Basic Instructions In MIPS

Prof. Charles W. Kann

Overview of Class

- mov, movw, and movt
- What is a register and an Arithmetic Logic Unit (ALU)
- What is an Instruction Set Architecture (ISA)?
- How does a 3-Address Load/Store Architecture work?
- Arithmetic
  - Addition and Subtraction
  - Multiplication
  - Division
- Boolean Operations
  - Bitwise Operations
  - Logical Operations
  - Shift Operations

mov

- **ARM has 3 move commands**
- **mov can be used to:**
  - Move (copy) the value from one register to another: mov r0, r1
  - Move an 8 bit (1 byte) immediate into a register: mov r0, 0x12
  - Note: if you try to move move than 8 bits, you will get the following error:

```
MovExample.s: Assembler messages:
MovExample.s:11: Error: invalid constant (4714) after fixup
make: *** [Makefile:22: MovExample] Error 1
```

movw and movt

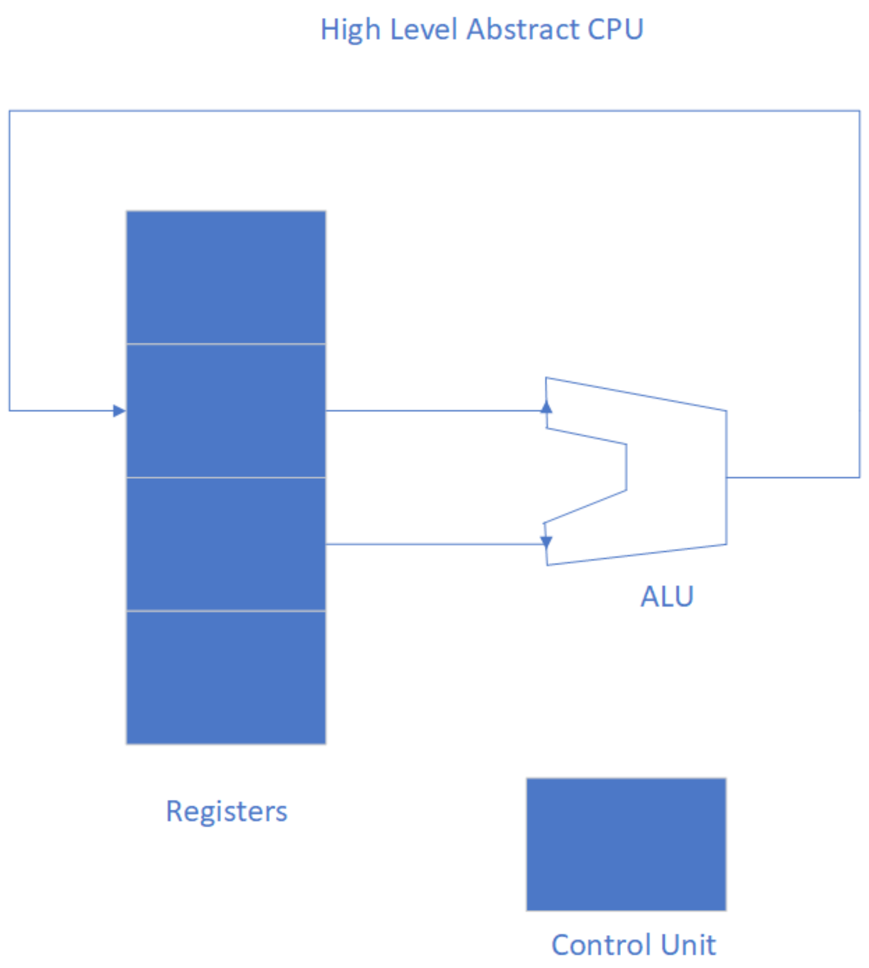
- **Registers are 32 bits wide.**
  - What if you need to load a 32 bit immediate into a register?
  - ARM has a movw (move wide) and movt (movt) commands that load 32 bits
    - movw loads the lower half (16 bits) of the register with an immediate
    - movt loads the upper half (16 bits) of the register with an immediate
- **However**
  - The movw and movt were not supported on the chip in my Pi zero.
  - To move a 32 immediate value to a register, you need to do the following:

```
#movw r0, #0x1234
#movt r0, #0x5678
mov r0,#0x12000000
orr r0,r0,#0x00340000
orr r0,r0,#0x00005600
orr r0,r0,#0x00000078
```

