Encoding & MIPS

14 October 2017 21:36

Architecture:

- Instruction Set Architecture - **ISA**

- machine organisation - **micro architecture** - circuitry

Simple ISA can lead to simple micro arch,
too simple will cause hardware to be complex
try to achieve equal balance in complexity

Instruction:

- **Opcode**
  - the action - verb
- **Operand**
  - objects - e.g. register, mem address, immediate value

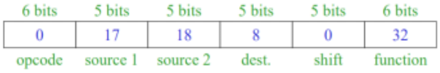
- a vector of bits
  - order is important
- DECODE raw bits before Execution
  - find opcode
    - what action
  - amount of operands
  - what are the operands

MIPS:

- architecture
  - 32 registers
    - **makes reg allocation simple for compiler**
    - **uniform treatment - use ad both address or data registers**
    - **separate multiply and divide registers**
    - reg file has 47 non-control input bits
    - need to address with 5 bits
      - □ affects instruction width
    - **exploit temporal locality** - store most recent variables
  - 32-bit wide
  - $0 (register zero) wired to 0
    - use ADD with $0 for MOV
  - load-store architecture
    - immediate offset to reduce instr count
    - from base register + offset
      - □ **exploit spatial locality**
      - □ data around a common address used next
  - destination register - comes first in assembly
  - Big-endian
  - 4 byte words
  - goal: minimise memory access
    - as access is slow

- has 3 formats of instructions
- Opcode - 6 MSB

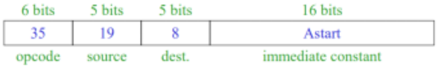
  - **R-type**



6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
0	17	18	8	0	32
opcode	source 1	source 2	dest.	shift	function

    - Opcode for R type - **000000**
    - get action from shift / function bits
    - **Also - used to Jump**
      - □ jr $ra # jump to address in registrar ra

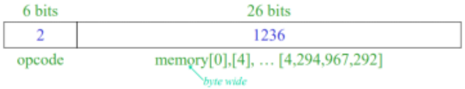
  - **I-type**



6 bits	5 bits	5 bits	16 bits
35	19	8	Astart
opcode	source	dest.	immediate constant

    - Opcode - **??????** - anything else
    - source - register - (contains base address)
    - immediate constant - contain offset
    - Uses:
      - □ memory access
      - □ conditional branches (BEQ / BNE..)
      - □ arithmetic involving constant - e.g. $01 = $01 + 1

  - **J-type**



6 bits	26 bits
2	1236
opcode	memory[0],[4], … [4,294,967,292]

    - Opcode - 00001? - last bit can be either 1 or 0
    - Standard Jump - unconditional
    - Jump And Link - JAL - save address of next instruction before jumping

Control & Data Paths

21 October 2017 17:00

Datapath - **result of calculation**

- separate datapaths for
    - register-based
    - memory access
    - branching
- combine datapaths
    - use multiplexers
    - minimal HW copies
- add control signals
    - for MUX's and the ALU control

Single Cycle Datapath Control

- combinatorial circuit - single cycle
- opcode - 6-bit input
- 9-bit output
    - MUX control
    - ALU control
    - read / write to memory and register

Control Path

- PC Source
    - increment or branch
    - signalled from ALU zero out
- Derived from Opcode
    - Reg Destination
    - MemtoReg
    - ALU Source
- ALU - control
    - combinatorial circuit
        - inputs Opcode and Function Code

Control

- data -independent - just find out what to calculate
    - might include memory access
- data-dependant - decide if branching
    - minimise - can take more than one cycle to branch

ALUs

29 October 2017 22:56

Zipping:
- Zig-zag pattern from two inputs into gates
  - lots of crossover of traces
- bad because long wire length
  - wire capacitance - charge time
  - resistance - heat output

Ripple Carry Adder:
- single bit full adders in series
- carry propagates through full adders

Saturation:

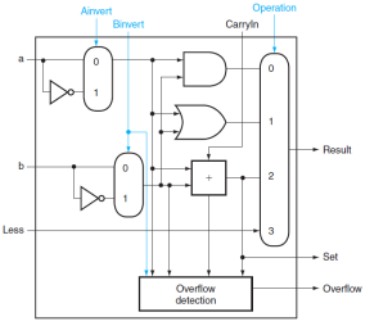
- Once max value reached - addition keeps value at the maximum
- can't overflow like a cycle

