**Design Considerations for CS6461 Group Project:**

Complex Simulation of Simple Computer System

# Introduction:

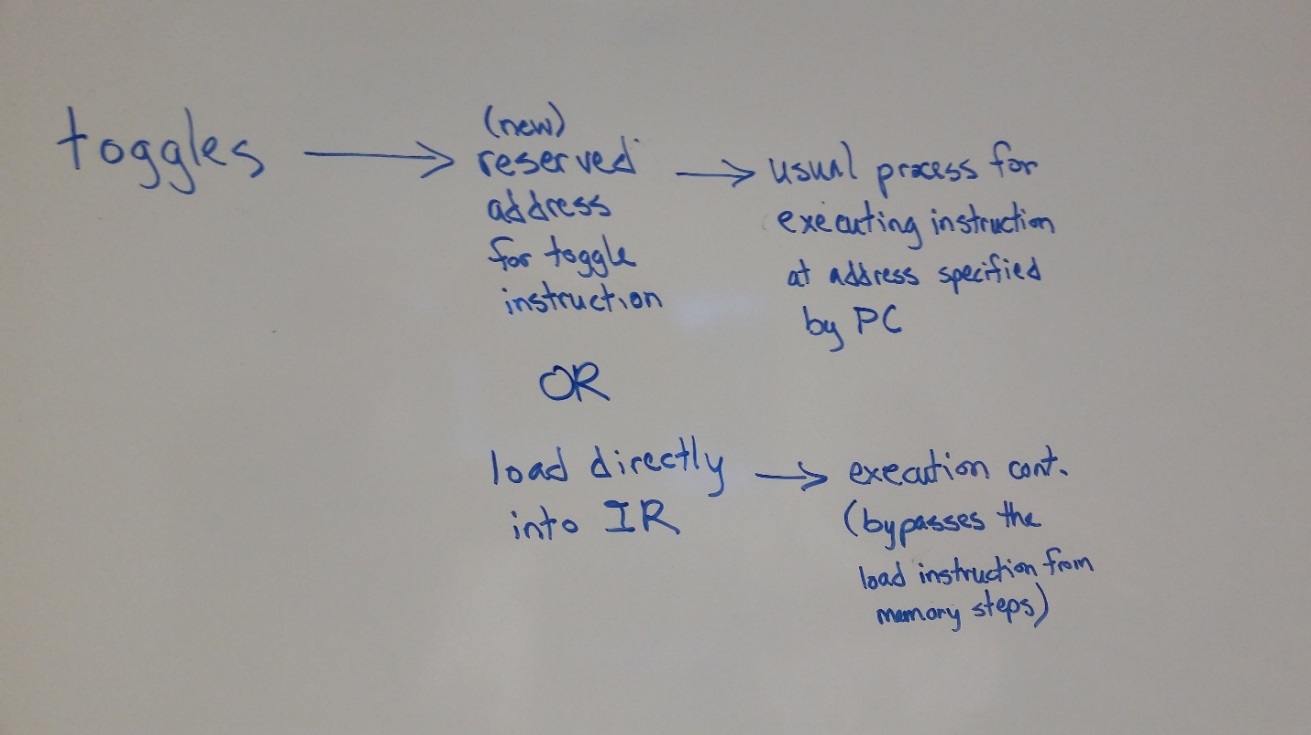
This document highlights the initial design formulation of group project for CS6461 Computer System Architecture, which focuses on simulation of a modern computer at the machine level. The development of this simple machine is broken down into four major phases or sprints. The primary requirements for the system are contained in our Class Project Description document. Key portions of these requirements are also extracted from that word document and contained in the deign-requirements table contained within this document. This document contains initial frontend and backend designs. This document is currently a living document and will evolve as the simulation design evolves.