Main components of a CPU

- **At a very high abstract level, a CPU can be thought of as having 3 major components:**
  - Memory that is part of the CPU. Memory that is part of a CPU is called a register. Memory that is not part of the CPU (even cache memory that is on the die) is called memory.
  - An Arithmetic Logic Unit, or ALU. The ALU performs all calculations including arithmetic (add, sub, etc), as well as all bit operations such as AND, OR, shifting, etc.
  - A Control Unit, which interprets the instruction and configures the CPU to run the instruction.
  - The job of the CPU is to process instructions to the ALU and store the result

Abstract view of a CPU



Arithmetic Logic Unit

- **The Arithmetic Logic Unit**
  - Is a binary unit: it takes two inputs, processes them according to the instruction operation, and produces an output.
  - The Control Unit selects the two inputs from register memory, and the register that is to save the value produced out of the ALU.
  - The whole point of a CPU is the care and feeding of the ALU.

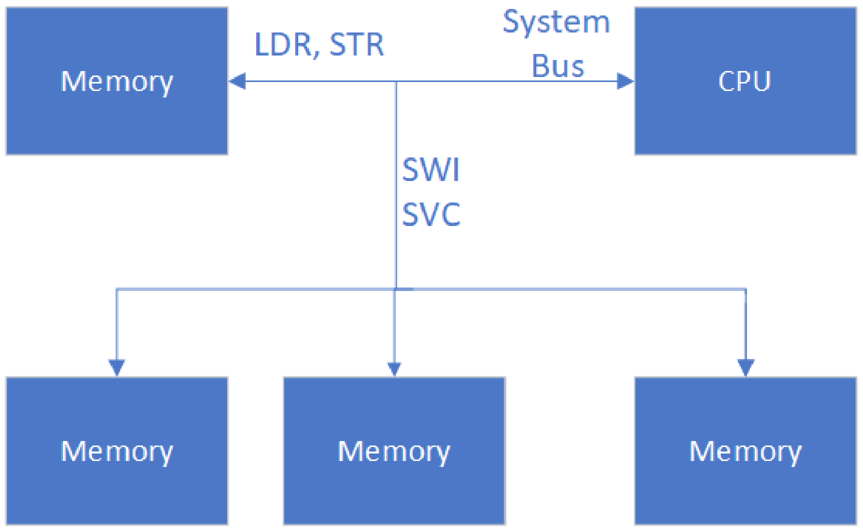
Why do we need an Instruction Set Architecture (ISA)?

- The number of registers is, on any CPU is by necessity and design, limited.
- To have a real computer, there is a need to access external memory and access to other system resources.
- The ISA is the definition of how values get from memory and external devices to registers so they can be used by the CPU.
- The ISA uses the system bus to retrieve values from memory or other devices, so the next slide explains the system bus.

What is a system bus?

- Memory values are passed to/from the CPU using the system bus.
- The system bus is bi-directional
- The instructions of the type ldr and str are used to access memory.
- Other system resources, such as the console, disk drives, network cards, etc., are also accessed via the system bus but use interrupts with system calls. This was shown in Module 1.

