COMPSYS304 Notes 2017

Theodore Oswandi

August 18, 2017

1 Lecture 1 & 2

1.1 Improvements

- Semiconductor technology and computer architecture has increased lots in last 50 years
- Performance, which has also increased can be measured from standardised benchmarks
- Clock rate/frequency has also increased considerably in this time.

1.2 Computer Architecture

ISA: Boundary between hardware and software
Oragnisation: high level computer design aspects
Hardware: detailed logic and circuit design

Note: You want to separate your instruction set from implementation

1.3 Memory Organisation

- See memory as single 1D array
- Address is index of this array, points to byte of memory.
- Memory Access Time: time to read data to/from memory

- Memory Speed != Processor speed.
- Fast memory is very expensive. Heirarchy used to maintain fluid functionality and keep things cheap.

Processor Registers

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

Cache Memory

- Slower than register
- 8-256k
- Few nanoseconds access time
- Levels to this as well. L1, L2, L3 cache used in multiprocessor systems.

Main Memory

- Slower than cache
- Megabytes to gigabytes of size.
- Tens of nanoseconds lookup time.

1.4 Instruction Set Architecture (ISA)

ISA is interface between hardware and low level software.

Modern ISA include 80x86, MIPS, ARM

1.4.1 Using Fixed ISAs

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

Advantages

- 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 since it is so widely used in today's world.

1.4.2 ISA Design

Need to ask:

- What operations do the CPU need to do?
- How to provide data for given operations?
- How to store results of these calculations?

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.

1.5 Architecture Types

1.5.1 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.

1.5.2 Accumulator Based Archictecture.

- Using inputs from memory.
- Not used anymore today. Used in 1970s
- ADD function takes one operator, $mem_address$ which contains the value to add to above loaded value.

1.5.3 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

1.5.4 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

1.5.5 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

1.6 ISA Classes

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

1.6.1 Classes

RISC (Reduced Instruction Set Computers) is where size of all instruction words are the same. May lead to simpler decoding hardware.
  MIPS is an example processor that uses this type of ISA.

CISC (Complex Instruction Set Computers) are when the size of instruction words may vary. This is more complex than RISC but code footprint may become smaller due to condensing multiple RISC instructions into one CISC instruction.

Intel x86 is an example of processors based off this.

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

Intel Itanium uses this kind of ISA.

1.6.2 Abstractions

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

Instruction Processing in CPU

1. **Fetching** accesses memory to get to next instruction. Gets the correct memory address on the bus, and reads its contents.
2. **Decoding** Interprets the bits of the instruction word. Identifies which operation to do and data requied (from memory/registers)
3. **Execution** Performs the operation. Uses processor and writes result to register/memory.

1.6.3 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?

1.7 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

2.1 ALU Operations

- Addimmediate uses value, not pointer to register
- No Subimmediate as if Addimmediate allows negative numbers then we good.
- Destination register always before source register/s
- Register 0 is static containing all 0s, and cannot be written to

Endian-ness

1. **Little Endian** Least significant bit at top of memory addresses
2. **Big Endian** Most significant bit at the top of memory addresses

Sizes of things   in relation to memory size
    Word=4bytes
    halfword=2bytes
    1byte=byte

This course will use MIPS simlulator on PC called SPIM

Memory addressing in MIPS   machine is C(r_x)
    Where C is contstant which may be used to reserve part of memory.
    And Rx 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

2.2   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

2.3   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.

2.3.1   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.

2.3.2   I-Type Format

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

2.3.3 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 enocded information in instruction.
- Conditional Branching is **PC-relative** meaning that the PC provides current address and the Label provides an offset.

2ND HALF beq = branch equal, taking 2 inputs bne = branch not equal, taking 2 inputs. slide20 Else and Exit are kind of like GOTO (symbolic name) Need dollar sign if doing assembly.

bne and beq only have 16 extra bits. Therefore need to get to 32bits again somehow for 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 (from PC+4) + 400 (from 0x100 * 4 [left shift 2]) = 0100 0804 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.

Yay another example, so excited ATTEMPT Start: if pc bne $10 + 32*100 + 32 Exit shift right 3 pc

Start

Exit: hi there

ANSWER Need to make a counter, use addi for increment/decrement

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   //Store contents of R9 back into R10 addu $10, $10, 4 addi $11, $11, -1 bne $11, $0, L1

OP DES SRC

OP SRC DEST

Lecture 5 & 6 Example: Translate the below code to MIPS while(w[i] == x) i = i + j

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

ANSWER ¡¡SOMETHING MISSING¿¿ 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:

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 lessthan or greater than.