# Design Concepts:

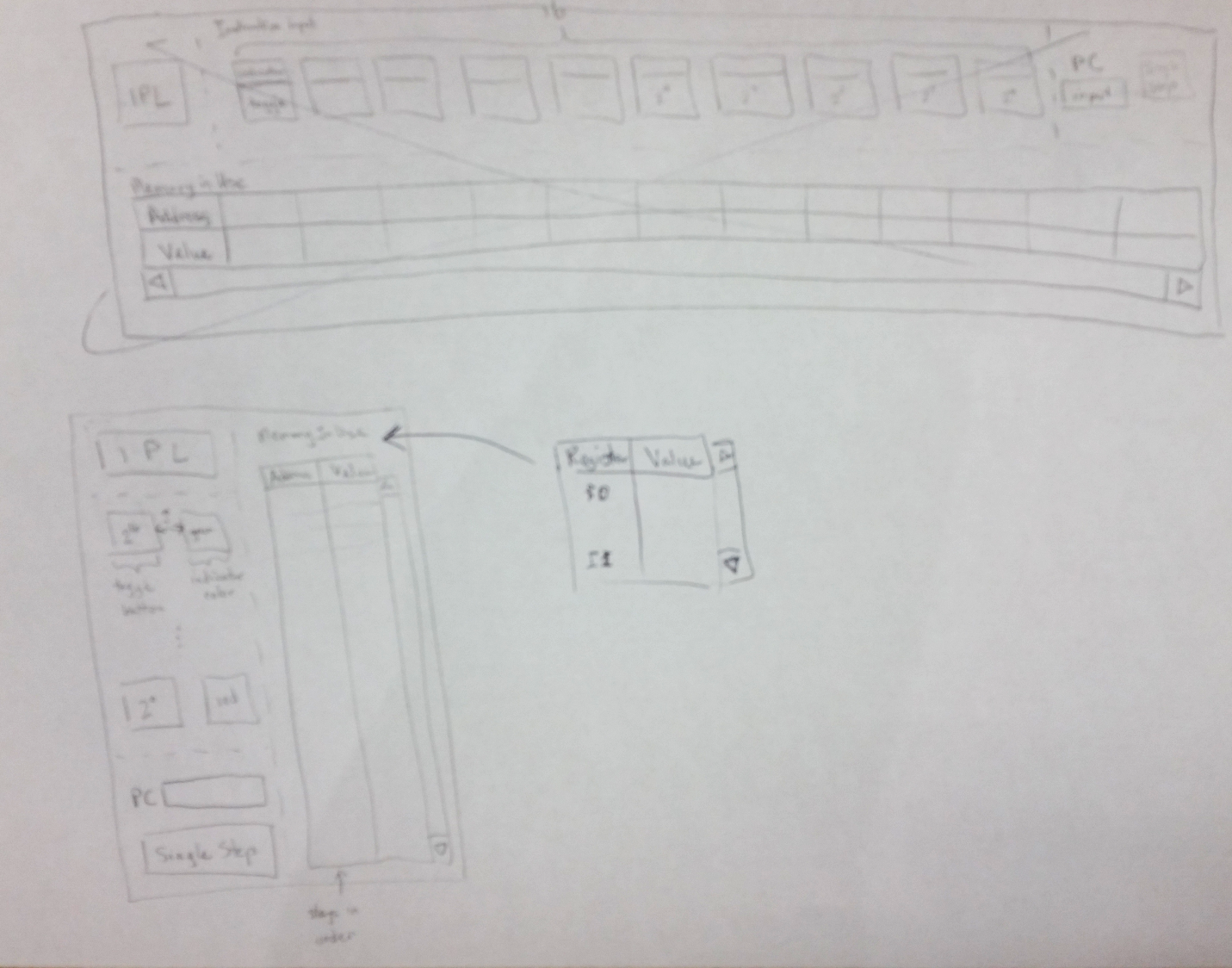
The design concept section contains the initial choices and considerations made while designing the simple-computer program simulation

## Front-end Design Concepts:

1. The Memory panel GUI table should be empty before IPL is pressed, since pressing IPL should load the memory.
2. The value in the toggle switches (i.e. the instruction) will be input for MiniComputer.



1. MiniComputer will either:
   * Load that value into IR (is there danger in overwriting what was there before from some program?)
   * Or we add a new Register whose purpose is to store the instruction that is directly inputted from the GUI (Lancaster did say we would need more registers than what he listed).
2. The GUI Memory panel will show the values for memory addresses that are being used, so we don't have to hunt through a list of 0-2048 addresses!
3. The GUI will roughly look like:



## Back-end Design Concepts:

1. The binary inputs (e.g.16 bits in binary) we will read in as strings
2. We will create objects and encapsulate our code -- this may be overkill in some areas, but in general this will ensure we are using standardized and expected objects.
3. We will consider using Enums or constants for Opcodes, fault codes, and condition codes
4. Our instructions will be binary strings since they allow us to have leading zeros and is still easy to parse (we also considered ints and arrays)
5. We will use GitHub as our Centralized Code Repository
6. We will use Java Swing to build the GUI
7. We will use a HashMap<String, MemoryLocation> to store the memory being used.
8. MemoryLocation will have an isAddressReserved() method, actual reserved address business logic will be handled in our ‘minicomputer’ main program
9. We will follow Java naming conventions: variableName, CONSTANT\_NAME, ClassName

## Major Class Structure Design of Back-end:

The main classes of the back-end should be:

* MiniComputer – our main control flow class. Conceptually, it will contain:
  1. The Registers:
     1. PC: Register
     2. CC: Register
     3. IR: Register (Note: Our IRR class is an array of Registers, so that we may use multiple if necessary)
     4. MAR: Register
     5. MBR: Register
     6. MSR: Register
     7. MFR: Register
     8. R0: Register
     9. R1: Register
     10. R2: Register
     11. R3: Register
     12. X1: Register
     13. X2: Register
     14. X3: Register
     15. Other registers we may need.
  2. Methods:
     1. calculateEffectiveAddress(int indexRegister, bool isIndirectAddress, BitWord address): BitWord
     2. loadRegisterFromMemory(int register, int index, bool isIndirectAddress, BitWord address): void
     3. storeRegisterToMemory(int r, int x, bool i, BitWord address): void
     4. loadRegisterWithAddress(int r, int x, bool i, BitWord address): void
     5. loadIndexRegisterFromemory(int x, bool i, BitWord address): void
     6. storeIndexRegisterToMemory(int x, bool i, BitWord address): void
     7. Other instruction methods that are added in future phases…
* OpCode – “library” of the codes for the available operations our computer is capable of performing
  1. HLT
  2. TRAP
  3. LDR
  4. STR
  5. LDA
  6. LDX
  7. STX
  8. AMR
  9. SMR
  10. AIR
  11. SIR
* FaultCode – “library” of the fault codes within the simple computer. Handling Machine Faults is handled in depth in phase III. The class is implemented as an Enum within JAVA.
  1. ILLEGAL\_MEMORY\_ADDRESS\_TO\_RESERVED\_LOCATION
  2. ILLEGAL\_TRAP\_CODE
  3. ILLEGAL\_OPCODE
  4. ILLEGAL\_MEMORY\_ADDRESS\_BEYOND\_2048
* ConditionCode – “library” of the condition codes used to communicate the machine state to report to the user. The class is implemented as an Enum with JAVA.
  1. OVERFLOW
  2. UNDERFLOW
  3. DIVZERO
  4. EQUALORNOT
* BitWord – this class is a wrapper around the binary strings that the machine simulator is simulating. We added this layer of abstraction to more closely mimic the conceptual logic at the machine level.
* BitInstruction – this class extends BitWord to be specific to handle parsing of instructions.
* Register – this is a class that helps implement the needed functionality for the various identified registers within our requirements.
* MemoryLocation – this class represents an address location in memory and the value to be stored in binary. It also keeps track of the identified reserved locations:
  1. RESERVED\_ADDRESS\_TRAP
  2. RESERVED\_ADDRESS\_FAULT
  3. RESERVED\_ADDRESS\_TRAP\_PC
  4. RESERVED\_ADDRESS\_3
  5. RESERVED\_ADDRESS\_FAULT\_PC
  6. RESERVED\_ADDRESS\_5
  7. RESERVED\_ADDRESS\_TOGGLE\_INSTRUCTION

