

# COMPSYS304 Notes 2017

Theodore Oswandi

November 5, 2017

## 1 Lecture 1 & 2

### Improvements

- Semiconductor technology, clock rate/frequency, and computer architecture improved
- Performance measured using standardised benchmarks

### Computer Architecture

**ISA:** Boundary between hardware and software

**Oragnisation:** high level computer design aspects

**Hardware:** detailed logic and circuit design

**Note:** Want to separate your instruction set from implementation

### Memory Organisation

- Memory is indexed 1D array
- Memory Access Time: time to read data to/from memory
- Memory Speed != Processor speed.
- Fast memory is very expensive. Heirarchy used for functionality and cost.

### Processor Registers

- Smallest and fastest memory for CPU
- about 32-64 of them. Each 32/64bits in size.
- Nanosecond access time

### Cache Memory

- Slower than register, but larger (8-256k)
- Few nanoseconds access time
- Levels (L1, L2, L3) used in multiprocessor systems.

### Main Memory

- Slower than cache. But really big.
- Tens of nanoseconds lookup time.

### Instruction Set Architecture (ISA)

**ISA** is interface between hardware and low level software. (80x86, MIPS, ARMs)

### Using Fixed ISAs

Uses old instruction set (1970s), also used with extensions to enable newer technologies such as internet, etc...

## Advantages

- Can have different implementation of same architecture
- AMD/Intel both have same ISA but different implementation.

## Disadvantages

- power consumption is higher than things like iPad which use different ISA and consume a lot less power
- Also prevent some new innovation due to wide adoption.

## ISA Design

### Need to define:

- Instruction Format and Encoding
- Data types and their sizes
- Location of operands and where to store results

**Operands and Opcodes** To carry out these calculations, an **opcode** must be defined to define these calculations. Upon these opcodes, zero to three **operands** are used for data inputs and result outputs.

## Architecture Types

### Stack Base Architecture

- Top of stack will contain result of operation.
- If receive ADD then processor knows next 2 inputs contain 2 numbers that need to be added.
- PUSH add something to top of stack.
- POP use value in top of stack.
- JVM designed to use Stack based architecture.
- ADD function has no operators. Operates on last 2 loaded values.

### Accumulator Based Architecture.

- Using inputs from memory.
- Not used anymore today. Used in 1970s
- ADD function takes one operator, *mem<sub>a</sub>address* which contains the value to add to above loaded value.

### Register Memory Architecture

- Currently used today as x86
- Uses register for input as well as access values from memory.
- ADD function contains 3 operator.
  1. **Rd** Destination Register
  2. **Rs** Source Register
  3. **mem.address** Address of value to add from memory

### Register-Register Architecture

- Operands from register.
- LOAD and STORE only way to access memory.
- Need to specify destination register for output.
- ADD function has 3 operators.
  1. **Rd** Destination Register
  2. **Rs** Source Register
  3. **Rt** Register containing other value you want to add

## Examples

Example is  $A(1000) + B(2000) = C(3000)$  in the 4 types of architectures

### Stack Based Architecture

```
PUSH 1000
PUSH 2000
ADD
POP 3000
```

### Accumulator Based

```
LOAD 1000
ADD 2000
STORE 3000
```

### Register Memory

```
LOAD R2, 1000
ADD R1, R2, 2000
STORE R1, 3000
```

### Register Register

```
LOAD R2, 1000
LOAD R3, 2000
ADD R1, R2, R3
STORE R1, 3000
```

## ISA Classes

**Classification** generally based on: Instruction word size, number of different instructions, and number of clock cycles to complete a given instruction.

### Classes

**RISC** (Reduced Instruction Set Computers) all instruction words same size. Simpler decoding hardware.

**MIPS** is an example processor that uses this type of ISA.

**CISC** (Complex Instruction Set Computers) instruction word sizes may vary. Code footprint may be smaller than RISC due to condensing multiple RISC instructions into one CISC.

**Intel x86** is an example of processors based off this.

**EPIC** (Explicitly Parallel Instruction Computers) have parallel operations in their instruction set. The compiler is very important.

**Intel Itanium** uses this kind of ISA.

### Abstractions

**Abstractions** remove unnecessary details and hide complexity so that it is easier to understand.

### Instruction Processing in CPU

1. **Fetching** access memory, get next instruction.
2. **Decoding** Interprets the instruction. (Operation and data required from memory/registers)
3. **Execution** Perform operation. Uses processor and writes result to register/memory.

## Questions to ask when designing ISA

1. What type of ISA should be used?
2. What operations are needed?
3. How data (operands) are provided in instructions?
4. Instruction and Data word sizes?