Barrel Shifter:
- not part of ALU
  - reason why shifting is done outside of ALU

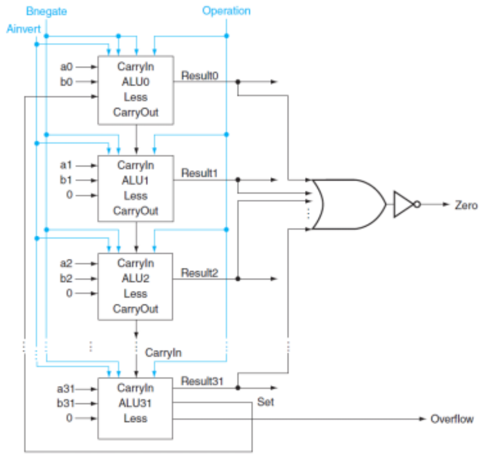
Basic ALU:

- single bit ALU
  - can do add, subtract, logic operations
    - does all at same time
  - MUX selects correct output from block
  - connect these in series
- control
  - sends signal for add vs subtract
  - send carry in - which ripples through
  - control signals for MUX
- Overflow detection
  - 4 inputs
    - MSB of A
    - MSB of B
    - Carry Out
    - Add / Subtraction control signal
  - 1 output
    - overflow detection
- Set on less than
  - logic comes from MSB ALUb
  - output - set - connected to 'less' input for LSB ALUb
  - rest of less input = 0
- Branch
  - test if A-B=0
    - zero result
    - obtain by NOR results form ALUb's

Single Bit ALU - MSB specific for MIPS



32-bit MIPS Ripple ALU



Fast Addition

- decrease in time means increase in area cost
- "infinite hardware"
  - CarryIn for any bit n
    - can be expressed in terms of A B and Cin0
    - requires VERY LARGE amount of logic for wide adders
- Carry-lookahead
  - First Level of abstraction
    - generate $gi = ai\,.bi$
    - propagate $pi = ai + bi$
  - second level of abstraction
    - Super Generate and Propagate
      - for each block of smaller adder
  - carry doesn't have to propagate through as many gates

$$gi = ai \cdot bi$$
$$pi = ai + bi$$

Using them to define $ci + 1$, we get

$$ci + 1 = gi + pi \cdot ci$$

Multiply

- Basic Multiply Algorithm
- Uses shifting and partial sums - 32 cycles

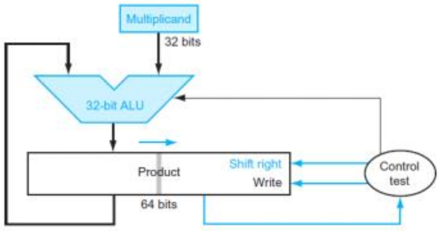


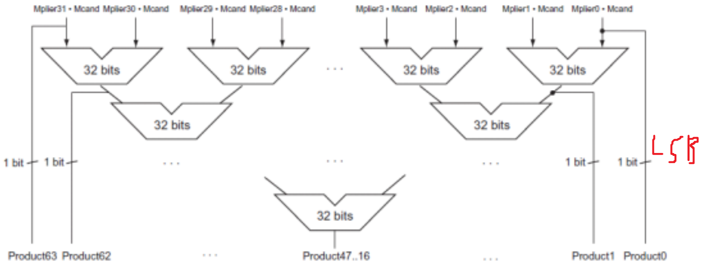
FIGURE 3.5 Refined version of the multiplication hardware. Compare with the first version in Figure 3.3. The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register. These changes are highlighted in color. (The Product register should really be 65 bits to hold the carry out of the adder, but it's shown here as 64 bits to highlight the evolution from Figure 3.3.)

Division- restoring divider:
- $\boldsymbol{dividend = (qoutient\ * divisor) + remainder}$
  - **remainder and dividend SAME sign**
- continue until
  - **(dividend == (q*divisor) +r) || (r>- divisor)**

- restoration algorithm

More covered on Chapter 3 Ed 5

Fast Multiply:
- Detect which bits of multiplier are set
- use these with a tree of adders



  - Input to head adders: Multiplicand AND'd with bits 0,1
- even faster:
  - use carry look ahead adders
  - pipeline the design

Screen clipping taken: 24/12/2017 21:10

ISAs

29 October 2017 23:01

Traditional View	RISC	CISC
Compiler	clever	dumb
CPU	dumb	clever

Temporary Storage

- Stack
  - push pop
  - top of stack - operands
- Accumulator
  - only the one temporary operand
- Register
  - explicit operands
  - register faster than memory
  - reduce memory access
  - compiler friendly

register-register
ALU op requires no memory access
register-memory
ALU op with 1 operand in memory
memory-memory
ALU operates on memory

x86:

- Intel's primary concern was backwards compatibility
- Pre-existing user base made AMD adopt it
- EIP- instruction pointer
- SP- Stack Pointer
- x86 defines the way the function call stack looks like in C
- x86 has Condition Codes like ARMv8
- 4 MIPS instr req. to implement x86 MOVSL
- C library 'memcpy' inspired addition of MOVSL to x86
- x86 'string' instructions are too slow to be used for modern C