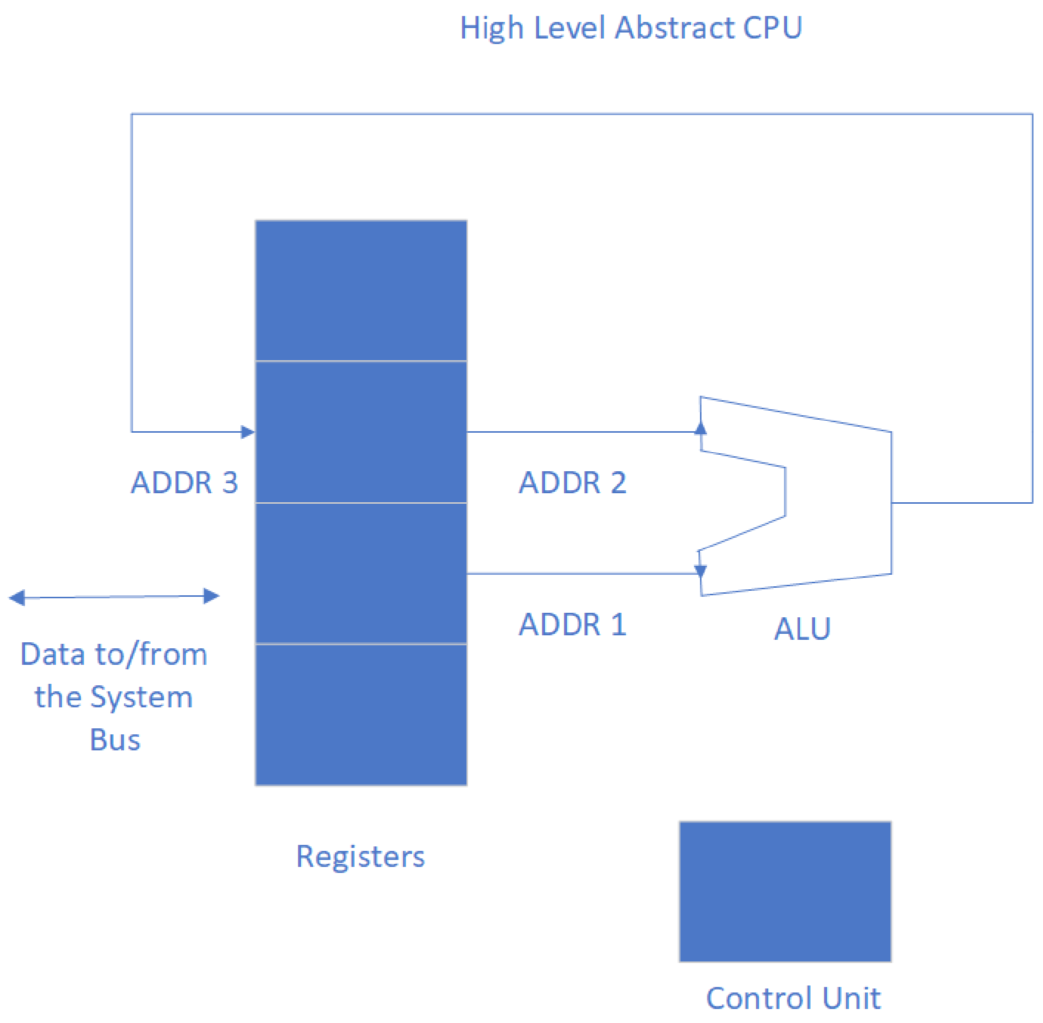
System bus



What is the ARM ISA?

- ARM uses a 3 address load/store ISA.
- The three address means all ALU operations specify 2 input registers and an output register, which are the three operands to the ALU, called addresses.
- The load/store means that any value to the ALU must be retrieved via the system bus into a register before it can be used.

3 Address Load/Store Architecture



Arithmetic Instructions

Addition and Subtraction

- **Adding/subtracting two registers**
  - ArithmeticExample_1.s
- **Addition/subtraction with an immediate value**
  - ArithmeticExample_2.s

Normal Multiplication

- **In most cases the multiplication will be a 32 bit value multiplied by a 32 bit number, with the result being a 32 bit value.**
  - mul R_d, R_m, R_s # $R_d = R_m * R_s$
- **Multiplication with accumulate**
  - mul R_d, R_m, $R_{s,}$ R_n # $R_d = R_m * R_s + R_n$
- **Notes:**
  - Note there is no immediate multiply
  - R_d cannot be the same as R_m
  - R_d cannot be R15

Other Multiplication instructions

- **ARM provides other multiplication instructions**
  - Shown in program OtherMulitiply.s
  - mla – multiply and add to accumulator, which is self explanator

    ```
    mla r5, r1, r2, r3 @ r5 ← r1 * r2 + r3
    ```

  - Smull – multiply with overflow, which requires more explanation on following slides

    ```
    smull r6, r7, r1, r2 @ r6 and r7 get the hi and lo part of multiplying r1 and r2
    ```

Multiplication with overflow

- Remember 9*9 = 81 (two one digit number multiplied can be 2 digits)
- Remember 99 * 99 = 9801 (two two digit number multiplied can be 4 digits)
- So, two 32 bit numbers multiplied together can result in a 64 bit number.
- ARM has instructions to allow access to the upper 32 bits:
  - UMULL, UMLAL - Unsigned Multiply and Multiply with Accumulate
  - SMULL, SMLAL – Signed Multiply and Multiply with Accumulate
  - Note: The main reason to use these is this class is to check for multiply overflow.
  - SMULL R_{dLo}, R_{dHi}, R_m, R_s
  - Note that the answer is in R_{dLo}, and any overflow is in R_{dHi}. Suggest an algorithm to check for overflow (you do not have to implement it now, you will implement it later in the semester.)