## Extras

### Types of operations

- **Arithmetic** Addition, Subtraction, Multiplication, Division
- **Logical** AND, OR, Lshift, Rshift
- **Memory Access** LOAD, STORE
- **Control Transfer** Conditional/Unconditional Branches
- **Special Purpose** will talk later

**Notes: Shifting** You have to be careful when shifting as if you're dealing with signed integers then you may be messing with the sign bit when trying to multiply/divide

## 2 Lecture 3 & 4

### ALU Operations

- Add immediate uses value, not pointer to register
- No Subimmediate as if Addimmediate allows negatives.
- Destination register generally before source register/s
- Register 0 is **static final** containing all 0s, and cannot be written to

### Endian-ness

1. **Little Endian** Least significant bit at top of memory addresses  
LSB at addr, MSB at addr+3
2. **Big Endian** Most significant bit at the top of memory addresses  
LSB at addr+3, MSB at addr

**Sizes of things** in relation to memory size

Word=4bytes

halfword=2bytes

1byte=byte

This course will use MIPS simulator on PC called SPIM

**Memory addressing in MIPS** machine is  $C(r_x)$

Where C is constant which may be used to reserve part of memory.

And  $R_x$  is the contents of a given register

- lw, sw = Load/Store word
- lh, sh = Load/Store half-word
- lb, sb = Load/Store byte
- **NOTE:** lbu = Load byte unsigned. No need to sbu as it will only store the relevant least significant byte in register

## Class exercise

```
addi $10, $0, 0x3000
ori $12, $0, 0x8015
sw $12, 512($10)
```

\$10 = 0000 3000

\$12 = 0000 8015

sw register to put \$10 is  $512_{10} + 8015_{16} = 0x00003200$

Therefore Big Endian: 00 00 80 15

And Little Endian: 15 80 00 00

## Instruction Encoding Cont.

- **Note:** For efficient instruction encoding, we classify different instructions and formats for faster decode.
- If 32 registers then need 5bits to encode pointer to relevant register.
- Opcode needs 6bits to be represented. Can encode 64 opcodes.
- Also need some bits to represent immediate values and offsets.
- Program Counter (PC) used to signify where execution has got up to and therefore next instruction to execute.

### R-Type Format

- 6bit OpCode                      Operation Instruction
- 5bit Register                   First register operand
- 5bit Register                   Second register operand
- 5bit Register                   Third register operand
- 5bit Shift                      Amount for shift instructions
- 6bit Function Code            Operation variant.

### I-Type Format

- 6bit OpCode
- 5bit Destination Register
- 5bit Source Register
- 16bit Offset/Immediate Value (Depending on instruction)

### Jumping Memory Addresses.

**Changing sequence of execution:** is done through use of **branch** and **jump** instructions. This is done to let you have if/else and loops.

**Jump** Jumps to location in memory (unconditionally) to get next instruction. Like a GOTO. Uses J-Type Format

**Branch** lets you conditionally go to another point in memory, only if the condition is met. Uses I-Type Format

**Jumping** [6bit OpCode][26bit TargetAddress]

Target address must be 32 bit, so to get this 26 bit value to 32 bit you shift the 26 bit number left 2 times, then add the 4 MSB of PC to front of value. This results in final expected 32bit address needed for the jump.

**Branching** [6bit OpCode][5bit Reg1][5bit Reg2][16bit Label]

- Once again, the target address must be 32bit, so have to calculate it using encoded information in instruction.
- Conditional Branching is **PC-relative** meaning that the PC provides current address and the Label provides an offset.

- **BEQ** branch equal, taking 2 inputs and a label if true
- **BNE** branch not equal, taking 2 inputs and a label if true

bne and beq only have 16 extra bits. Therefore need to get to 32bit target address. Need to use PC again. Use 14 bits after shifting the 16bits in bne/beq to left by 2.

**Another example** target address =  $0100\ 0400 + 4 + 400 = 0100\ 0804$

+ 4 from PC+4

+400 from  $0x100 * 4$  [left shift 2]

**NOTE** Have to also consider is little/big Endian by looking at the machine code of original instruction. The 16bit offset at the end will let you know.

```
addi    $11, $0, 100      // Initialise counter as 100
lw      $8, 0($10)        // Load word from R10 into R8
sra     $9, $8, 3         // Shift right arithmetic on that and save in R9
sw      $9, 0($10)        // Store contents of R9 back into R10
addu    $10, $10, 4       // find the address of next element
addi    $11, $11, -1      // Decrement loop counter
bne     $11, $0, L1
```

### 3 Lecture 5 & 6

#### Example of while loop

Example: Translate the below code to MIPS

```
while(w[i] == x)
    i = i + j
```

Given that  $i = \$3$ ,  $j = \$4$ ,  $x = \$5$ ,  $w(\text{int array}) = ]\$6$

Answer:

```
loop: #perform loop test
    sll    $10, $3, 2      #Shift left 2 == x4
    add    $10, $6, $10    #Get address of w[i]
    lw     $11, 0($10)     # $11 = w[i]
    #perform w[i] == x
    bne    $11, $5, exit   #Exit when w[i] != x
    # i = i + j
    add    $3, $3, $4
    j      loop
```

exit: #out of the for loop

#### Comparison Instructions

- Used for things like  $(x < y)$
- Will set destination register different value depending on result of comparison
- Will then use bne/beq to check if less than or greater than.