Temporary Storage	Pros	Cons
Stack	• simple eval model • dense code	• no random access • slow if stack located in memory
Accumulator	• short instructions • min. internal storage	• increased memory access
Register	• code generation follows a general model • register access is fast	• have to name every operand • long instructions

GPR type	Pros	Cons
reg-reg	• fixed length instructions • similar cycle times • simple code generation model for compilers	• higher instruction count • wasted instruction bits for small instructions
reg-mem	• access data without load • dense code	• instruction time in cycles depends on operands • not enough bits in instruction for entire memory address
mem-mem	• even more dense code • not waste registers for temporaries	• instruction time and size varies • slower due to memory access

Performance

25 November 2017 14:15

Always want measure relative to TIME

CPI: average clock cycles per instructions
IPC: inverse of CPI

$$Execution\ time = \frac{1}{clock\ rate} * number\ of\ instructions * CPI$$

Use **ratios of execution time** to compare different processors

Aspect	instr count	CPI	clock rate
program	X	X	
compiler	X	X	
ISA	X	X	X
$\mu arch$		X	X
technology		X	X

<u>RISC ISA principles:</u>

- reduce CPI - common case fast
- simple and regular format
  - reduce number of GPRs
  - easier to implementation
- Compromise:
  - increased code size
  - need better compilers

<u>Reducing Power:</u>

- $Power = CV^2F$
- cannot reduce voltage anymore
- need new ways to improve performance other than frequency

<u>Improving Performance:</u>

- make use of caches
- pipelining, multi-thread, multi-cpu
- FPGA/ASIC
- GPUs, async designs

Different types of evaluation:

Method	Pros	Cons
user target workload	• representative	• specific, non-portable • difficult to measure performance • hard to identify problem causing low scores
full application benchmark	• portable • wide spread • performance quantified	• less representative
small kernel benchmarks	• easy to use • early in design cycle • identify peak performance	• peak performance far away from typical performance

Memory & Caches

29 October 2017 23:01

temporal locality
- recently used items tend to be used again soon
- what to replace when out of space in a level of memory

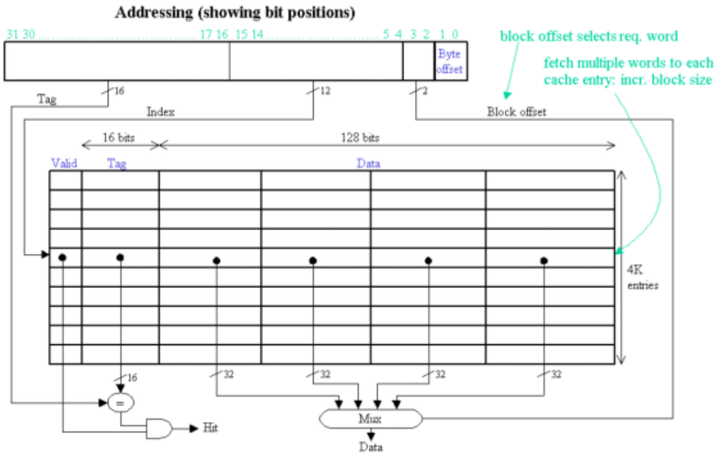
spatial locality
- items close to recently used items tend to be used next
- how many successive items to store when accessing one of them

can introduce padding in software structures to align with memory design

Cache:
- organised in **blocks** of one or more words
- **hit**: CPU finds required block in cache
- **miss**: doesn't find, fetches from memory
- **hit rate**: ratio of reading from cache to memory
- **hit time**: cache access and determining if hit
- **miss penalty**: time to replace into cache and then transfer to CPU
- separate I and D caches
  - increases bandwidth to memory

Block Size - multiple words per block:
- exploits spatial locality
- helps smaller caches more
  - unless block size becomes too large
- no latency penalty, unless oversaturate bandwidth
- on **write miss, need to read block into cache**
  - then write word

Cache Table: (DMap)

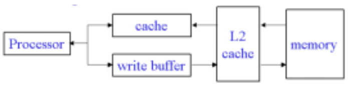

Addressing (showing bit positions)
31 30
17 16 15 14
5 4 3 2 1 0
Byte offset
block offset selects req. word
fetch multiple words to each cache entry: incr. block size
Tag
16
Index
12
2
Block offset
16 bits
128 bits
Valid
Tag
Data
4K entries
16
32
32
32
32
=
Hit
Mux
Data

