Chapter 2

Word

- The natural unit of access in a computer
- 32 bits in MIPS

Registers

- primitives used in hardware design
- size of a register is a word
- Why do we only have 32 Registers?
    - Design Principle: Smaller is faster
        - Takes electronic signals longer when they must travel farther
    - Also for the number of bits it would take in the instruction format
- Have special characters/names for different types of registers

Memory Operands

- Big data structures such as arrays and structs are stored on the memory (another one of the five components of a computer) instead of in the limited space of the registers
- MIPS only operates on registers, so we need **data transfer instructions** to bring data from memory into registers
    - Instructions must supply the memory address
        - Memory can be thought of as a large,single-dimensional array, with the address acting as the index into the array
    - Data transfer instructions are called load (copies data from memory to a register) and store (copies data from a register to a memory)
        - Includes a register for a base address and a constant for the offset from this base address for the memory

Spilling

- Programs tend to have more variables than registers
- Compiler tries to keep most frequently used variables in registers and rest in memory
    - Uses loads and stores to move variables between registers and memory
    - Process of putting less commonly used variables or those needed later is called **spilling**
- Registers are both faster and more efficient than memory (MIPS arithmetic instruction can read two registers, operate on them, and write the result where as a data instruction only reads/writes one operand without operating on it)
    - Registers have higher throughput and less time to access and therefore need to be used efficiently by compiler

Immediate Operands

- If we just continue to use these regular arithmetic instructions, every time we had to do something with a constant, we would have to load the constant from some place in memory where it is stored
    - Extremely inefficient, especially noting the fact that these are so common
- Have special instructions that deal with immediate

- Reflects Design Principle 3: Makes the common case fast
- The constant 0 is used so often (think, the instruction move is simply an add with a 0) that it is hardwired into the register $zero or $0

Signed and Unsigned Numbers

- Everything is stored in computers as 1's and 0's
    - kept as a series of high and low electronic signals
- In any number base, the value of the ith digit d is d x Base^i
    - Number the bits 0, 1, 2, 3, ... from right to left in a word
    - The rightmost bit (bit 0) is the **least significant bit**
    - The leftmost bit (bit 31) is the **most significant bit**
- 32 bits can represent 2^{32} different bit patterns
    - Unsigned ints represent the numbers from 0 to 2^{32} - 1
- **Overflow** occurs when the proper result of these arithmetic operations (add, subtract, multiply, divide) cannot be properly represented by the amount of bits we have
- Keep in mind that bit patterns are simply representatives of numbers
- We need to differentiate between positive and negative numbers:
    - Sign and Magnitude:
        - Add a separate bit to indicate sign
        - Several shortcomings:
            - Don't know where to put the sign bit
            - Might need an extra step to get the sign
            - Has two zeros (a positive and a negative one)
- What happens if we try to subtract a larger unsigned number from a bigger unsigned number?
    - It would try to borrow from a string of leading 0s, so the result would have a string of leading 1s
    - Therefore, the final solution was for leading 0s to represent a positive number and leading 1s to represent a negative number
        - Know as **Two's Complement**
            - Easy on the hardware:
                - Add just as normally and it will take care of itself
                - Every negative number has a 1 in it's most significant bit and the opposite holds for positive numbers
            - The positive half of the numbers (from 0 to 2^{31} - 1) represented the same as before
            - The negative half goes from (1000......0; -2^{32}) to (1111....1; -1) in a declining set of negative numbers
                - You calculate the value of a number just as before, by figuring out the value of each digit as mentioned above and adding them up together. However, if the most significant bit is 1, then you do (d x -2^{msb}) instead of positive 2.

- Just like an operation on unsigned numbers can overflow the capacity of hardware to represent the result, so can an operation on signed numbers
  - This happens when the most significant bit doesn't match up to what sign the result should be
    - Essentially, this means that the msb of the result is a 0 when the result should be negative or a 1 when the result is positive
  - Shortcut for Two's Complement Numbers:
  - Flip all the bits of a number *x,* and call this x^*
  - $x + x^*$ must be represented as $11111.....111_{two}$
    - This means that $x + x^* = -1$, or that $$x^* + 1 = -x$$
    - So say x is 3, and you want to find what -3 is
      - You can simply invert all of the bits of 3 (so 011 becomes 100) and then add 1 (101) which is:
        - $(1 \times -2^2) + (0 \times 2^1) + (1 \times 2^0) = -3$
- To represent an n-bit number with more than n bits, we simply **sign extend**
  - We take the most significant bit, and fill in the new higher-order bits with it
    - This works because positive two's complement numbers really have an infinite number of 0s on the left and negative two's complements numbers have an infinite numbers of 1s

