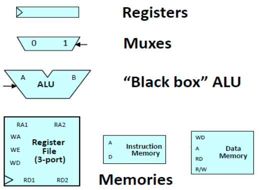
Building the Beta

Instruction Set Architecture + Digital Circuit Components: Hardware implementation of reg, PC, control logic & data paths

Digital Circuit Components:



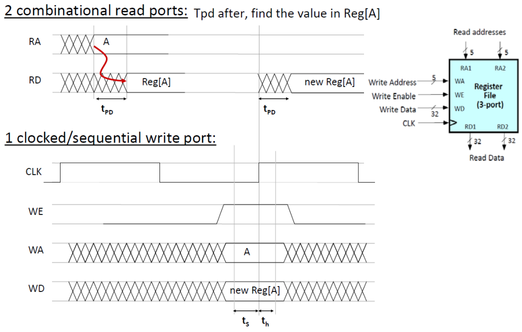
Classes of Instructions:

1. ALU: OP & OPC
2. LD & ST
3. Branches
4. Exceptions

Register File (RF):

- use of multiplexers (combinational logic) and registers (sequential logic)
- 2 combinational READ ports, 1 clocked WRITE port
- Static & Dynamic Discipline:

2 combinational read ports: Tpd after, find the value in Reg[A]



To read a value from RF, supply stable address input (RA) on one of read ports, after RF's Tpd, value of RA will appear on corresponding read data port (RD). For writing, write address, write data and write enable signals must all be valid and stable for Ts before rising edge of clock and for Th after rising edge of clock. The RF will then reliably update value of selected register.

- Add(R0,R1,R0) Read R0 at first then write R0 at the end.

Can read and write the same register in a single clock cycle. Read address becomes valid at beginning of cycle, old value of register will appear on data port for rest of cycle, then write occurs at end of cycle, new register value will be available in next clock cycle.

PC & Control Logic:

- **PC** is 32-bit register, least significant 2 bits always 0, is used as a **memory address** to fetch instruction (32-bit word) from **Instruction Memory**.
- **OPCODE** (6 most significant bits) is used by **Control Logic** to create control signals for ALU and other parts of β architecture.

Data Paths:

WERF - 1 when you wanna write into reg, 0 if not
Output of ALU goes in as Write Data to Register File
Sign Extend 16 bits constant into 32 bits. (Copy bit 16 to [32:17])
BSEL - what goes into B port of ALU, RD2 or SXT(Constant)
WR (1bit) - 0 if read, 1 if write
RA2SEL - select Rb/Rc to go into Read Port 2
WDSEL - select what goes into Write Port of Reg File
ASEL - select what goes into A port of ALU
*PC+4 & C -- combi logic --> (PC+4)+4*SXT(C) [at PCSEL=1]*
Ra (32bit) -- NOR gate --> Z=1 if all bits of Ra = 0
For BEQ: Z=1: branch, PCSEL=1
PCSEL=0 -> Incremented PC value is chosen
PCSEL=2 -> Value of Reg A is chosen
(expression) ? value if true : value if false

EG. implement STX(Rc,Rb,Ra) which is a shortcut for

ADD(Rc,Rb,Rb)	Mem[Reg[Ra]+Reg[Rn]] <- Reg[Rc]
ST(Rc,0,Rb)	Need to amend data path & reg file

Reg File needs another RA/RD port which could eliminate RA2SEL.

EG. implement LDX(Ra,Rb,Rc) which is a shortcut for

ADD(Rb,Ra,Ra)	Reg[Rc] <- Mem[Reg[Ra]+Reg[Rb]]
LD(Ra,0,Rc)	let ALUFN= '+', WERF=1, WDSEL=2, rest=0

Synchronous/software interrupts:

- Illegal OPCODE in instr word
- Reference to non-existent memory
- Divide by 0

IllOp: (forced by hardware)
PUSH(XP)
fetch inst at Mem[Reg[XP-4], fill result reg
POP(XP)
JMP(XP)

Asynchronous/hardware interrupts:

- User hits a key
- A packet comes via network
- In general: input from I/O hardware device

Exception Handling:

- Do not execute current instr
- Save current PC (actl PC+4) -> Exception pointer Reg (R30)
- Load PC with exception pointer (0x4 for synch ex, 0x8 for asynch ex, both 32bits)

Note that the interrupted instruction has not been executed. So if the exception handler wishes to execute the interrupted instruction, it will have to subtract 4 from the value in the XP register before performing a JMP(XP) to resume execution of the interrupted program.