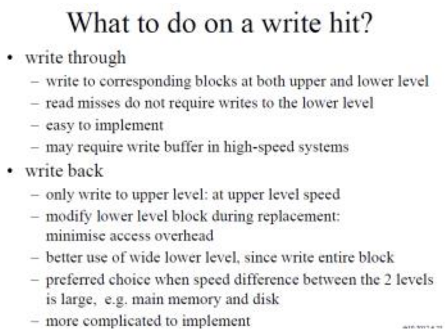
Handling a **miss on read:**
- miss instruction read
  - restore PC
  - stall CPU and wait on memory
  - re-fetch by un-stalling CPU
- miss data read
  - stall CPU
  - wait until data from main mem in cache
  - more common

Types of miss:
- compulsory - first access to a block
- capacity - insufficient space
- conflict - placement scheme not flexible

Decision	decrease miss rate	possible neg effect
^size	capacity	^access time, ^hardware
^associativity	conflict	^access time, ^hardware
^block size	compulsory	^miss penalty

Writes:
- **write back**
  - only write to cache
  - need 'dirty' bit for every entry
  - have to flush entry when it's replaced
- **write through**
  - write to both cache and memory
  - memory write bandwidth can cause a bottleneck
    - insert buffer between cache and memory
    - CPU write to cache and buffer
    - Memory Controller (IMC) write buffer to memory
      - □ FIFO queue
        - ◆ store frequency << 1/ memory write cycle time
        - ◆ CPU freq !>> mem freq
          - ◇ else get write buffer saturation
          - ◇ solution: L2 cache





What to do on a write hit?

- write through
  - write to corresponding blocks at both upper and lower level
  - read misses do not require writes to the lower level
  - easy to implement
  - may require write buffer in high-speed systems
- write back
  - only write to upper level: at upper level speed
  - modify lower level block during replacement: minimise access overhead
  - better use of wide lower level, since write entire block
  - preferred choice when speed difference between the 2 levels is large, e.g. main memory and disk
  - more complicated to implement

Cache Performance:

$$CPU\ time = \left[n_{instr}.CPI + R.R_{missrate}.R_{misspenalty} + W.W_{missrate}.W_{misspenalty} + WBufferStall\right].clock\ cycle$$

$$\%\ time\ on\ CPU\ stall = \frac{i_{nstrmissrate}p_{misspenalty} + d_{atamissrate}f_{loadstorefreq}p}{CPI + ip + dfp}$$

(assumes hit time and WBufferStall insignificant / doesn't happen)

Average Memory Access Time:

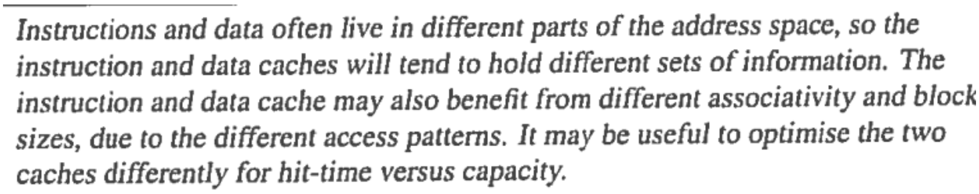
$$AMAT_{CPU} = HitTim_{L1} + MissRate_{L1}[HitTim_{L2} + MissRate_{L2}(AMAT_M)]$$

Direct Mapped Cache

- many memory locations mapped to a single cache location
- Algorithm:
  a. $word_{address} = \frac{byte_{adrress}}{w}$
  b. $block_{address} = \frac{word_{adress}}{k}$ (which block in cache)
  c. $block_{index} = block_{address} \% N_B$ (which row)

Cache Features:

- word size : $w \geq 1$, $w\ in\ bytes$
- block size : $k \geq 1$, $k\ in\ words$
- cache size : $c \geq k$, $c\ in\ words$
- Number of blocks $N_B = \frac{c}{k}$



Instructions and data often live in different parts of the address space, so the instruction and data caches will tend to hold different sets of information. The instruction and data cache may also benefit from different associativity and block sizes, due to the different access patterns. It may be useful to optimise the two caches differently for hit-time versus capacity.