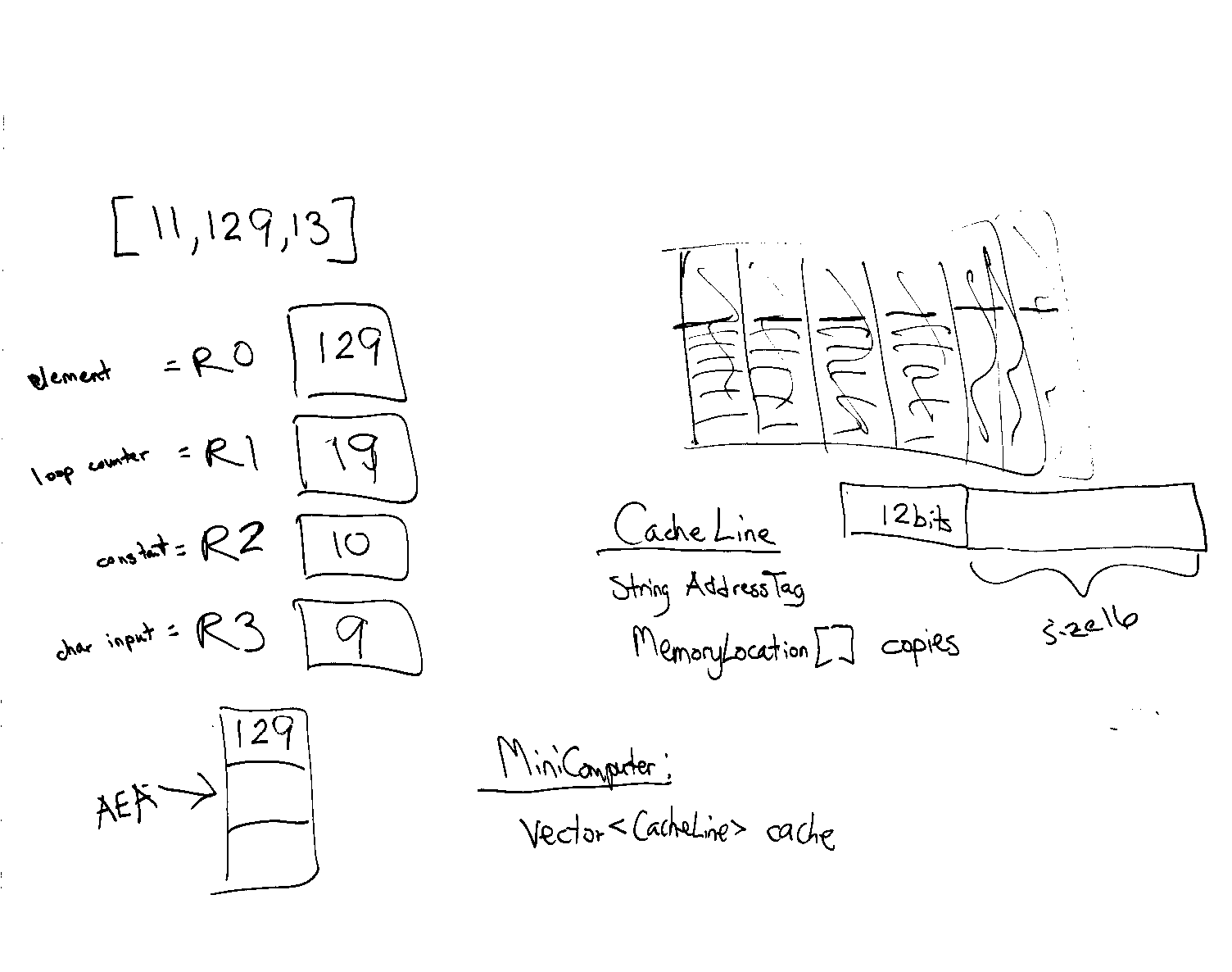
|  |  |  |
| --- | --- | --- |
|  | Design Requirements | Design Comments |
|  | Design and implement the basic machine architecture | Using an agile approach we will build in Four primary phases with potentially additional smaller sub-sprints. |
|  | Implement a simple memory | Address and Value stored in binary string to represent memory locations, map of memory locations to represent memory |
|  | Execute Load and Store instructions | Phase I |
|  | Build initial user interface to simulator | Phase I |
|  | Program Counter: address of next instruction to be executed. Note that 212 = 4096. | PC |
|  | Condition Code: set when arithmetic/logical operations are executed; it has four 1-bit elements: overflow, underflow, division by zero, equal-or-not. They may be referenced as cc(0), cc(1), cc(2), cc(3). Or by the names OVERFLOW, UNDERFLOW, DIVZERO, EQUALORNOT | CC |
|  | Instruction Register: holds the instruction to be executed | IR |
|  | Memory Address Register: holds the address of the word to be fetched from memory | MAR |
|  | Memory Buffer Register: holds the word just fetched from or stored into memory | MBR |
|  | Machine Status Register: certain bits record the status of the health of the machine | MSR |
|  | Machine Fault Register: contains the ID code if a machine fault after it occurs | MFR |
|  | Index Register: contains a base address that supports base register addressing of memory. | X1…X3 |
|  | Memory Address 0 | Reserved for the Trap instruction in Phase III. |
|  | Memory Address 1 | Reserved for a machine fault |
|  | Memory Address 2 | Store PC for Trap |
|  | Memory Address 3 | Not Used |
|  | Memory Address 4 | Store PC for Machine Fault |
|  | Memory Address 5 | Not Used |
|  | Stops the machine. | HLT - halt instruction |
|  | Traps to memory address 0, which contains the address of a table in memory. Stores the PC+1 in memory location 2. The table can have a maximum of 16 entries representing 16 routines for user-specified instructions stored elsewhere in memory. Trap code contains an index into the table, e.g. it takes values 0 – 15. When a TRAP instruction is executed, it goes to the routine whose address is in memory location 0, executes those instructions, and returns to the instruction stored in memory location 2. The PC+1 of the TRAP instruction is stored in memory location 2. | TRP - Trap Instruction |
|  | The subroutine is available for further functionality to be implemented, however they are configured currently to not execute any specific code. |
|  | Opcode | 6 characters - Specifies one of 64 possible instructions; |
|  | IX | 2 characters - Specifies one of three index registers; may be referred to by X1 – X3. 0 value indicates no indexing. |
|  | R | 2 characters - Specifies one of four general purpose registers; may be referred to by R0 – R3 |
|  | I | 1 character - If I =1, specifies indirect addressing; otherwise, no indirect addressing. |
|  | Address | 7 characters - Specifies one of 63 locations |
|  | Load Register From Memory | Load Register From Memory, r = 0..3 r <- c(EA) |
|  | Store Register To Memory | Store Register To Memory, r = 0..3 EA <- c(r) |
|  | Load Register with Address | Load Register with Address, r = 0..3 r <- EA |
|  | Load Index Register from Memory | Load Index Register from Memory, x = 1..3 Xx <- c(EA) |
|  | Store Index Register to Memory | Store Index Register to Memory. X = 1..3 EA <- c(X0) |
|  | JZ r, x, address[,I] | Jump If Zero: |
| If c(r) = 0, then PC <- EA or c(EA), if I bit set; |
| Else PC <- PC+1 |
|  | JNE r, x, address[,I] | Jump If Not Equal: |
| If c(r) != 0, then PC <-- EA or c(EA) , if I bit set; |
| Else PC <- PC + 1 |
|  | JCC cc, x, address[,I] | Jump If Condition Code |
| cc replaces r for this instruction |
| cc takes values 0, 1, 2, 3 as above and specifies the bit in the Condition Code Register to check; |