Examples:

- **slt** Rd, Rs, Rt = signed less than
- **sltu** Rd, Rs, Rt = unsigned less than
- **slti** Rd, Rs, Immed = signed immediate less than
- **sltiu** Rd, Rs, Immed = unsigned immediate less than

## Load Upper Immediate

No implicit instruction to copy from one register to another. But can use contents of \$0 with the ADD or OR function to achieve the same effect. Use a pseudo instruction (scriptlike) called **Load Upper Immediate (lui)**

Used when loading values to upper section of a register. Operations like **ori** and **andi** do this for the bottom half of a register

## Integer Multiplication and Division

Since multiplying two 32bit numbers generates a 64bit number, need to store it somewhere. LO and HI registers are special registers that hold the result of the multiplication or division.

- **mflo \$REG** move from LO
- **mfhi \$REG** move from HI

## Subroutines

Procedure that gets called multiple times. Used to create modular program.

### Issues:

- Need to call it.
- Need to give it arguments sometimes.
- Need it to return values sometimes.

### Unconditional Jumps:

- **j address** Jump to the 26bit address doing the shifting and 4MSB of PC attached.
- **jal address** (Jump and link) lets you jump to subroutine. Also uses \$31 used to store return address (PC+4 saved to the register)
- **jr register** Uses the value of register number as target address. Return instruction.

### Conditional Branches:

- **beq r1, r2, offset**
- **bne r1, r2, offset**  
offset being 16 bit number (then shifted 2bits left and added to PC+4)

### Relevant Registers:

- **\$a0, \$a1, \$a2, \$a3** used for integer arguments
- **\$v0** used as return register
- **\$sp** is stack pointer register
- **\$fp** is frame pointer register

Stack frames size should be multiple of 16.

### Memory Layout:

- Stack Segment (goes down) 0x7ffffff
- Data Segment (goes up)
- Text Segment 0x40000000
- Reserved Memory
  - Used for special OS tasks

If have nested functions then need to save and restore value of \$31. Save this in the current stack so the value can be extracted and restored.

Send 5th and onwards function input arguments using the stack space.

## 4 Lectures 7 & 8

### Stack Frame Usage Summary

- Procedure that doesn't call another one is a **leaf**
- Leaf procedure doesn't save \$ra
- Store values in registers used by calling procedure.
- Most of time local variables use registers, but can be stack frame sometimes.

### MIPS assembly programming

- Main CPU with ALU and special registers for mult/div
- Coprocessor for floating point stuff
- Coprocessor for traps and memory

### Assembly directives

Identifiers are used to provide commands to assembler, prefaced with a dot (.)

### Rest of the lecture about SPIM



## 5 Lecture 9 & 10

### Jump Tables

More efficient way to do switch-case statements than to do a bunch of if-else statements.

```
switch(k){
    case 0:      f = i + j;      break;
    case 1:      f = g + h;      break;
    case 2:      f = g - h;      break;
    case 3:      f = i - j;      break;
}
```

```
JumpTable .word L0, L1, L2, L3 #Creating jump table

jr k #Jump to relevant k value from jump table

L0: add f, i, j #k = 0, f = i + j
    j exit

L1: add f, g, h #k = 1, f = g + h
    j exit

L2: add f, g, -h #k = 2, f = g - h
    j exit

L3: add f, i, -j #k = 3, f = i - j. No break;

exit:
```

### Floating Point Representation

#### Single Precision

- bit 31 Sign bit
- bit 30-23 Exponent
- bit 22-00 Mantissa

#### Double Precision

- bit 63 Sign bit
- bit 62-52 Exponent
- bit 51-00 Mantissa

k

#### Extras

- **zero** Exponent and Mantissa are all 0s
- **Infinity** Exponent 255, Mantissa 0s. Sign determines if positive or negative.
- **NaN** Exponent 255, non-zero mantissa

### MIPS

MIPS has a floating point co-processor with special registers (pairs for doubles) to represent floats.

Co-processor itself has 32 32bit registers. \$f0 \$f31.

## Registers

- 00 - 02     Results of a function
- 04 - 10     Temp registers
- 12 - 14     Pass first 2 arguments
- 16 - 18     Temp registers
- 20 - 30     Caller saved registers

**lwc1** and **swc1** are used to load/store values to the coprocessor 1 (floats)

## Operations

- **Single**     add.s, sub.s, mul.s, div.s
- **Double**     add.d, sub.d, mul.d, div.d

## Compare Instructions

You use \$31 to see comparison instructions between 2 floats.

Function **c.X.s** or **c.X.d** is used to compare the values in the registers and return value saved in \$31. X can represent:

- eq = equal
- neq = not equal
- lt = less than
- le = less than or equal
- gt = greater than
- ge = greater than or equal

## Converting

Can convert from any Integer/Word, Single Float, Double Float to any other using **cvt.X.Y** where y is [w = word, s = single, d = double]

## Moving to/from coprocessor

**mtcZ/mfcZ** is move to/from coprocessor. Z value = 1 represents co-processor for floating point.

**mtc1 \$12, \$15** moves value from Register12 to Register15 in coprocessor1.

This function just moves the bits to/from the registers and doesn't do the conversion.