Screen clipping taken: 13/05/2018 18:43

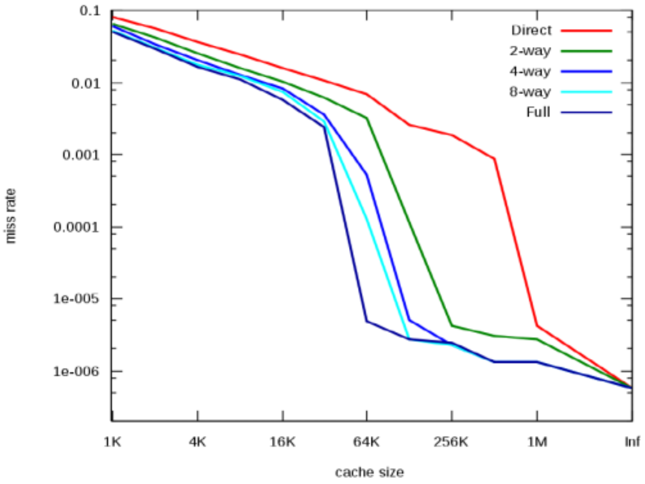
Placement Scheme	Description		
Direct Mapped	each address maps to one block **only set = all blocks**	• fast access	• high miss rate • no choice to NOT replace frequently used block
Fully Associative	each address maps to any block **one set = one block**	• low miss rate	• costly - uses hardware to implement algorithm to decide block to replace • slow - algorithms searches every block
n-way Set Associative	each address maps to n-blocks **one set = n blocks**	number of sets: $s = \frac{c}{n * k}$ $S_{index} = B_{address} \% s$	increased associativity: • **more blocks per set,** increased n • fewer sets per cache, decreased s • more choice on which block to replace with larger n • more search overhead: n compares per lookup

Block Replacement Policy:

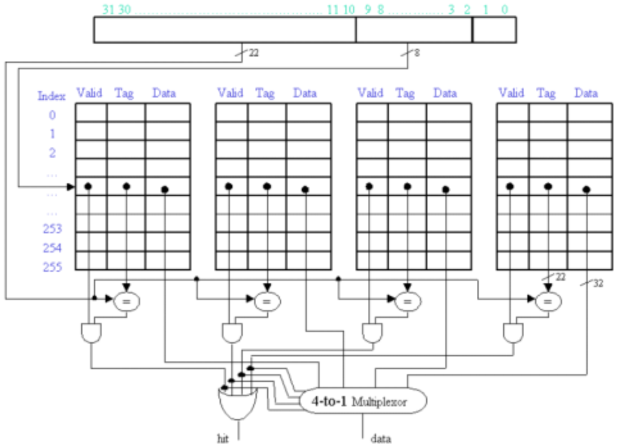
- (fully or set associative)
- Option 1
  - random
- Option 2
  - LRU
  - As Cache size increases
    - miss rate decreases
    - hence LRU > random

Cache Features:

- word size : $w \geq 1$, $w\ in\ bytes$
- block size : $k \geq 1$, $k\ in\ words$
- cache size : $c \geq k$, $c\ in\ words$
- Number of blocks $N_B = \frac{c}{k}$



4-way set associative cache



w = bytes/word	k = words/block	s = blocks/set	c = bytes/cache

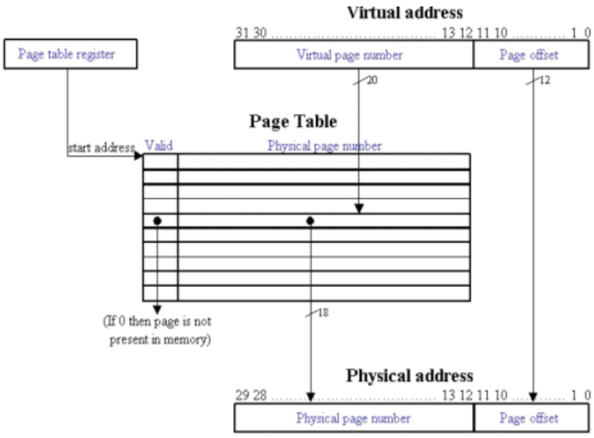
Virtual Memory

24 December 2017 21:24

Each program compiled using a virtual address space
Use main memory as 'cache' for disk

- **page** - fixed size virtual block
- **page fault -** page miss from physical memory
- **relocation** - can place page anywhere in physical memory
- **address translation** - map virtual address to physical or disk address

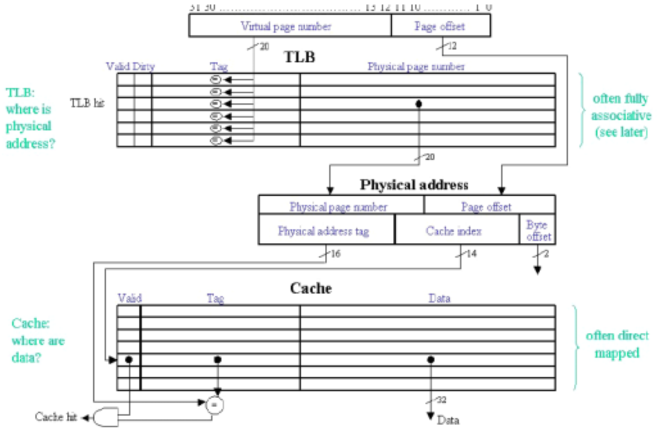
Page Table(per process):