| If cc bit = 1, PC <- EA or c(EA), if I bit set; |
| Else PC <- PC + 1 |
|  | JMA x, address[,I] | Unconditional Jump To Address |
| PC <- EA, if I bit not set; PC <- c(EA), if I bit set |
| Note: r is ignored in this instruction |
|  | JSR x, address[,I] | Jump and Save Return Address: |
| R3 <- PC+1; |
| PC <- EA or PC <- c(EA), if I bit set |
| R0 should contain pointer to arguments |
| Argument list should end with –17777 value |
|  | RFS Immed | Return From Subroutine w/ return code as Immed portion (optional) stored in the instruction’s address field. |
| R0 <- Immed; PC <- c(R3) |
| IX, I fields are ignored. |
|  | SOB r, x, address[,I] | Subtract One and Branch. R = 0..3 |
| r <- c(r) – 1 |
| If c(r) > 0, PC <- EA; but PC <- c(EA), if I bit set; |
| Else PC <- PC + 1 |
|  | JGE r,x, address[,I] | Jump Greater Than or Equal To: |
|  | If c(r) >= 0, then PC <- EA or c(EA) , if I bit set; |
|  | Else PC <- PC + 1 |
|  | AMR r, x, address[,I] | Add Memory To Register, r = 0..3 |
|  | r c(r) + c(EA) |
|  | SMR r, x, address[,I] | Subtract Memory From Register, r = 0..3 |
|  | r c(r) – c(EA) |
|  | AIR r, immed | Add Immediate to Register, r = 0..3 |
|  | r  c(r) + Immed |
|  | Note: |
|  | 1. if Immed = 0, does nothing |
|  | 2. if c(r) = 0, loads r with Immed |
|  | IX and I are ignored in this instruction |
|  | SIR r, immed | Subtract Immediate from Register, r = 0..3 |
|  | r  c(r) - Immed |
|  | Note: |
|  | 1. if Immed = 0, does nothing |
|  | 2. if c(r) = 0, loads r1 with –(Immed) |
|  | IX and I are ignored in this instruction |
|  | MLT rx,ry | Multiply Register by Register |
|  | rx, rx+1 <- c(rx) \* c(ry) |
|  | rx must be 0 or 2 |
|  | ry must be 0 or 2 |
|  | rx contains the high order bits, rx+1 contains the low order bits of the result |
|  | Set OVERFLOW flag, if overflow |
|  | Note: Multiply actually allows for ”OVERFLOW” by holding a 32 bit answer within 2 registers. |
|  | DVD rx,ry | Divide Register by Register |
|  | rx, rx+1 <- c(rx)/ c(ry) |
|  | rx must be 0 or 2 |
|  | rx contains the quotient; rx+1 contains the remainder |
|  | ry must be 0 or 2 |
|  | If c(ry) = 0, set cc(3) to 1 (set DIVZERO flag) |
|  | Note: Divide catches the remainders and stores them in the next register |
|  | TRR rx, ry | Test the Equality of Register and Register |
|  | If c(rx) = c(ry), set cc(4)  1; else, cc(4)  0 |
|  | AND rx, ry | Logical And of Register and Register |
|  | c(rx)  c(rx) AND c(ry) |
|  | ORR rx, ry | Logical Or of Register and Register |
|  | c(rx)  c(rx) OR c(ry) |
|  | NOT rx | Logical Not of Register To Register |
|  | C(rx)  NOT c(rx) |
|  | SRC r, count, L/R, A/L | Shift Register by Count |
|  | c(r) is shifted left (L/R =1) or right (L/R = 0) either logically (A/L = 1) or arithmetically (A/L = 0) |
|  | XX, XXX are ignored |
|  | Count = 0…15 |
|  | If Count = 0, no shift occurs |
|  | RRC r, count, L/R, A/L | Rotate Register by Count |
|  | c(r) is rotated left (L/R = 1) or right (L/R =0) either logically (A/L =1) |
|  | XX, XXX is ignored |
|  | Count = 0…15 |
|  | If Count = 0, no rotate occurs |
|  | IN r, devid | Input Character To Register from Device, r = 0..3 |
|  | OUT r, devid | Output Character to Device from Register, r = 0..3 |
|  | CHK r, devid | Check Device Status to Register, r = 0..3 |
|  | c(r) <- device status |
|  | FADD fr, x, address[,I] | Floating Add Memory To Register |
|  | c(fr)  c(fr) + c(EA) |
|  | c(fr)  c(fr) + c(c(EA)), if I bit set |
|  | fr must be 0 or 1. |
|  | OVERFLOW may be set |
|  | FSUB fr, x, address[,I] | Floating Subtract Memory From Register |
|  | c(fr)  c(fr) - c(EA) |
|  | c(fr)  c(fr) - c(c(EA)), if I bit set |
|  | fr must be 0 or 1 |
|  | UNDERFLOW may be set |
|  | VADD fr, x, address[,I] | Vector Add |
|  | fr contains the length of the vectors |
|  | c(EA) or c(c(EA)), if I bit set, is address of first vector |
|  | c(EA+1) or c(c(EA+1)), if I bit set, is address of the second vector |