Representing Instructions

- Instructions are stored as a series of high and low electronic signals and can be interpreted as numbers
- Each piece of an instruction is essentially a number
  - We need a way to represent registers as numbers because they appear in instructions
    - For example $t0 to $t7 map onto 8 to 15 and $s0 to $s7 map to 16 to 23
- Instruction format: the form of representation of an instruction composed of fields of binary numbers
- Can use hexadecimal (base 16) to make it more convenient
  - Base 16 is a power of 2, so we can trivially convert by replacing each group of four binary digits by a single hexadecimal and vice versa
- Instructions in MIPS are stored as 32 bits (size of a word) to ensure simplicity/regularity

R-Type Instructions

- Stands for Register
- R-type instructions have 6 fields - *op*, *rs*, *rt*, *rd*, *shamt*, and *funct*
  - *op (6 bits)*: Basic operation of the instruction, called the **opcode**
  - *rs (5 bits)*: The first register source operand
  - *rt (5 bits)*: The second register source operand

Enter summary here

  - *rd (5 bits)*: The register destination operand. It gets the result of the operation.
  - *shamt (5 bits)*: Shift amount
  - *funct (6 bits)*: Function. This field selects the specific variant of the operation
- Note that the opcode for R-type instructions is 0 because we have space for the funct field and this allows us to use more opcodes for other types of instructions
- The registers are all represented as 5 bits because we have 32 registers

I-Type Instructions

- Stands for Immediate
- If we were to continue to use R-Type instructions and use one of the 5-bit fields to represent our constants, we would only reach 2^5 different constants, or 32. This certainly isn't enough, especially for accessing array and structures or generic arithmetic
- There is a desire to keep all instructions the same length and to have a single instruction format
  - We introduce a new instruction format
    - Reflect design principle: Good design demands good compromises
    - Compromise: Keep all instructions the same length, but introduce new types
- There are only 4 fields now:
  - *op (6 bits)*: The opcode; same meaning as R-Type
  - *rs (5 bits)*: The source register; same meaning as R-Type
  - *rt (5 bits)*: Specifies the destination register; unlike R-Type
  - *immediate (16 bits):* The constant used in the operation
- Note that the immediate is much larger and therefore we are able to represent 2^{16} bit patterns with it as opposed to just 32
- Even though the R-Type and I-Type instructions differ, the first three fields of the two are the same, which makes it easier on the hardware.
  - The hardware also knows whether to treat the last half of the instruction as three fields (R-Type) or as a single field (I-type) depending on the opcode

- Today's computers are built on two key principles:
  - Instructions are represented as numbers
  - Programs are stored in memory to be read or written, just like numbers
- This has led to the stored-program concept
  - Memory can contain the source code for a program, the corresponding compiled machine code, the data for the program, and the compiler
- Computers that do one particular task, like accounting, can in an instant switch over to another task, like text editing, by simply loading memory with

programs and data and telling the computer to begin executing at a given location in memory

Logical Operations

- Instructions that simplify the packing and unpacking of bits into words

Shift

- Moves all the bits in a word to the left or right, filling the emptied bits with 0s
    - sll (in MIPS/ << in C) shifts to the left by the specified amount and srl (in MIPS/ >> in C) shifts to the right by the specified amount
- Shift right arithmetic is distinct from shift right logical in that it sign extends the number as it shifts it to the right
- Shifting left by *i* has the added benefit of multiplying by 2^i

And

- Bit-by-Bit operation (represented as & in C)
- Leaves a 1 in the result if and only if both bits of the operands are 1
- Can be used to apply a bit pattern to a set of bits to force 0s where the is a 0 in the bit pattern
    - Known as a **mask**

Or

- Bit-by-Bit operation that places a 1 in the result if either operand bit is a 1
- Dual to And
    - Used to place a value by automatically putting 1s in the result where there are 1s in the bit pattern

Not

- Logical Bit-by-Bit operation with one operand that inverts the bits
    - Replaces every 1 with a 0, and every 0 with a 1

Nor

- A logical bit-by-bit operation with two operands that calculates the *not* of the *or* of two operands
    - Calculates a 1 only if there is a 0 in both operands

Xor

- Known as the exclusive or
- Sets the bits to 1 when two corresponding bits differ, and to 0 when they are the same

