COMPSYS304 Notes 2017

Theodore Oswandi

August 4, 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 assignent 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 (.)