|  | Let V1 be vector at address; Let V2 be vector at address+1 |
|  | Then, V1[i] = V1[i]+ V2[i], i = 1, c(fr). |
|  | VSUB fr, x, address[,I] | Vector Subtract |
|  | fr contains the length of the vectors |
|  | c(EA) or c(c(EA)), if I bit set is address of first vector |
|  | c(EA+1) or c(c(EA+1)), if I bit set is address of the second vector |
|  | Let V1 be vector at address; Let V2 be vector at address+1 |
|  | Then, V1[i] = V1[i] - V2[i], i = 1, c(fr). |
|  | CNVRT r, x, address[,I] | Convert to Fixed/FloatingPoint: |
|  | If F = 0, convert c(EA) to a fixed point number and store in r. |
|  | If F = 1, convert c(EA) to a floating point number and store in FR0. |
|  | The r register contains the value of F before the instruction is executed. |
|  |  |
|  | LDFR fr, x, address [,i] | Load Floating Register From Memory, fr = 0..1 |
|  | fr  c(EA), c(EA+1) |
|  | fr <- c(c(EA), c(EA)+1), if I bit set |
|  | STFR fr, x, address [,i] | Store Floating Register To Memory, fr = 0..1 |
|  | EA, EA+1  c(fr) |
|  | c(EA), c(EA)+1 <- c(fr), if I-bit set |
|  | ROM Loader should read the boot program from the ROM and place it into memory in a location you designate. The ROM Loader then transfers control to the program which executes until completion or error | Our initial implementation writes the boot program instructions to memory directly, then executes thosee instructions that load/store values to/from memory. |
|  | If your program completes normally, it returns to the boot program to read the next program (at this point your simulation should stop with PC having the value of the first address of the boot program). Returning to the boot program means that it prompts the user to either run the currently loaded program again or to load a new program and run it. | Phase I |
|  | Upon powering up your system, all elements of memory should be set to zero | Our implementation in JAVA implicitly defaults memory values to be all zeroes, until it is set to something else by the boot program, other program, or console instruction. |
|  | Your memory simulation should accept an address from the MAR on one cycle. It should then accept a value in the MBR to be stored in memory on the next cycle or place a value in the MBR that is read from memory on the next cycle. | Phase I |
|  | The UI should include both the console plus some additional capabilities to support the debugging of your simulator | Our initial implementation displays register values, and memory address and memory values to check to see if instructions are completed correctly. Messages are also printed to the IDE console. |
|  | The UI should display registers which are not programmer-accessible, but are required for correct operation of the computer (and your simulator). | The primary registers displayed in the GUI will be the General Register (R) and the Index Register (X). The back-end will be implemented first before adding extra Registers to the GUI. |
|  | Operators Console should display all registers | Phase II+ implementation |
|  | Operators console should show display for machine status and condition registers | Phase II+ implementation |
|  | Operators console should show an IPL button (to start the simulation) | Phase I |
|  | Operators console should show switches (simulated as buttons) to load data into registers, to select displays, and to initiate certain conditions in the machine | Our initial implementation has switches that the operator can use to input load/store instructions. |
|  | Operators console should show Current Memory Address | Phase I |
|  | Operators Console should show switches that the user can inform the program. | Phase II+ implementation |
|  | Operators Console should show Run button | Hold off for later phase |
|  | Operators Console should show Halt button/switch | Hold off for later phase |
|  | Operators Console should show Single Step switch | Phase I |