Division

- Division is not included in many chips, including the version of the ARM chip used in the Pi 0. The reason is division is expensive compared to other hardware, and can be done in assembly as cheaply as it can be done in hardware.
- https://www.quora.com/Why-doesnt-ARM-contain-divide-instruction
- There is a sdiv (signed division) and an udiv (unsigned division) operation in ARM assembly, but it may or may not be implemented in your chip (it will be on a Pi 2, 3, and 4).
- On the Raspberry Pi 0, the sdiv and udiv are not included.

How this class will implement division

- Later in the class, we will write a slightly faster division algorithm.
- For now, ARM provides a helper function for division, __aeabi_idiv. This function will be used to implement division.
- For this function, r0 is the dividend, r1 is the divisor, and r1 is the quotient.
- See Divide.s
- Remember that integer division truncates the remainder.

Logical and Bitwise Instructions

Logical vs bitwise operations

- **Logical vs Bitwise Operators**
  - In high level languages (hll, like Java or C++) logical operators result in a true or false value. The operands are a single bit, and can short circuit. In Java and C++, they are operators like && or ||
  - Bitwise operators are used to operate on all bits in operand values. Operands generally contain n bits (n > 1), and each bit from the first operand and the second operand are used to create a n bit resultresult. These operands cannot short circuit. In Java and C++ they are operators like &, |, ^, etc…
- **Logical Operators In Assembly**
  - Assembly does not have a concept that equates to a hll Logical operator. All Boolean operations are bitwise, but these operators are called *Logical* operators. Just realize the naming convention, and accept it. It is confusing, but it is what it is.

Bitwise and Logical operations in Assembly

- **In assembly**
  - All logical operators are bitwise, they work on each individual bit in the operand.
  - Bitwise operators are used to accomplish what is called *bitmasking*. Bitmasking turns on or off specific bits.
  - For example, the network device might turn on bit 4 of a 8 bit IO status word. By using AND on with word with binary 00010000 (or 0x10, with bit numbering being zero based), any bits in the IO status word except bit 4 are turned off.
  - For example, if the IO status word contained 11011001, the result of the and is:

```
  11011001
& 00010000
  00010000  Only the 4th bit is left on.
```

Logical Operators in ARM

- **ARM has the following logical operators**
  - AND, OR, EOR, and MOVN
  - EOR is for an XOR operation, MOVN is used for a binary NOT operation
- **ARM has the following shift operations**
  - LSL – Logical Shift Left
  - LSR – Logical Shift Right
  - ASR – Arithmetic Shift Right
  - ROR – Rotate Right
  - Note that there is a RRX operation, but that instruction will have to wait until the basic ARM machine code is covered.

Bitmask example: using AND and ORR ASCII operation to convert upper case to lower case

- Note that in ASCII, and upper case ‘A’ (0x41) and lower case ‘a’ (0x61) differ by one bit (0x20).
- An upper case ‘A’ can be converted to a lower case ‘a’ by turning on that bit (and only that bit) as follows (this is faster and more correct than addition, for anyone who thought about that).

```
 # Change upper case to lower case letter
  mov r0, #0x40
  orr r0, r0, #0x20
# examine value in r0, should be 0x60

# Change lower case to upper case letter
  and r0, r0, #0xdf
# examine value in r0, should be 0x40
```

EOR (XOR) Instruction

- **The ARM eor instruction is a very useful instruction**
  - It can reduce the number of operations that are needed to implement a circuit.
  - It can be used to store a reversible value
  - It can be use to store two values at the same time
- **The following example shows how to store and retrieve values with an EOR**

```
mov r0, #0x47
mov r1, #0xf4
EOR r2, r0, r1
# r2 now contains both r0 and r1!
# to show this retrieve r1 into r0
eor r0, r2, r0
```

Shift instructions

- **The following shift operations are available in ARM assembly:**
  - LSL – Logical shift left. Shifts all bits #n places to the left, inserting 0’s. Used for multiplication and to examine bits one at a time in a word (e.g. string processing).
  - LSR – Logical shift right. Shifts all bits #n places to the right, inserting 0’s. Used to examine bits one at a time in a word (e.g. string processing).
  - ASR – Arithmetic shift right. Shifts all bits #n places to the right, inserting the 2’s complement sign bit. Used to examine bits one at a time in a word.
  - ROR – Rotate right. Shifts bits to the right, and as they shift out they are appended to the left of the word (e.g. string processing).
  - See ShiftExamples.s for examples. The shifts can be viewed in gdb.