Instructions for Making Decisions

- Decision making is crucial as based on the input data and values created during the computation, different instructions execute
- In MIPS, there are two decision-making instructions - beq and bne
    - These are like if statements in C with a go to
- These two are **conditional branches**
    - Defined as instructions that require the comparison of two values and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the comparison
- If tests utilize these branches, as well as loops

- Another useful instruction is *slt*, which compares two registers and sets a third register to 1 if the first is less than the second (instruction is known as *set on less than)*
    - There is also an immediate version of *slt*, known as *slti*, which deal with constant operands.
- MIPS compilers use the *slt, slit, beq, bne,* and the fixed value of 0 to create all of the relative conditions: equal, not equal, less than, less than or equal, greater than, greater than or equal
    - Don't include instructions such as branch on less than because it is too complicated
        - Either it would stretch the clock cycle time or it would take extra clock cycles per instruction
        - Two faster instructions are more useful
- With comparison instructions, must deal with the fact that we have unsigned numbers and signed numbers
    - Unsigned numbers means that a 1 in the most significant bit is automatically bigger than any number without a 1 in the msb, whereas a signed number means that a 1 in the msb is a negative number, which is thus less than any positive number (a number with a 0 in the msb)
    - MIPS offers two versions of *slt* - *slt* and *slti* work with signed whereas *sltu* and *sltui* work with unsigned
        - If you took two signed integers and did an unsigned comparison, $x < y$, it would also check if x is negative as well as if x is less than y because of the fact that an msb of 1 indicates a negative number for signed but a really large number for unsigned
- Case/Switch Statement's simplest implementation is a sequence of *if-then-else* statements
    - An alternative, more efficient way would be to use a jump address table where the program indexes into the table and then jumps to the address there (to the appropriate instruction sequence)
        - The program loads the appropriate entry from the jump table to a register and then use the instruction *jr* to jump to the address held in it

Supporting Procedures

- Procedure/function is a stored subroutine that performs a specific task based on the parameters with which it is provided
    - A procedure is like a spy that leaves with a secret plan, acquires resources, performs the task, covers his tracks, and returns to the point of origin with the desired result
    - Nothing else should be perturbed once the mission is complete and a spy operates on only a "need to know" basis
- Execution of a program must follow:
    - 1. Put parameters in a place where the procedure can access them

- Transfer control to the procedure
- Acquire the storage resources needed for the procedure
- Perform the desired task
- Put the result value in a place where the calling program can access it
- Return control to the point of origin, since a procedure can be called from several points in a program

- Want to optimize use of registers, so MIPS follows following for procedure calling:
  - $a0 - a3: four argument registers in which to pass parameters
  - $v0 - $v1: two value registers in which to return values
  - $ra - one return address register to return the point of origin
- MIPS includes the instruction *jal* that jumps to an address and simultaneously saves the address of the following instruction in a register ($ra for MIPS)
- The return address is a link to the calling site that allows a procedure to return to the proper address (this is important because the procedure can be called from several parts of the program)
- MIPS includes *jump register* instruction (jr), meaning an unconditional jump to an address in a register
  - We would do *jr $ra* to return back to the caller
- The caller is the program that instigates a procedure and provides the necessary values
- The callee is a procedure that executes a series of stored instructions based on parameters provided be the caller and then returns control to the caller
- The PC (program counter) is the register containing the address of the instruction in the program being executed
  - The *jal* instruction saves PC+4 in $ra to link o the following instruction to set up the procedure return

Using More Registers

- A subroutine must cover it's track after the mission is complete, any registers needed by the caller must be restored to the values that the contained before the procedure was invoked
- A stack is a data structure for spilling registers
  - Organized as a last-in-first-out queue
  - Needs a pointer to the most recently allocated address in the stack to determine where the next procedure should place the registers to be spilled or where old register values are found
    - Known as stack pointer and stored in $sp
    - Adjusted by one word for each register that is saved or restored
- Placing data onto the stack is called a push and removing data from the stack is called a pop
- Stacks grow from higher addresses to lower addresses
  - Push values by subtracting from $sp
  - Pop values by adding to $sp

- A process may need more registers for a procedure than the four arguments and two return value registers
    - We spill these onto the stack
        - For extra parameters, we place them on the stack just above the frame pointer and can be accessed from memory
- We must also save the registers used by the caller before moving to the callee
    - In order to not save each register each time (really expensive) we have the following convention:
        - $t0 - $t9: ten temporary registers that are not preserved by the callee
        - #$s0 - $s7: eight saved registers that must be preserved