Read-only Memory (ROM) (64 location) indexed by 6-bit OPCODE field is easiest way to generate the appropriate control signals for current instruction.
WR always have defined value, ensuring only write to memory during ST inst, and **WERF** is also well-defined except during RESET. During RESET, set WR=0, rest dc.

CPU Design Tradeoffs:

Max Performance: measured by no. of instructions executed per second
MIPS (Millions of Inst/sec) = Clock Rate (MHz) / Clocks per inst (CPI)
Benchmark Runtime = #inst executed (in millions) / MIPS
Min Cost: measured by size of circuit
Best Perf/Price: measured by ratio of MIPS to size.

SRAM (Fastest, most exp):

- Made up of 6 transistors & amp senses in each cell (inv needs 2 transistors; 2 nfets)
- Each cell stores 1 bit, loop between 2 inv can store 1 bit as long is it is connected to power.
- Word line is driven to HIGH during both READ or WRITE such that the cell is connected to both bit lines. When Word line=0, value in cell remains stable (can rmb)
- 2 bit lines: faster to use amp sensor to find out output bit, don't need to wait till value cross over invertor to be a valid 0/1.

DRAM: Tech of mem (slower; cheaper)

- A DRAM cell is made of 1 FET & 1 capacitor (stores the bit)
- Word line: is driven to HIGH when reading/writing
- Problem: capacitor will lose charge, DRAM has to be refreshed frequently to keep data intact ∴ slower than SRAM.

Decoding Address with SRAM/DRAM cells

- Billions of these cells are assembled together to form a large memory unit.
- Each cell (or groups of 8 cell in byte addressing) can have specific addresses
- To decode: split address into row & column, read hundreds of bits in parallel (one row contains hundreds of bits), and select which of the 32 bits we want using column address.

Eg. 8 location SRAM array, each location hold 6 bits of data. Each bit cell organised as 8 rows by 6 columns. 3 address bits to select one of 8 locations, address decoder logic sets 1 of 8 word lines HIGH to enable particular row (location) for upcoming access. Active wordline enables each of SRAM bit cells on selected row, connecting each cell to pair of bit lines. During READ op, the bit lines carry analog signals from enabled bit cells to sense amplifiers, converting analog signals to digital data. During WRITE op, incoming data driven into bit lines to be stored into enabled bit cells.

Disk: Tech of Secondary Storage (vslow, vcheap)

- Ideally, should not be accessed during run-time.
- Made up of spinning magnetic medium, unlike SRAM & DRAM, it is non-volatile (can store info even after power is cut)
- To READ/WRITE to a disk, device has to mechanically move head of disk and access particular sector.

GOAL: Perform with SRAM speed at cost of DISK

Locality of Reference: Reference to memory location X at time t implies that reference to X+△X at t+△t becomes more probable as △X,△t approaches zero

Evidence: local stack frame grows nearby to one another, program inst are near to each other, data (eg. arrays) also nearby one another.

Cache Idea:

- Cache contains temporary copies of selected memory locations A and its content Mem[A].

1. Look for requested info (from CPU) in cache
2. If in cache (hit), return contents to CPU
3. Otherwise (cache miss), look for info in physical memory, and subsequently, the disk

$$T_{avg} = \alpha T_c + (1-\alpha)(T_c + T_m) = T_c + (1-\alpha)T_m$$

α is hit ratio (the amount of information found in cache) Tc is cache access time and Tm is memory access time.

Fully Associative Cache (FA):

- TAG contains all bits of address A
- DATA contains all bits of content at A: Mem[A]
- Expensive; SRAM + comparator at each row
- Real Fast: parallel look up
- Flexible: memory + content can be stored on any TAG-DATA row
- Needs Replacement Strategy to decide which cache line to write to when cache is full
- Good when cache size is small, less impt when big

Direct Mapped Cache (DM):