## 6 Lecture 11 & 12

Digital Circuit are

- **Combinational** output only dependent on current input value
- **Sequential** output dependent on current input and state

## Propagation Delay

- It exists
- Charging and discharging of load capacitance means nonzero  $D_{prop}$
- Need to take this into account or may get glitches

## Sequential Circuits

- Have storage element to keep state
- Flipflops have delays, and are edge triggered.
- $D_{setup}$  minimum time for input before change in CLK
- $D_{hold}$  time output needs to be stable after CLK edge.

## Implementing

Area between ISA and Hardware. Organisation of aspects of computer design.

## CPU implementation

- **Control Unit** Generate signals to direct datapath
- **Datapath** perform CPU operations
- **Control Signals** from CU to DP
- **Status Signals** from DP to CU

## Datapath components

### Arithmetic Logic Unit - ALU

- Arithmetic + Local operations on register value.
- Has list of operations it can do
- Extra output for **Zero** or **Overflow** result
- 3bit code to specify what operation you want to do.

### Register File

- Provide operands for operations
- Has bunch of potential inputs.
- Get values of input registers, used to represent what registers need to be written/read to.

### Program Counter - PC

- Indicate address of next instruction
- Signals for reading/writing

### Memory Interfacing

- Used to access memory for instructions

### Sign Extension

- Change 16bit signed number to 32bit signed number

## ISA Subset Implementation

### Single Cycle Implementation

- Assume single clock cycle for each instruction.
- Longer clock cycle to accommodate this.
- **Duty Cycle** ratio of system being active to being inactive.

Fetch → Decode → Execute (and loop back around)

- **Fetch** Read instruction word from memory, increment PC.
- **Decode** Use fields from instruction to get values necessary and ignore un-needed registers

## Datapath for Section

### Fetch

- PC - to get address of next instruction
- Register File - put instruction address
- ALU - for incrementing PC

### R-Type Instruction

- Register File - get value from register
- ALU - Perform operation on value of register
- Register File - specify output register.

### Load/Store

- ALU - result of ALU to Memory Interfacing
- Register File - Destination register.

### Immediate values and ALU

Can get value straight from register (not through register file). Ensure it goes through Sign Extender first, to convert 16bit input to 32bit input.

### Implementing BEQ Example

- Use ALU and subtract
- Check if **zero** bit is set in ALU (will be set if values are the same)
- Need to set new address if **zero** value is set
- ALU ADD - used for calculating new memory address
- ALU SUB - used to check for equality
- Control Signal for ALU
  - lw 00
  - sw 00
  - beq 01

## 7 Lecture 13 & 14

### 7.1 Single Cycle Datapath Implementation cont.

(11) not correct as inputs for read/write registers for src/dest of some operations not guaranteed. Some have OP rs rd and some have OP rd rs

(12) Sequential circuits need to have clock inputs, not shown in the diagram.

(14) have another section used for generating and allocating the various control signals - easiest way to set these control signals is to just generate a table to map 1/0 values to the various outputs needed for each opcode

Control Unit Input = 6bit Opcode Output = various control signals for sections of datapath

Outputs: RegDst ALUSrc MemtoReg RegWrite MemRead MemWrite branch ALUop1 ALUop0

This control section is combinational since it just takes the opcode and asserts values on the output control signals without need of clock

Implementing Jump Instruction Jump instructions doesn't use registers Since it changes PC unconditionally, only datapath section that relates to PC needs to be extended. Need to create new control signal to gate either jump or branch instruction

(20) can't execute archive instructions on given datapath

Implement BNE Instruction Can still use the sub with ALU and zero bit Also can use the register loading from BEQ Need to create new control signal to distinguish BNE and BEQ Same as BEQ, but you negate the zero value Then OR the result of both the BEQ and BNE AND gates

Implementing addi Can modify ALU control Or can also choose not to modify ALU at all and change how the registers are loaded, however this is harder to do

Example, list values of control signals for: ori \$t0, \$t1, 0x101A

RegDst 0 Not reading from third register Jump 0 Not jump Branch\_ne 0 Not Branch Branch\_eq 0 Not Branch Memread 0 Don't read memory MemtoReg 0 Using value of address ALUop 11 Use value for immediate value (no table given) Memwrite 0 ALUsrc 1 Regwrite 1

Improving Performance In reality these components have delays associated with them Memory Unit = 2ns ALU & Adder = 2ns Register File (R/W) = 1ns Others = assumed to be negligible

6ns for archive instruction (R-type) 2ns for getting from memory 1ns for reading from register 2ns for ALU adding 1ns for writing to register

LW instruction DIDN'T GET IT

Example 2 Timing Since single cycle implementation then get the highest delay as the total delay per cycle. No need to add all delays

Performance Single Cycle Implementation Comparing:

1. single-cycle implementation with pessimistic clock timing due to longest instruction
2. Single cycle implementation where each clock cycle only runs for as long as it needs to. Not possible in real life, but just for conceptual purposes

IC = Instruction Count

Multicycle implementation

- Uses multiple clock cycles of clock that runs faster than doing everything on one long clock cycle
- Different clock cycles used for each of the different phases of execution of an instruction
- control units have to become more complex
- however datapath may become more simpler
- has shared memory unit and ALU
- don't need to separate instruction and memory fetch since can be done in multiple cycles like in single cycle
- Single memory interface enough for this implementation

Goal: to separate it to allow for multiple cycles for a single instruction

## 8 Lecture 15 & 16

Multicycle implementation of MIPS ISA

- Multiple clock cycle for single full instruction
- Some functional units not used in some clock cycles, can share between instructions
- Need to store state between these cycles. So will need some temporary registers
- Simpler data path, but more complex control unit as a result

High level view ALUout are temporary register to keep value of ALU computed in a clock cycle IR write, Memory Data, RegisterReadA, RegisterReadB, and ALUout are temporary registers

Problem - to figure out how many clock cycles to allocate to each of the different types of instructions.