Nested Procedures

- Procedures that do not call others are called *leaf* procedures
    - There are many times when a procedure will call another subroutine
        - One such instance is recursion
- A procedure that is in progress that calls another procedure is not done when the second procedure returns, so the caller pushes any argument registers ($a0 to $a3) or temporary registers ($t0 to $t9) that are needed after the call
- The callee in turn pushes the return address register $ra and any saved registers ($s0 - $s7) used by the callee
    - $sp is adjusted accordingly
    - Upon return the registers are restored from emory ant the $sp is readjusted
- To reduce all this overhead, certain programs can be rewritten iteratively without using recursion

Static

- A C variable depends on two things - it’s type and storage class
- For type, examples include integers and characters
- C has two storage classes: automatic and static
- Automatic variables are local to a procedure and are discarded when the procedure exists
- Static variables exist across exits from and entries to procedures
    - C variables declared outside all procedures are considered static
    - Also variables declared with the keyword, static
    - The rest are automatic
- To simplify access to static data, we have a register called the global pointer, or $gp

Stack

- The stack is also used to store variables that are local to the procedure but don’t fit in registers
    - Local arrays or structures

- Segment of the stack containing a procedure’s saved registers (ranging from argument to temporary to $sx registers)/local variables is called a procedure frame or activation record
- Can use a frame pointer ($fp) to point to the fist word in the frame of a procedure
    - The stack pointer may change during a procedure so references to local variables might have different offsets depending on the time of access
        - A frame pointer offers a stable base register within a procedure
    - If there are no local variables, the compiler usually doesn’t bother with saving and restoring the $fp
        - If it does use it, it sets the $fp to the sp on a call and $sp is restored using $fp

Heap

- Programmers need memory for static variables and dynamic data structures on top of automatic variables
- There is a set convention for allocating memory:
    - The stack starts at the top of memory and grows down
    - The first part of memory (the lowest addresses) are usually reserved
    - Then comes the text segment of the code
        - This is the segment of a Unix object file that contains the machine language code for routines in the source file
    - Then comes static data segment
        - Used for constants and other static variables
    - Then comes the heap
        - Known as dynamic data and used for array, structs, linked lists, and so on so forth
        - The stack and the heap grow towards each other because they are both dynamic
            - The stack grows down and the heap grows up
- C allocates and frees space on the heap with malloc() (allocates memory and returns a pointer to it) and free() (frees the allocated memory associated with a pointer)
    - Forgetting to free space is a memory leak
        - Program uses up so much memory that the operating system may crash
    - Also there is the issue of dangling pointers
        - Occurs when freeing space too early
            - Cause pointers to point to things that the program never intended

Characters

- We represent a single character in a single byte (8 bits) with each bit pattern corresponding to an unique character
- Computers needed a standard for representing text and characters

    - The most popular one is the ASCII standard
- An interesting tidbit is that numbers could really be represented as a string of ASCII digits instead of integers (represents how everything is just in terms of 1s and 0s)
- Because char's are 1 byte in MIPS, we see that there are many times when we need to load just one byte instead of an entire word
    - We could use the logical instructions provided to extract the byte from the word
    - MIPS have instead provided an analog to the word data transfer instructions - lb and sb
        - These two instructions will sign extend, but for characters, this does not make sense, so we use lbu (load byte unsigned)
- Strings of characters are represented by C by terminating it with a byte whose value is 0 (which is "null" in ASCII)
- Because the MIPS software tries to keep the stack aligned to word addresses, because we wish to use lw and sw with the stack and lw and sw must be word aligned, a char variable allocated on the stack occupies 4 bytes even though it only needs 1
    - A C string variable or an array of bytes will pack 4 bytes per word

Psuedoinstructions

- Sometimes we need the full 32 bits to specify a constant
    - We have the MIPS instruction lui (load upper immediate) which sets the upper 16 bits of a constant into a register
    - We can then follow up this instruction to set the lower 16 bits of a register (usually something like an ori)
- Either the compiler or the assembler must deal with the splitting and reassembling of the constants
    - In MIPS, this falls to the assembler, which therefore has the $at register reserved for itself for tasks like these

J-Type Instructions

- J-type instructions are used for jumps and have only two fields
    - *op (6 bits)*: This is the opcode, and has the same meaning as before
    - *address (26 bits)*: This is the address where we are jumping to
- These are also known as **unconditional branching**
- Since all MIPS instructions are 4 bytes long, they are word addressed
    - Therefore, we can treat our address in jumps by the number of instructions/words we are jumping and not bytes
        - We can now branch 4 times as far, and the 26-bit field represents a 28-bit byte address
        - Simply multiply by 4 to get our real address
    - We only have 28 bits of our address, so the remaining must come from the top 4 bits of the current PC

