- Average Rotational Latency = $\frac{1\ rotation\ time}{2}$
- 1 sector Transfer Time = $\frac{1\ rotation\ time}{number\ of\ sectors\ per\ track}$
- Sequentially stored N sector transfer time
  = Seek Time + Rotational Latency + N * 1 sector Transfer Time

- Randomly stored N sector transfer time
  = N * (Seek Time + Rotational Latency + 1 sector Transfer Time)
- Disk addressing $< c, h, s >$ c = cylinder number, h = surface number, s = sector number
- Sector number for given address
  = c * sectors per cylinder + h * sectors per track * s
- c = sector number / sectors per cylinder
- h = (sector number % sectors per cylinder) / sectors per track
- s = (sector number % sectors per cylinder) % sectors per track

Pipeline:

- Pipelining is useful, When same processing is applied over multiple inputs
- Pipeline time = $(k + n - 1) * tp$
- Speed up = $\frac{n*tn}{(k+n-1)*tp}$
- Ideal Speed up = $\frac{tn}{tp}$
- Ideal condition: k – 1 cycles ignored to fill pipe
- tp= max(segment delays) + register delay
- tn = sum of all segment delays
- Latency: After how much time new input given to machine
- Latency of pipeline: tp
- Latency of non-pipeline: tn
- Throughput of pipeline = 1 / tp
- Delayed load: Software solution for data dependency by provided by compiler
- Operand forwarding: Hardware solution for data dependency by provided by compiler
- For ALU to ALU data dependency, Operand forwarding provides zero stall cycles
- CPI_{avg} = 1 + (stall frequency * stall cycle)
- 1 Instruction execution time (average) = $CPIavg * tp$
- Stalls because of branch = i – 1 (if after i^{th} stage the condition is evaluated)
- Even branch is not taken then too stalls are there due to branch instructions
- Result of branch condition evaluation available after execution phase of branch instruction
- Operand forwarding and register renaming can not solve the memory access dependencies

Floating Point Representation:

- Number represented in form:

S	E	M

- Exponent is biased
- Mantissa is normalized
- Value (Explicit Normalization) = $(-1)s * 0.M * 2^{E-bias}$
- Value (Implicit Normalization) = $(-1)s * 1.M * 2^{E-bias}$
- More bits in exponent => Larger range
- More bits in Mantissa => Greater precision or accuracy
- Conventional representation can not store zero and very small numbers
- IEEE-754 Single precision 32-bits: Bias = 127

S	E	M
1	8	23

- IEEE-754 Double precision 64-bits Bias = 1023

S	E	M
1	11	52

- Special Number: E = all 0's OR E = all 1's
- Value (Implicit Normalization) = $(-1)s * 1.M * 2^{E-bias}$
- Value (Denormalized) = $(-1)s * 1.M * 2^{-126}$ Single precision
- Value (Denormalized) = $(-1)s * 1.M * 2^{-1022}$ Double precision

S	E	M	Number
0	00….0	00…..0	+0
1	00….0	00…..0	-0
0	11…..1	00…..0	+∞
1	11…..1	00…..0	−∞
0/1	11…..1	≠ 00…..0	NAN
0/1	00…..0	≠ 00…..0	Denormalized
0/1	≠ 00…..0 And ≠ 11…..1	xxxx…xxx	Normal Number (Implicit Normalized)