3 different sections

1. Get from memory section
2. Register Section
3. ALU section

Some 4 clock cycle instructions

1. Get from memory
2. Load registers
3. Calculate result
4. Store result back to memory

R type

- Instruction Fetch
- Decode/Register Fetch
- Execution
- Memory Access, finish R-type, write ALUout to register values

#### Memory Reference

- Instruction Fetch
- Decode/Register Fetch
- Execution
- Memory Access, either LOAD or STORE
  - If load, then need to finish memory read

#### Branch Instructions

- Instruction Fetch
- Decode/Register Fetch
- Execution, and update PC to ALUout

#### Jump Instructions

- Instruction Fetch
- Decode/Register Fetch
- Execution, and update PC to register values

IorD = used to signify when you want to use PC as location of memory address when set to 0

For a given set of signals that need to be asserted, make sure that other signals won't affect the intended result of the asserted signals

#### TODO Explanation of Control Unit Signals

- Outputs Control
  1. PCWriteCond
  2. PCWrite
  3. IorD
  4. MemRead
  5. MemWrite
  6. MemtoReg
  7. IRWrite
  8. PCSrc
  9. ALUOp
  10. ALUSrcB
  11. ALUSrcA
  12. RegWrite
  13. RegDst

Also need to calculate average cycles per instruction

Loading takes highest number of cycles, therefore reducing this should make overall performance faster if doing multiple cycle datapath

Example with machines with different characteristics Look at overall FSM and combine states that are in description of machines, as they are performed on the same clock cycle Then look at critical path of number of cycles and determine the machine with best overall performance

Performance depends on both clock period and average instructions per instruction so need to find best balance between the two.

## 9 Lecture 17 & 18

Went over question on slide 4.

Exceptions and Interrupts MIPS uses interrupt for expected event, and exception for unexpected event When happens, control transferred to Exception Handler/Interrupt Service Routine that processes event that occurred. Then return back to program to resume.

Causes External Event Timer, Keyboard/Mouse input - All asynchronous things that we can't predict all the time.

Internal Events Traps, happens inside the code for a situation like divide by 0, or overflow Software Interrupts, some instructions that create an interrupt. These things are synchronous, happen at a certain point in the code.

Fixed by handler Simulated if do not have necessary hardware to do something (Something old did not have float\_coprocessor, so would emulate it) Set result to null

MIPS convention for exceptions SOMETABLE

Identifying what caused an exception EPC CAUSE BADVADDR STATUS

MIPS uses single address to store the exception handling code due to memory constraints. Is at 0x8000 0180, and in kernel somewhere.

If using spim, use eret

Classic user mode and kernel mode (3 levels of privilege for kernel mode)

Status Register register 12, coprocessor 0 bit0 Interrupt enable (1 = enabled) bit1 Exception level (1 = exception has occurred) bits8-15 Interrupt mask (6 hardware interrupt, 2 software interrupt)

Register 13, co0 bit 2-6 Encoding cause for interrupt/excp bit 8-15 Pending interrupt mask

Some more examples Overflow ADD, ADDI, SUB will cause exception if overflow ADDU will not, will just get wrong result if that is the case

Address Error

System Call

MIPS EXAMPLE

Adding Exception Handling to multicycle datapath Only consider undefined instruction arithmetic overflow Special registers EPC and Cause used The value of cause will determine which kind of exception was thrown

## 10 Lecture 17 & 18

Pipelining Basically doing stuff concurrently if not dependent on each other If washing machine done, put in drier so next person can use washing machine

Execution Latency; DEFINITION Throughput, total work done in given time

Multi-cycle to Pipeline Switching them is difficult as some resources are required in every clock cycles for some kinds of instructions All instructions should have same execution latency to make pipelining chunks of work easier.

If have  $M$  pipelined stages and  $N$  instructions, then need  $M + N - 1$

Therefore cycles per instruction approaches 1  $CPI = \frac{M+N-1}{N}$  Assume ideal case when doing calculation ( $CPI = 1$ )

Pipeline Datapath Need to have pipeline registers to keep information between the clock cycles for the specific instructions

Register if read, RHS is shaded If written then LHS is shaded

In single cycle, use clock cycles to assert control signals In multi-cycle then use the FSM to assert clock signals

Control signals also pipelined for the following stage, so registers also contain state for next stage signals Put it between Instruction Fetch and Instruction decode, to allocate for Execution and Mem/WB

Pipelining Hazards Not everything in the world is ideal Due to nature of pipelining, if doing operations that depend on each other. If not correctly spaced apart, may read old values and operate on them instead of new value

These are called hazards They reduce performance if not handled correctly May need to stall pipeline to guarantee the values are read and written correctly and as expected.