- large page size to make up for long access time
- clever page-replacement to reduce miss rate
- write-back policy
  - dirty bit in table
    - copy back when set during replacement
- Page fault:
  - OS exception (SWI)
  - locate page in disk
  - use LRU to replace if no space in main memory
    - considers temporal locality
    - most recently used pages likely to be used again, so don't replace those
  - Requires EPC and Cause register:
    - EPC - saves address of offending instruction
    - Cause reg - values indicates cause of exception, tells OS what to do
    - Step 1: find virtual address
      - in EPC if instruction page fault
      - compute from base register + instruction offset if data
    - Step 2:
      - look up page table to find disk address
      - page replacement
      - copy page to main memory, update valid bit and physical page number (page number replaces disk address in tablee)
  - once OS dealt with fault, (avoids writing to reg/mem as not saved)
    - PC <- EPC

Translation Look-aside Buffer:

- on memory access:
  - on TLB hit - get physical page address
  - on TLB miss - load from page table into TLB and retry
- Works with normal cache
  - provides physical address which is used to search in cache as normal
  - causes delay before accessing L1 cache
- TLB is a special cache itself, for page table



- TLB miss:
  - handled in hardware
  - if miss:
    - page in memory, update TLB
      - valid bit set - update TLB with physical address
    - page not in memory, OS control
      - valid bit not set- page fault

Page Table protection:

- page table can have Read Only, Read/Write, Executable as well as valid bits
- CPU supports user and supervisor execution modes
- user process runs in user, can:
  - access its own page table by reading page table register
- OS runs in supervisor mode, can:
  - change all page tables
  - change page table register
  - clear TLB for a new user process

Pipelines

29 October 2017 23:02

What is it: split up combinatorial circuit by **pipeline registers**
Different instructions in different stages: parallelism
Benefits:

- shorter cycle time
  - determined by slowest stage
- reduce power consumption - as reduces glitches

Pipelining a processor:

- balance the propagation delay in different stages
- resolve data and control dependencies (hazards)

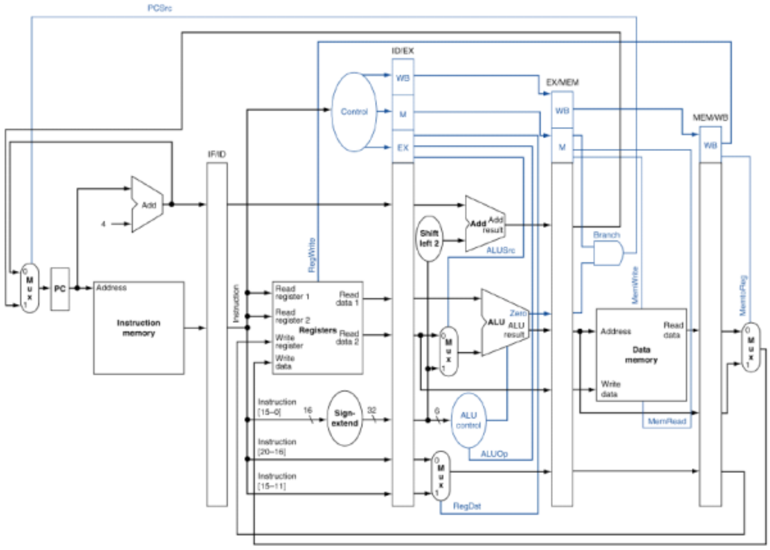
Performance:

- speedup due to increased throughput
- peak speedup when all stages evenly balanced
- normally increases latency a little
  - due to hazards

MIPS ISA was designed for pipelining:

- fixed instruction width
  - easier for IF and ID to be one cycle each
- few and regular instruction formats
  - decode and register read done is same step
- load/store
  - calculate address in 3rd stage, access memory in 4th
- alignment of memory operands
  - memory access (IMC) takes only one cycle

MIPS 5-stage Pipeline:



Stages:

- IF, instruction fetch from memory (I cache)
- ID: instruction decode & register read
- EX: execution
- MEM: access memory if instruction requires
- WB: write result back to register

Controls Signals are passed through the pipeline Registers

Hazards

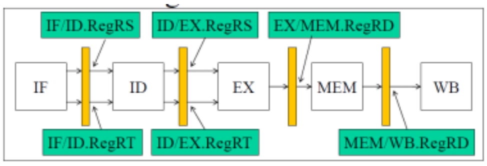
25 December 2017 18:58

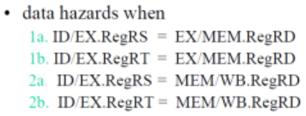
Hazard: Current instruction depends on previous instruction

Structural Hazard: required resource is busy
Data Hazard: need to wait for previous instruction to complete its data read/write
Control Hazard: control action decision depends on previous instruction
(most common hazard for a compiler program)

Data Hazards:

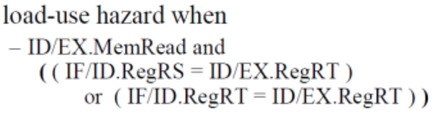
- need data from previous instruction in the pipeline
- can stall CPU to adjust - but causes big performance loss
- Solution: forwarding results
  - do not wait for result to be stored in a register
  - requires extra connections in the datapath
  - doesn't work for load data hazard
    - result has to be loaded from mem, data not ready
      - a change to ISA of no register-based mem access removes this
    - Must cause one-cycle stall
    - reduced by compiler:
      - reorder code with loads at start
- Detection:





- data hazards when
  - 1a. ID/EX.RegRS = EX/MEM.RegRD
  - 1b. ID/EX.RegRT = EX/MEM.RegRD
  - 2a. ID/EX.RegRS = MEM/WB.RegRD
  - 2b. ID/EX.RegRT = MEM/WB.RegRD

Load-Use data hazard detection:

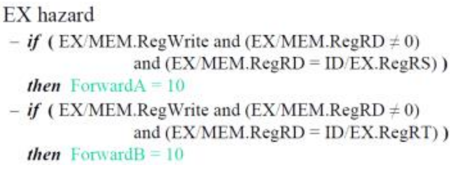


load-use hazard when
– ID/EX.MemRead and
((IF/ID.RegRS = ID/EX.RegRT)
or (IF/ID.RegRT = ID/EX.RegRT))

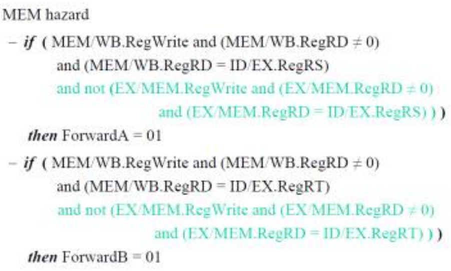
How to Stall:

- force ID/EX registers to 0 - nop
- prevent update of PC - current instr decoded again
- data forwarded in next cycle

Need a Forwarding unit in HW
Carries out this function:



```
EX hazard
– if ( EX/MEM.RegWrite and (EX/MEM.RegRD ≠ 0)
            and (EX/MEM.RegRD = ID/EX.RegRS) )
  then ForwardA = 10
– if ( EX/MEM.RegWrite and (EX/MEM.RegRD ≠ 0)
            and (EX/MEM.RegRD = ID/EX.RegRT) )
  then ForwardB = 10
```



```
MEM hazard
– if ( MEM/WB.RegWrite and (MEM/WB.RegRD ≠ 0)
       and (MEM/WB.RegRD = ID/EX.RegRS)
       and not (EX/MEM.RegWrite and (EX/MEM.RegRD ≠ 0)
                 and (EX/MEM.RegRD = ID/EX.RegRS) ) )
  then ForwardA = 01
– if ( MEM/WB.RegWrite and (MEM/WB.RegRD ≠ 0)
       and (MEM/WB.RegRD = ID/EX.RegRT)
       and not (EX/MEM.RegWrite and (EX/MEM.RegRD ≠ 0)
                 and (EX/MEM.RegRD = ID/EX.RegRT) ) )
  then ForwardB = 01
```

Structural Hazards:

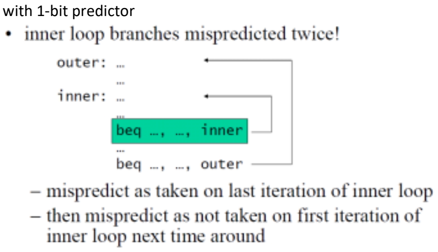
- MIPS has single memory
  - instruction fetch
  - load/store
- to solve:
  - require separate I and D caches
  - (or separate instruction / data memories)

Control Hazards:

- fetching next instruction depends on outcome of branch
- in MIPS:
  - Option 1: just stall one-cycle until branch outcome calculated
  - Option 2: compare registers and get target early in the pipeline
    - have to add HW in ID stage
  - Option 3: predict that branch not taken
    - only stall if prediction is wrong
  - Option 4:
    - move branch detection and address calculation to ID stage
      - causes one stall - cannot remove
        - the instruction proceeding branch instruction would then have to become nop
      - MIPS ISA - instead of nop, proceeding instruction in AWLAYS executed
    - data hazards on branches:
      - comparison register used in 2nd or 3rd preceding ALU instruction as destination register
        - can solve this via forwarding
      - 1 stall (beq turned into nop) caused by
        - comparison register used in 2nd preceding instruction which is a LOAD to the register
        - comparison register used in preceding ALU instruction as destination register
      - 2 stalls caused by
        - comparison register used in preceding instruction which is LOAD to the register

Dynamic Branch Prediction:

- for deeper and superscalar pipelines branch penalty is much more significant, so have to predict to reduce unavoidable stalls
- Branch Prediction Buffer
  - stores history of recent branches
  - branch *instruction* address is used as the index
  - stores outcome (branch (not) taken)
  - on a branch:
    1. check table, expect same outcome
    2. start fetching from target
    3. if wrong, flush pipeline and update prediction
- issue with 1-bit predictor

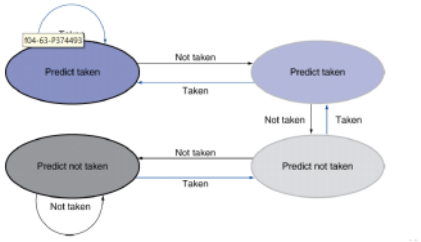


- inner loop branches mispredicted twice!