- The loader/linker must be careful not to place a program across an address boundary of 256 MB (64 million instructions)
    - In such a case, we must use that has the full 32-bits address

Branch Target Address

- For branches, we only have 16 bits for the address field, but we can apply the same word addressing principle as we did for jumps
    - This means that our programs would only be restricted to a size of 2^{18} instructions
        - Instead, we should specify a register that we add to the constant so programs can still be as large as 2^{32}
        - We choose the PC as the register we add to, because conditional branches are found in loops and *if-else* statements, and tend to branch to a nearby instruction
            - This is known as **PC-relative addressing**
    - If it is not a nearby instruction, we use the j-type instructions instead
        - The assembler will insert an unconditional jump to the branch target, and inverts the condition so that the branch decides whether to skip the jump

Synchronization

- Parallel tasks often need to cooperate (usually means some tasks are writing new values that others must read)
    - Tasks need to synchronize, to know when a task is finished writing so that it is safe for another to read
    - Without synchronization, a **data race** may ensue
        - Two memory accesses form a data race if they are from different threads to the same location, at least one is a write, and they occur one after another
        - results of a program may change depending on how events happen to occur
- We wish to atomically read and modify a memory location
    - Nothing can interpose itself between the read and the write of the memory location
- We can see a data race forming between two processes if they are both trying to lock a particular memory location and modify it
- A pair of instructions is effectively atomic if it appears as if all other operations executed by any processor occurred before or after the pair
    - There is a pair of instructions called a *load linked* and a *store conditional*.
        - If the contents of the memory location specified by the load linked are changed before the store conditional to the same location occurs, then the store conditional fails

- The store conditional will store the value of a register in memory and change the value of that register to a 1 if it succeeds and to a 0 if it fails

Compiler

- The compiler transforms the C program into an assembly language program
  - A symbolic form of what the machine understands

Assembler

- An assembly language is an interface to higher-level software
  - The assembler can treat common variations of machine language instructions as if they were instructions in their own right
    - Known as psuedoinstruction
    - One example is the psuedoinstruction *move* that copies the contents of one register to another
      - This is simply *add $t0, $zero, $t1*
    - Another example is where the assembler converts *blt (branch on less than)* into two instructions - *slt* and *bne*
- The assembler converts the assembly language program into an **object file**
  - This is a combination of machine language instructions, data, and information needed to place instructions properly in memory
- To produce the binary version of each instruction in the assembly language program, the assembler must determine the address corresponding to all labels
  - Keeps track of labels used in branches/data transfer instructions in a **symbol table**
    - Contains pairs of symbols (labels) and addresses
- The 6 distinct pieces of an object file:
  - *Object file header:* Describes the size and position of other pieces of object file
  - *Text segment:* Contains the machine language code
  - *Static data segment:* Contains data allocated for the life of the program
  - *Relocation information*: Identifies instructions and data words that depend on absolute addresses when the program is loaded into memory
  - *Symbol table:* Contains the remaining labels that are not defined, such as external references
  - *Debugging information*: Concise description of how the modules were compiled so that a debugger can associate machine instructions with C source files and make data structures readable

Linker

- A linker is a systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file

- Allows for programs to be compiled in modules to avoid compiling the entire program every time a change is made
- The three steps for a linker:
  - Place code and data modules symbolically in memory
  - Determine the addresses of data and instruction labels
  - Patch both the internal and external references
- Linker uses the relocation and symbol table in each object module to resolve all undefined labels
- The linker then determines the memory locations each module will occupy
  - All absolute references (memory addresses that are not relative to a register) must be relocated to reflect its true location
- The linker produces an **executable file** that can be run on a computer
  - A functional program in the format of an object file that contains no unresolved references
  - Can contain symbol tables and debugging information

Loader

- The loader is a system program that places an object program in main memory from disk so that it is ready to execute
  - Reads the executable file header to determine size of text and data segments
  - Creates an address space large enough for the text and data
  - Copies the instructions and data from the executable file into memory
  - Copies the parameters (if any) to the main program on the stack
  - Initialize the machine registers and sets the stack pointer to the first free location
  - Jumps to a start-up routine that copies the parameters into the argument registers and calls the main routine of the program
    - When the main routine returns, the start-up routine terminates the program with an *exit* system call
-