Structural Hazards Resource conflict Simultaneous access of same resource

Data Hazards Instruction dependent on result of another, need to wait for that one to be done first

Control Hazards Instructions exist in pipeline that need PC to be changed.

Detect and Resolution **Forwarding Mechanism**- Pipe data straight from result of ALU into register input. Forwarding Check if rt/rs of given instruction is same as rd of previous instruction. Also need to check if not same as 2 previous destination. To prevent the value being piped 2 instructions ahead when 3 consecutive instructions depend on value of single register.

Stalling If next instruction dependent on value of previous instruction that hasn't been generated yet. Need to stall for  $n$  clock cycles to let it generate the value expected. Done by adding delay cycles to stall overall

execution of instructions. Check if memread (loading word), and then if previous pipeline register is expecting same value, then need to delay for a cycle.

If need to do this, make sure that PC isn't changed as it may skip the executing of the instruction you want to delay. Therefore make operation have no effect by setting all control signals to 0 in EX, MEM, and WB stages

## 11 Extra Morteza

### 11.1 Single Cycle Implementation Control Signals

1. **RegDst**

Denotes if ALU will be used

- **ZERO** if ALU not used. For things like `mult`, `div`, `jr`, `mfc0`
- **ONE** if ALU used. For thing slike `add`, `sub`, `or`, `and`, `lt`

2. **Branch**

Denotes if branching will happen. Feeds into an AND gate.

- If **ONE** then will get 16 bit offset, shift left by 2, and add 4MSB of PC+4
- If **ZERO** then continue using PC+4 as next instruction

3. **MemRead**

Only used for LoadWord operations, however can be used for some R-type

4. **MemtoReg**

Only used for LoadWord operations

5. **ALUop**

Denotes what operation the ALU will do

- 000 AND
- 001 OR
- 010 ADD
- 011 SUB
- 111 LessThan

6. **MemWrite**

Only used for StoreWords operation

7. **ALUsrc**

Used for Load and Store operations. Denotes if ALU uses register or 16bit immediate value from instruction

8. **RegWrite**

R-type and load word use. Potentially used for write data for Register File.

### 11.2 Multiple Cycle Implementation Control Signals

1. **RegDst**

Denotes if ALU will be used

- **ZERO** if ALU not used. For things like `mult`, `div`, `jr`, `mfc0`
- **ONE** if ALU used. For thing slike `add`, `sub`, `or`, `and`, `lt`

2. **Branch**

Denotes if branching will happen. Feeds into an AND gate.

- If **ONE** then will get 16 bit offset, shift left by 2, and add 4MSB of PC+4
- If **ZERO** then continue using PC+4 as next instruction

3. **MemRead**

Only used for LoadWord operations, however can be used for some R-type

4. **MemtoReg**

Only used for LoadWord operations

5. **ALUop**

Denotes what operation the ALU will do



- 000 AND
- 001 OR
- 010 ADD
- 011 SUB
- 111 LessThan

#### 6. **MemWrite**

Only used for StoreWords operation

#### 7. **ALUsrc**

Used for Load and Store operations. Denotes if ALU uses register or 16bit immediate value from instruction

#### 8. **RegWrite**

R-type and load word use. Potentially used for write data for Register File.

## 12 Caches 1 - Generic/Introduction

### CPUs need fast memory

CPUs have frequencies of 3GHz, and **random** memory accessing takes approximately 1.2ns (and occurs about 25% of the time). The fact that random is bolded is due to the fact that sequential accessing is much much faster than random access.

Problem is that **RAM** has 40ns+ access time. Therefore faster cache exist and if had to only rely on that, the CPU would be doing nothing and waiting a lot of the time.

However, **Static RAM** (SRAM) is much faster at 0.5-2ns access times, but is very expensive. Also need to be **small to be fast**. The larger the total size, the larger it is and the longer it takes for you to access something at a random location.

Classic memory heirarchy again of [Cache, RAM, Disk]

### Cache Principle

Keep memory consistent and fast.

- Higher levels contain copies of the lower level caches
- If try get memory location, start at lower level and go up.

### Cache

- Transparent to program
- Hold subset of memory
- Commonly accessed data/instr in cache
- Different levels in itself. (Usually 3, but can be 2)

### Locality of Reference

You don't want to just store random things in the cache. Want to actually have a plan of what you want to put there.

;;SOMETHING;;

**Spatial Locality** If you access an item at a given space, you are likely to access another item next to it, or closeby in memory. Therefore loading more than one item at a time may produce speedup.

**Temporal Locality** If you access an item, you are likely to do it again within a certain timeframe. Can be things like looping through a section of memory.

## Cache Operation

Ask cache first. If there then is a HIT and return it. Otherwise ask memory (or level of cache).

**Hit rate** Times you hit, per number of memory accesses. The higher the rate the better it is. Different memory access times per level. Measured by timing when you first try get the resource, up until you get it.