- TAG contains T-upper bits of address A
- DATA contains all bits of content at A: Mem[A]
- Lower k-bits of A decides which row of DM cache
- Made of SRAMs; but cheaper than FA since only one comparator circuit per DM cache
- No PARALLELISM, but fast mapping between address & cache line index
- Inflexible: a unique combi of K-bits of A is mapped to exactly one row of DM cache
- Suffers from contention, 2 different addresses can be mapped to same location if both has same K-lower bits. Selecting K-lower bits for mapping is better than T-upper bits due to LOR but does not completely eliminate contention

$k = log_2(\#rows)$ EG. 3bits if 8 rows
FA cache size = any; DM cache size = 2^k

N-way Associative Cache:

- N DM caches in parallel
- Cells in same 'row' belong to same set
- Cells in same 'column' of DM caches belong in same cache line
- Given same k-bits lower address, can be stored in any N cache lines in same set
- However, different combi of k-bits lower address will have to be direct mapped on a different set
- Lookup operation: Parallel op of N-DM cache, each with 2^k lines in cache line.
- M-size N-way cache: no. of rows = M/N; then calculate bits needed

EG. 4-word 2-way => 2 rows, 1 bit

Replacement Strategy:

1. LRU: Least Recently Used
- Good when N is small, need to keep ordered list of N items (N! orderings)
- Overhead=O(log2(N!))=O(Nlog2(N) "LRU bits" for FA (/set for NW)
- Complex logic to re-order list after each cache access

2. FIFO/LRR: Least Recently Replaced
- Replace oldest item
- Overhead is O(log2N) bits/set. (just need one pointer that tells us oldest item within each set; diff pointers for diff sets(NW))

3. Random
- Use pseduo random generator to get reproducible behaviour
- Good when N is huge
- Overhead=log2(N)/cache to point to which line to replace

No best replacement strategy, one replacement strategy can be better depending on cases.

Increase Block Size

- Take advantage of locality, reduces size of tag memory
- Con: Fewer blocks in cache
- Increase size of data field, no. of data words in each cache line = block size, always a power of 2.
- Fetch a blocks of words tgt on a cache miss, trading increased cost of miss against increased probability of future hits.

In β architecture, address is 30bits if word addressing is used. k: 2 bits more when byte add

Add 2 extra bits of storage in cache: V & D.

V: Valid Bit - indicates if particular cache row contains data from memory and not empty or redundant value. Only check cache lines with valid bit=1
D: Dirty Bit - set to 1 iff CPU writes to cache and it hasn't been stored to memory (memory is outdated)

LRU Bit: (Not written) present in each cache line (for FA) and each cell in N-way regardless of block size because R/W with block size more than 1 is always done in parallel.

#bits required for each LRU indicator is log2(N) where N=number of ways

EG. 4-way, 8 rows (4x8 items in total) so 32 LRU cells, each size of 2 bits.

Cache Writes:

1. Write-through

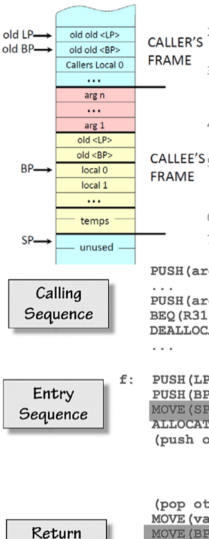
CPU writes are done in cache first by setting TAG=Addr, DATA= new Mem[Addr] in available cache line, but also written to main memory immediately. Stalls CPU until write to memory is complete. Memory never outdated.

2. Write-Behind

CPU writes are also cached, and write to main memory is immediate but is buffered. CPU keeps executing next inst while writes are completed (in order) in background

3. Write-Back (Most used)

CPU writes are also cached, but not immediately written to main memory. MM contents can be 'stale'. When entry in cache, safely just write into cache, leaving memory entry incorrect. But when cache is replaced, write into memory. This requires dirty bit in cache.



1. Caller pushes arguments onto stack, in reverse order
2. Caller branches to callee, putting return address (current PC) into LP
3. Callee pushes LP & BP, sets BP=SP, allocates local vars, pushes any used registers
4. Callee performes computation, leaving return value in R0
5. Callee pops registers, (deallocates local vars,) sets SP=BP, pops BP & LP
6. Callee branches to return address (LP)
7. Caller pops/deallocates arguments from stack