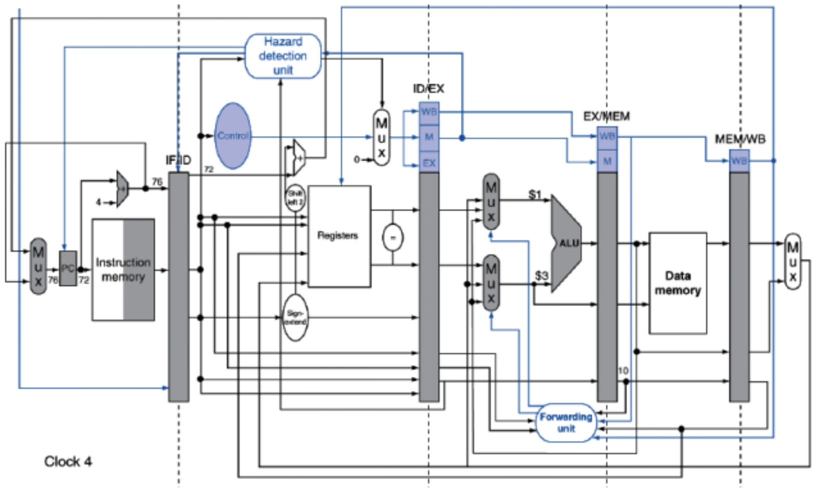
```
outer: …
       …
inner: …
       …
       beq …, …, inner
       …
       beq …, …, outer
```

– mispredict as taken on last iteration of inner loop
– then mispredict as not taken on first iteration of inner loop next time around

- 2-bit predictor

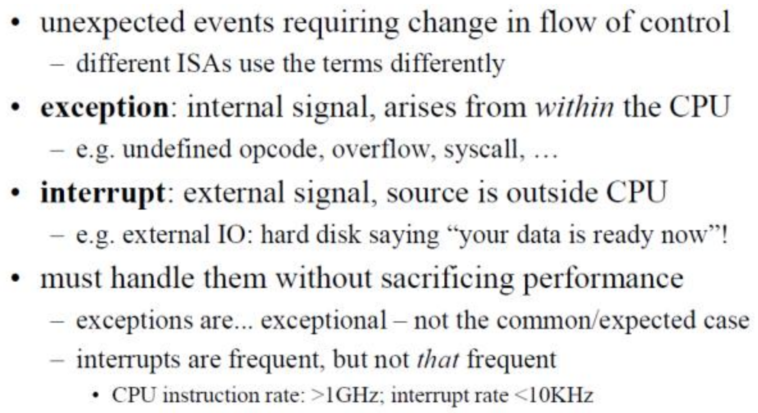


- Branch Target Buffer
  - still have to calculate target if taking branch
    - forces 1-cycle stall to calculate target address
  - use a cache to store recent target addresses
    - avoid stall
    - indexed by PC value during IF
      - entry added every cycle
    - can use value immediately if cache hit
      - else must stall and calculate, as well as update data field
      - must cleared for new process


Hazard detection unit
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Control
M u x
0
IF/ID
76
72
4
PC
Instruction memory
Shift left 2
Registers
=
Sign-extend
$1
$3
ALU
Data memory
10
Forwarding unit
Clock 4

Exceptions & Interrupts

29 October 2017 23:02



- unexpected events requiring change in flow of control
  - different ISAs use the terms differently
- **exception**: internal signal, arises from *within* the CPU
  - e.g. undefined opcode, overflow, syscall, …
- **interrupt**: external signal, source is outside CPU
  - e.g. external IO: hard disk saying “your data is ready now”!
- must handle them without sacrificing performance
  - exceptions are... exceptional – not the common/expected case
  - interrupts are frequent, but not *that* frequent
    - CPU instruction rate: >1GHz; interrupt rate <10KHz

Never finished lecture:

- EPC - exception PC
- 2 methods
- Cause Register vs vectored interrupts
- CR - OS uses value to figure out what to do
- VI - OS knows what to do by reading what address it started execution from
  - places 8 bytes apart