**Miss rate** Times you don't get a hit for every

**Miss penalty** Access time after miss.

## Cache benefit

;;SOMETHING;;

The higher you go the faster the memory is perceived.

## Example

L1 = 4 clock 5%miss L2 = 11 clock 2.5% gloal miss L3 = 21 clock 1.5% global miss RAM = 120clock

Therefore

$(4 * 0.95) + (11 * 0.025) + (21 * 0.01) + (120 * 0.015) = 6.085$  cycles of perceived memory access time

## 13 Caches 2 - Organisation

### Cache Design

#### Objectives

- High hit rate
  - Low access time
- Can't do much about access time in this course, as it is reliant on underlying implementation of hardware)

#### Points to consider

- Size of cache
- Block/Line size (doesn't handle bytes or words, but larger entities called blocks and lines)
- Data location mapping
- Replacement (cache is small, gotta remove to add)
- Write policy (what happens if processor wants to write to memory location, and keep consistency)
- Levels of cache, exclusive/inclusive

## Direct Mapping

Directly using the bits of the address to where it is in memory.

Look at lower 3 bits and see where it is mapped in memory. But repeated many times in main memory so there is competition for all things with same suffix to have that spot in cache.

If it goes into cache, it goes into into the same position in cache.

;;STUFF;;

- **Valid bit.** Associated with cache entry. Represents if that position in cache has value loaded in it
- **Index.** Lower part of address, specifies where it will go if cached
- **Tag.** Upper part of address, specifies memory location

Therefore [Tag][Index] represents full memory address of relevant cached resource.

Address to cache entry [Tag][index][Offset]

;;YAY;;

Cache Metrics Machine dependent  $q$  = word size  $p$  = address size

Byte Addressed Memory  $m$  = byte offset bits (dependent on word size)  $b$  = block bits  $k$  = index bits  
 64byte line example 18bit tag 8bit index 4bit block offset 2bit byte offset  
 Direct Mapping Analysis Simple :) both conceptually and hardware  
 Low hardware cost :) single comparator  
 High potential for conflict :(  $\text{X D}$   
 Fully Associative Memory location go to any line of cache Line index no relation to address Larger TAG no K  
 no line index Comparator for every line Lots of comparators Memory location could be anywhere Sequential  
 comparison slow  
 Replacement policy needed.  
 MORE YAY  
 Set Associative Hybrid Doesn't have a single direct mapped section, but multiple of them.  
 Depends how wide or tall you want to go with the design.  
 N - way associative Performance similar to fully associative, as  $n$  increases. Much less hardware than having  
 it fully associative  
 Performance Comparison Using SPEC2000 15Kbyte cache, 64bytes per line  $p$  = 32bit word, 16 per line  $k$  =  
 $8 = 256$  lines  
 Associativity 1 10.3% 2 8.5% 4 8.3% 8 8.1%  
 If change size of cache, then significant reduction in miss rate up to approx 64KB cache size. Negligible after  
 then. Most significant from 1KB to 2KB  
 STOPPED LISTENING @ slide 18

## 14 Cache 3

Line replacement Replacement Policy Optimal Policy If you used it recently, you might want to use it again  
 remove the item that is oldest (has oldest last access time)  
 Random Doesn't trigger worst case scenario Bad for average performance  
 FIFO Replace longest item that's been in cache Round robin over sets Needs  $k$  pointers. Doesn't mean that  
 something isn't used if in cache for a long time.  
 LRU (Least Recently Used) Replace line that was accessed long ago Normally not good enough Can be  
 difficult to implement  
 Can be done by "flipping" the bits so that next cache position changes every time.  
 4 way pretty good but an approximation 8 way is a single more level 16 way gets to the point where it isn't  
 worth it anymore to continue getting deeper as the cost of storing all these bits increases. 8 line pairs Choose  
 random pair and choose single LRU  
 Writing to memory Due to nature of multilevel cache, if you write to something you have to ensure that it  
 propagates throughout all levels of cache. Don't want file written to to be replaced and updates lost.  
 Write through Every write goes through memory. Write updates go through hierarchy  
 If cache hit modify cache line write to memory  
 If cache miss Write allocate Item put into cache Write over it Then update propagates through levels to  
 memory  
 Write no allocate CPU ignores cache and updates memory directly.  
 Uses write buffer so that processor doesn't stall. Writes done in parallel with other processes, and has low  
 priority compared to reads.  
 Write Back Write only goes to cache Modification indicated with **dirty bit**  
 If cache hit modify cache line set bit  
 If cache miss Evict conflicting cache line Write to memory if dirty bit set Fetch cache line modify cache line  
 set dirty bit Can also use a write buffer for cache misses  
 Comparison Write Through Efficient Use of bus interface queuing Priority to reads Lots of traffic Cannot use  
 burst mode Only one word written, not entire line  
 Write Back Better for typical program. When generated writes  $\downarrow$  bandwidth More difficult to implement.