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

No implicit instruction to copy from one register to another. But can use contents of $0 with the ADD or OR $_{(haha)}$ function to achieve the same effect

Can create sudo instructions if use same set of instructions a lot (kind of like creating a function/script)

SECOND HALF OF LECTURES GO HERE, RAN OUT OF LAPTOP BATTERY

LEctures 7 & 8 Knowledge for assigment should be taught by next tuesday

Stackframes in MIPS have predetermined allocated area areas of stackframe Local and Temp variables Saved Register (+ return reg) argument build It will try to use registers if possible, but if memory requirements too high then will use stack frame Size of stack frame should be multiple of 16

Memory layout kernel bit top up to 0x7fffffffffffffffffffffffffffffff text segment program instruction program lower bit reserved for OS special task data segment objects with known size above text segment growing up

stack segment stack starting from 0x7fffffffff growing down

Problem If have nested functions then $31 contents will be overwritten. Therefore you save the value of $31 when you call more functions in the stack frame Use offset of 16 to store these values in the stackframe

Example Translate below to MIPS

main() int a, b, c, d, e, f, z; a = 10; b = -12; c = 120; d = 18; e = -2; f = 23; z = func1(a,b,c,d,e,f);

int func1(int x1, x2, x3, x4, x5, x6) int result; result = x1 + x2 - x3 + x4 + x5 + x6; return result;

Summary of stack frame usage SUMMARY HERE

Lecture 8 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 (.)

Lecture 9 MISSED IT

Lecture 10 (From slides 20/29 onwards) Can't just add a float to an int mtcZ/mfcZ is move to/from coprocessor. Z value = 1 represents co-processor for floating point This function just moves the bits to/from the registers and doesn't do the conversion

cvt = convert instruction .d.s = single precision convert to double precision .d.w = integer convert to double precision

Class example Do Z = 8*X + Y

Another example Check how many values in array are -Infinity

-Infinity cannot be represented so have to check contents of register to see if it contains proper value

Lecture 11 & 12 (2)Digital Circuit are classified as combinational or sequential circuits Combinational output only dependent on current input value Sequential output dependent on current input and state

(3) propagation delay exists (4) charge/discharge of load capacitance results in nonzero propagation delay (5) this nonzero prop delay may cause 'glitches' in the expcted output of the circuit Not concerned with details, but can't assume ideal case in the real world. Therefore should take prop delay into account at all times (6) Sequential circuits have some kind of storage element to keep current state of program clock to synch state transition (7,8) With flipflops, remember that it also has delays setuptime = minimum stable time for D input signal before change in CLK hold time = minimum time D has to be stable after CLK edge

FlipFlop = edge triggered. changes only occur when CLK positive or negative edge rise/fall Latch = level sensitive. changes only occur when CLK either 0 or 1

(9) IMPLEMENTATION AND ORGANISATION

(10) CPU implementation Control Unit Generate signals to direct datapath Datapath perform CPU operations

ALU performs on registers Has special extra outputs to show extra information about result given. Zero and Overflow Overflow for result that overflows 32bit Zero signifies the result being 0 or not

Needs 3 bits to say what operation to do SLT = Less than, can be performed by using subtract in ALU

Register File R type instruction need to take in 2 register values and return to specified register Uses to represent which registers need to be read and written to

Not using register read signal, but keep in mind for real world.

Program Counter PC Indicate address of next instruction to be executed Also have signals for reading or writing

Memory Interfacing used to access memory for instruction can add read signal if needed

Sign Extension Converts the 16 immediate value to the 32bit value needed needed for instructions. Also keeps the sign bit if needed (NOT SURE)

ISA subsert implementation Single cycle impelementation Assume single clock cycle long enough to accommodate the delays. This is the edges Therefore assume everything can be done in a single clock cycle

Duty cycle percentage of highs compared to lows (NOT SURE)

Fetch –¿ Decode –¿ Execute

Fetch Need the PC to get the address of instruction needed to carry out Instruction Momory to get instruction word Also need something to increment PC by 4, to be ready for the next instruction to be executed

R-Type Instructions need to get values from registers, so need Register File Need the ALU to perform some kind of operation on vlaues of these registers Need Register File to specify the output register for opertaion too

Load/Store If doing Load/Store then result of ALU needs to be fed into a Memory Intefacing which will then output to the Register File for destination register

Immediate Values into ALU get this value straight from register (not though Register File) and ensure that it goes through a Sign Extender first This is due to instruction needing the 32bit value to perform operation on, and the 16bit input needing to be converted to 32bit.

Example Impelementing BEQ Need to use ALU and Subtract, check the output Zero bit to see if values in registers are the smae Also need to generate the target address from the 16 bit value of immediate/offset from original instruction Use PC+4 and ALU. But cannot use ALU again since trying to do this all in the same clock cycle

If ALU(z) is true generate new PC value, equality check successful else use whatever PC currently contains, as equality check unsuccessful

Datapath and Control Unit

ALU used for [add, sub, and, or, lw, sw, beq] add - to calculate m]emory address sub - for beq ALUop[0,1] generated by main control unit, used for opcode conversion/translation for actual ALU operation input

Control Signal for ALU ALUop lw 00 sw 00 beq 01

Lecture 13 Single Cycle Datapath Impelementation cont. (11) not correct as inputs for read/write registers for src/dest of some operatiions 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 assert values on the output control signals without need of clock

Implementing Jump Instruction Jump insturctions 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 $10, $11, 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 impelmentation 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

Lecture 15 & 16 Multicycle implementation of MIPS ISA

- Mutltiple 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 elvel 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. PCSource
    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

Exaple 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.