## I/O Design

* I/O will only work with digits and the carriage return (Enter) and newline characters in Part II (all other characters are filtered out by the GUI), to prevent confusion on whether a bit string is representing a character in ASCII or a number in binary.
  + Enter key is represented by the bit string 1111111111111111 (16 ones)—since the highest expected number is 2^16 (1000000000000000 in binary), this sequence should never show up when inputting a number, unlike Enter’s ASCII character (13 in decimal).
  + Newline is represented by the bit string 1010101010101010
  + The only 2 valid Device Ids are 0 (keyboard) and 1 (printer)
* New Device Id 3 for reading characters from the file prgm-2-input.txt
  + Assumptions for prgm-2-input.txt:
    - There are exactly 6 sentences, each of which ends with a period
    - Periods are only used to indicate the ends of sentences
    - There is only one space after a period before the first word of the next sentence begins
    - Each sentence has no more than 30 words.
    - We will assume each sentence will have no more than 143 characters.
* New Device Id 4 for reading in ASCII characters from user (still one character at a time, with Enter key indicating the user is done inputting)
  + Device Id 2 is only for reading in numbers from the user, one digit at a time
    - The special bit strings for representing Enter and Newline apply only to Device Id 2
* Machine Faults are tested for.
  + Illegal Memory Address to Reserved Locations
  + Illegal TRAP code
    - When trap instruction is called, the program traps to reserved memory address 0, which contains address of another location that holds the instruction to run
    - Trap instruction contains a 4-bit trap code which lets the program know which of (up to) 16 subroutine to run.
    - Because trap interrupts the normal program flow, it stores the next PC value to reserved memory address 2
    - After trap instruction runs, it retrieves the saved PC and continues running the program from there
    - There is the ability to add functionality; this TRAP code implementation calls a subroutine, which currently is not robustly engineered.
  + Illegal Operation Code
  + Illegal Memory Address beyond 2048 (memory installed)

## User Input Design:

* User Entry: – using available Registers to calculate any address value
  + Element - this is our final output
  + Loop counter –
  + Constant – for base 10 multiplication we store a constant of 10 in R2
  + Char input – takes character typed by user and appends it to the previous entries to build out a base 10 number to be saved into memory
* User Entry process:
  + From the console the character input is temporarily stored in R3
  + The old value in Element R0 is multiplied by the constant in R2 of 10, effectively shifting the previous charters entered to the left
  + The output of that calculation is then added to the user entry
  + This process will gradually allow the user to enter any number from the console using base-10 counting which people are accustom to.



Cache Design Sketch

User Input

User Input Example:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Step | 1 | 2 | 2.1 | 2.2 | 3 | 3.1 | 3.2 | 4 |
| User Entry | 1 | 2 | - | - | 9 | - | - | <enter> |
| R0 | 1 | 1 | 1 | 12 | 12 | 12 | 129 | 129 |
| R1 | 1 | 1 | 10 | 10 | 10 | 120 | 120 | - |
| R2 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | - |
| R3 | 1 | 2 | 2 | 2 | 9 | 9 | 9 | - |
| Displayed | 1 |  |  | 12 |  |  | 129 | 129 |