## 15 Cache 4

### 15.1 Optimising cache performance

#### 15.1.1 Performance determined by

1. Miss rate
2. Hit rate
3. Miss penalty

#### 15.1.2 Causes of cache misses

- Compulsory misses
  - First time cache line accessed
  - Reduced by larger blocks
- Capacity Misses
  - Working set  $\geq$  Cache size
  - Reduced by larger cache
- Conflict misses
  - Data compete for same cacheline
  - Reduced by higher associativity (& Better replacement policy)
  - Reduced by larger cache
  - Reduced by **smaller** block
    - \* The larger the cache line, the more misses you get (more conflicts)

#### 15.1.3 Changing Cache Aspects

1. Larger cache size
  - :) Reduce miss rate
  - :( Increases hit time
2. Larger cache blocks
  - :) reduces miss rate (compulsory)
  - :( Increases miss rate (conflict)
  - :( Increases miss penalty (mitigated with higher bandwidth)
3. Higher associativity
  - :) Reduces miss rate
  - :( Increases hit time (possibly)
4. Write through/back
  - No clear influence on cache performance
  - More influence on system bus traffic

### 15.2 Multilevel Cache

Solution to memory not being able to be fast and large at the same time

1. L1 cache
  - Small
  - Split cache (DATA and INSTRUCTION)
  - Hit time very important
2. L2 & L3 Cache
  - Larger (lines may also be larger)
  - Unified (not split to I and D)
  - Reduces miss penalty for L1
  - Faster than main, slower than L1

- **Inclusive**
  - Populated from main memory up to L1
  - Simple management
  - Coherent easily (if in L2, also in L3)
  - Total size is size of L3
  - Used by Intel
- **Exclusive**
  - If exist in L2, doesn't exist in other cache levels
  - Complex management
  - Difficult to keep coherent
  - Works like victim cache
    - \* Line removed from L2 goes to L3
  - Total size is sum of size of levels of cache (higher than inclusive)
  - Used by AMD Bulldozer/Piledriver

## 15.3 Programming for cache performance

### Objectives

- Minimise miss rate
- Hide latency
- **Programmers** can't change hardware, access times or penalties
- **Prefetching** hides latency by loading cache line before accessing it. HW = CPU prefetching, SW = machine instruction
- **Bypassing** reduces miss rate by not caching writes. SW = machine instructions

#### 15.3.1 Mimising miss rate

- Programmer needs to know size of cache, line size and organisation.
- Try to use as much of lines as possible, physically and frequency before overwriting it.
- Maximise spatial and temporal locality

#### 15.3.2 Programming approaches: adjust...

- Size of working set
  - Divide problems to smaller chunk to fit in cache level
  - Usually not use full cache size, due to competition with instructions and OS calls.
- Access patterns
  - Algorithm have freedom
  - Choose best for performance (row major and column major)
- Stride
  - Longer the stride the worse it is.

## 16 Virtual Memory 1

## 17 Virtual Memory 2

### 17.1 TLB - Translation Look-aside Buffer

#### 17.1.1 Performance problem

Have to translate every address before it is accessed Therefore 3 memory accesses required: page table, data and address + translation More chance for cache miss.

TLB in memory management unit of every modern CPU. Address translation done before going to L1 cache.

### 17.1.2 TLB metrics

- Metrics - Typical
  - Size: 16-512 entries
  - Block size: 1-2 page table entries
  - Hit time: 0.5-1 cycle
  - Miss penalty: 10-100 cycles
  - Miss rate: 0.01% - 1%
  - Levels: 1-2
  - Split: data and instruction split
- Associativity
  - TLBs are small
  - Need to be fast
  - Fully associative OK for small caches
  - LRU too expensive
  - Rarely low associativity
  - Miss expensive, aim for low miss rate

(6) the addresses of physical page number + page offset are the same as address tag + cache index + block offset + byte offset

amd has level 0 caches as their fastest level

### 17.1.3 TLB misses

- Not the same as page fault!
- Page hit
  - Can be handled by hardware or software
- Page fault - When quick translation doesn't find anything and therefore not in memory and must get from disk
  - Handled by software
- Software is invoked via exceptions

## 18 Parallelisation 1

### 18.1 Principles

Motivation = Types =

#### 18.1.1 Forms of CPU parallelism

- Instruction level
  - Pipeline
  - Multiple issue
    - \* Large instruction word (static)
    - \* Superscalar (dynamic)
  - SIMD
- Thread level parallelism
  - Simultaneous multithreading
  - Modules - AMD bulldozer
  - Multicore

## 18.2 Thread level parallelism

Have to rewrite the programs Frequency of processors aren't improving much anymore, therefore need to utilise more cores.

SMT - Simultaneous multithreading (Hyperthreading) Still single pipeline, but 2 registers for each pipeline stage Can process 2 threads Superscalar (usually) Situations where HT is good, and others where it is bad/neutral

Modules - AMD bulldozer halfway between SMT and multicore Almost everything doubled, except for IF, ID and commit unit

Multicore Multiple independent processor on the same die Share caches (partially) Combined with SMT/HT

## 19 Parallelism 2

### 19.1 Memory and parallel systems

Location and access Location Centralised in PC Distributed in med-large parallel systems

Access Shared memory with multicore, pc, servers Message passing in clusters and large parallel systems

Focus on shared memory and multicore processors u

### 19.2 Shared memory multiprocessors