Calling Sequence

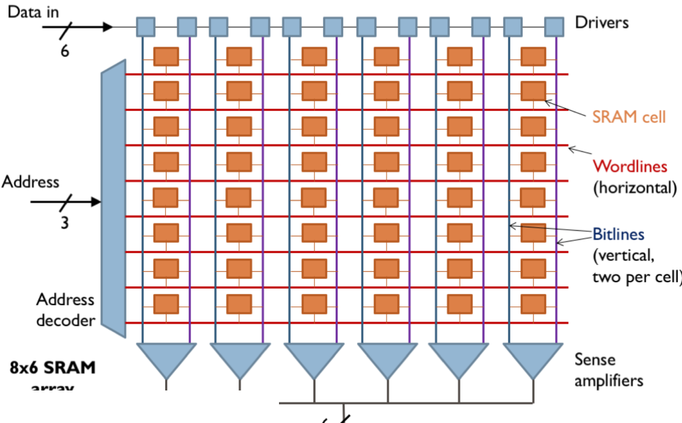
```
    PUSH(arg_n)                | push args, last arg first
    ...
    PUSH(arg_1)
    BEQ(R31,f, LP)             | Call f.
    DEALLOCATE(n)              | Clean up!
    ...                        | (f's return value in r0)
```

Entry Sequence

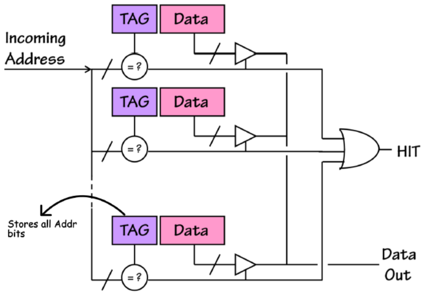
```
f:  PUSH(LP)                   | Save LP and BP
    PUSH(BP)                   | in case we make new calls.
    MOVE(SP,BP)                | set BP=frame base
    ALLOCATE(nlocals)          | allocate locals
    (push other regs)          | preserve any regs used
```

Return Sequence

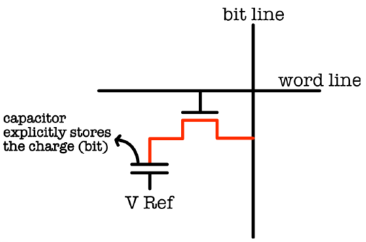
```
    (pop other regs)           | restore regs
    MOVE(val, R0)              | set return value
    MOVE(BP,SP)                | strip locals, etc
    POP(BP)                    | restore CALLER's linkage
    POP(LP)                    | (the return address)
    JMP(LP,R31)                | return.
```

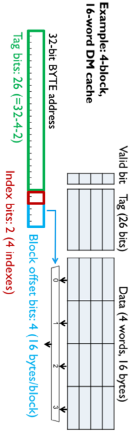


FULLY ASSOCIATIVE CACHE



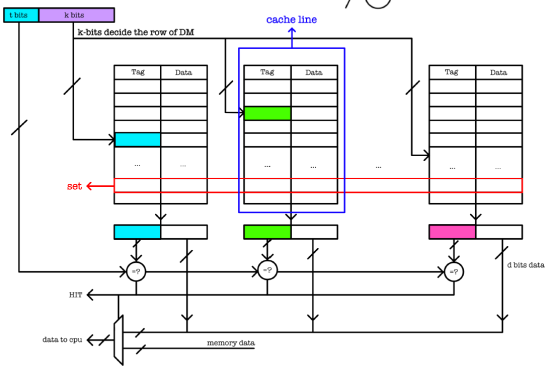
DRAM CELL



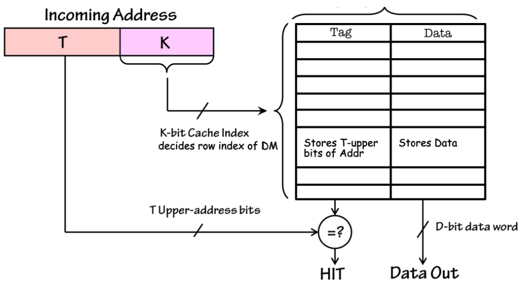


Dec	Hex	Bin
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000

N-WAY SET ASSOCIATIVE CACHE



DIRECT-MAPPED CACHE


Incoming Address
T
K
K-bit Cache Index
decides row index of DM
Tag
Data
Stores T-upper
bits of Addr
Stores Data
T Upper-address bits
=?
D-bit data word
HIT
Data Out