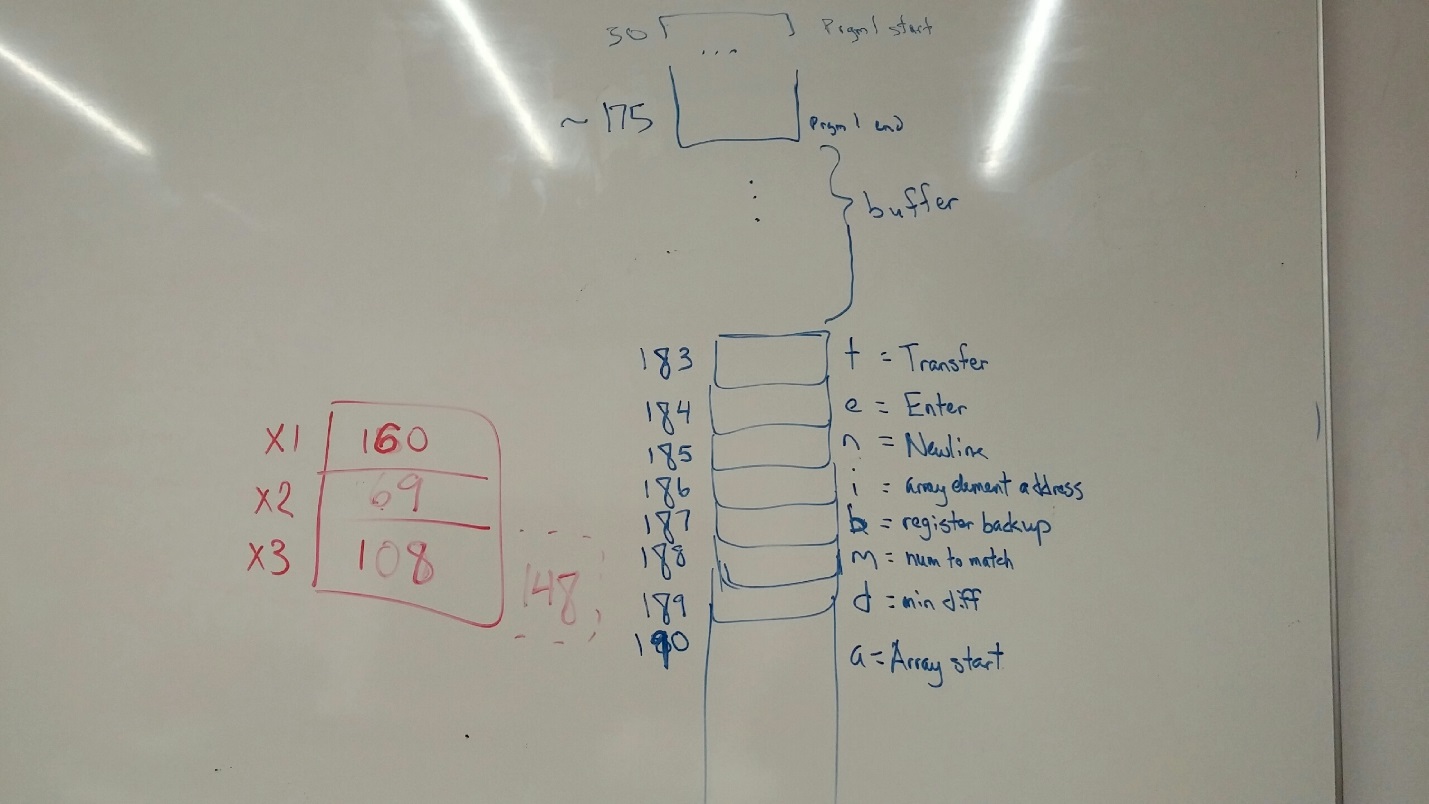
The table illustrates an example for how the number “129” would be entered by the user and how the simple machine would enable this entry.

## Major Memory & Cache Structure Design:

The main memory decisions:

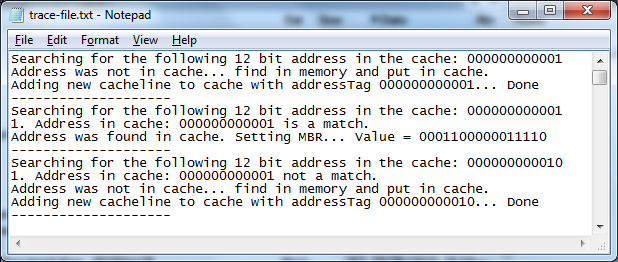
* Numbers stored in registers and memory are unsigned integers (i.e. all non-negative)
  + If subtract would result in a negative difference, UNDERFLOW flag is set to true, and the register contents is not written over.
  + Likewise, if divide results in a DIVZERO, the register contents will not be written over (i.e. will still contain the dividend and divisor instead of a gibberish quotient and remainder)

Certain areas of memory have been designated for specific purposes. The first 30 memory addresses are reserved as internal memory addresses (e.g. TRAP opcode, and machine fault were stipulated in the requirements, and 6-30 are a buffer before the boot program). The program will be allocated addresses 30-175. Addressing for data begins at address 183. Due to the limitations of our minicomputer we decided to maintain a simple memory architecture at this point in time.



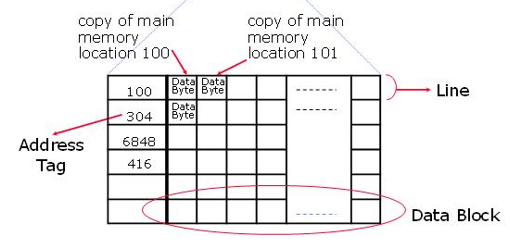
The main cache decisions:

* The mini-computer has a fully associated, unified cache
* The Cache will match on addressTag to determine a hit or miss. It will return the value on a hit, and will fetch from memory the appropriate address block to cache.
* Cache Writing
  + Write-through is easier to debug
  + Write no-allocate is simpler to implement than write allocate
  + We will start the cache simple in initial implementation and evolve in later releases.
* Cache Reading
  + The Cache sues a FIFO algorithm to replace cache lines, which is first-in first-out for up to 16 address tags.
* Cache reading and writing will write to a trace file called trace-file.txt to show caching is working



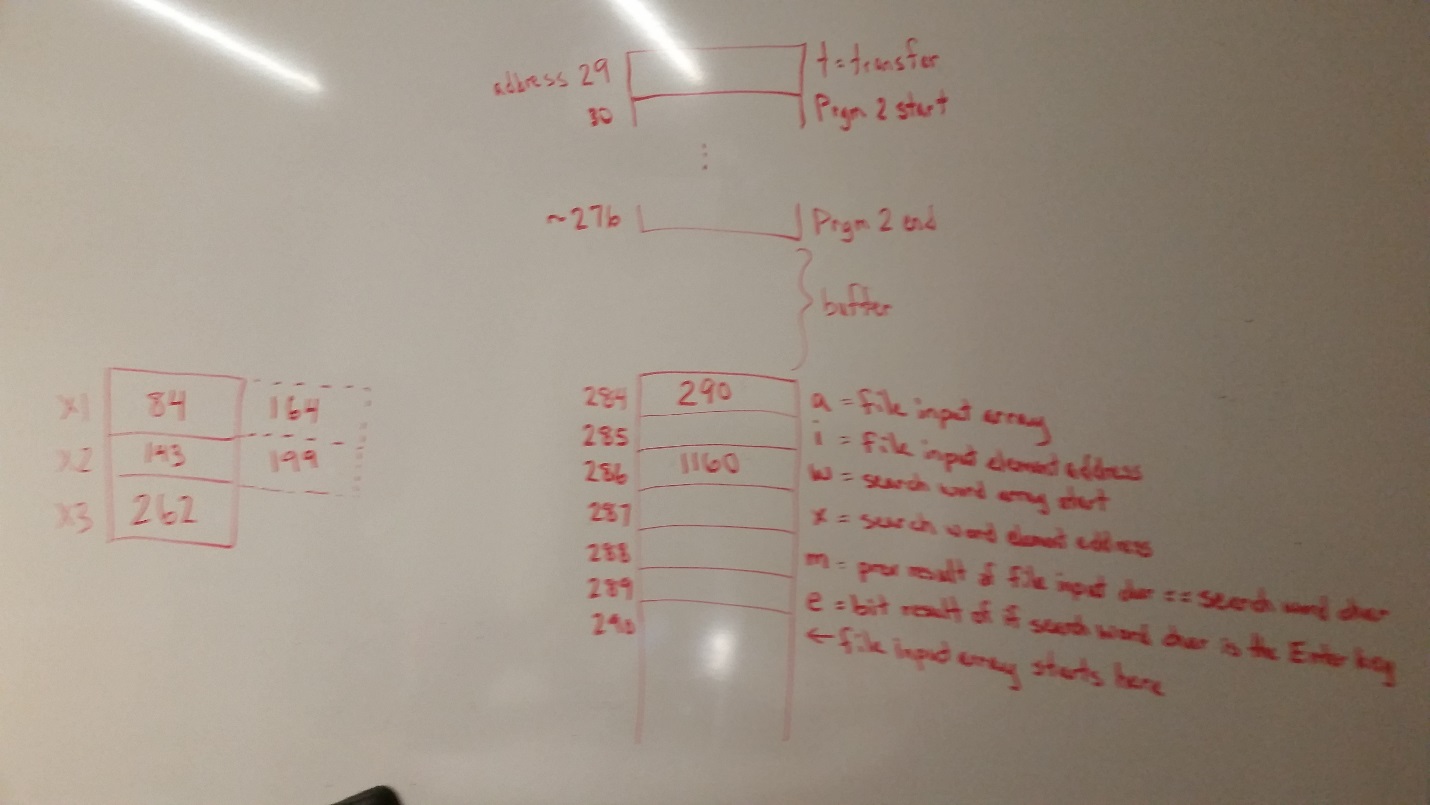
* 12 bits for the Address Tag – the first 12 bits of the memory address
* 4 bits for the blocks near that tagged address, for a size of 2^4=16 cache Data Block, where the index of an element is the offset from the address tag

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |  |  |  |  |



This simple cache design generates time-space efficiency savings by reducing the time “off-chip” and keeping as many of the calculations “on chip” as possible. By mirroring entire chunks from memory and caching it, we are hoping to increase our ‘hit rate’ of staying “on chip”. Since our cache block is 16 bits, and our ‘write to memory’ algorithm increments locally when writing to memory (with exceptions for jumps, etc) we will expect “roughly” that 16 sequential write-to-memory commands will reside in contiguous allocation within the data block on one address tag within the cache. Thus, we hope to achieve adequate ‘hit rate’ within our cache, while deploying a simple cache architecture.

Program 2:



* The additions to the mini-computer design that were implemented during program 2 were tested over the displayed address domains.
* The only punctuation allowed are periods “.” (NO commas, semi-colon, apostrophe, hyphen, etc.)
* The checking for words within the program is NOT case sensitive. All alphabetic characters will be converted to uppercase. This is to simplify our matching algorithm.
* The checking will not match for number values, it is only alphabet sensitive.

## Enhanced Scheduling Design:

This mini-computer contains a simple branch prediction and speculative execution implementation with trap-error handling.

**Branch Prediction and Speculative Execution Design Notes:**

* **What is it/How do we interpret this?**
  + Predict whether the branch (jump instruction) will be taken
    - Prediction bits are in the Branch Prediction Buffer (BPB)—just do a Map inside MiniComputer.
      * Tag = lower 8 bits of the instruction (i.e. the address to jump to)
      * Value = prediction bit (true/false)
  + Execute the instructions for the predicted path, but don’t commit them yet
    - Add parameter to instruction functions: Boolean executeSpeculatively
      * If executeSpeculatively is true, store result(s) to Reorder Buffer (ROB) instead of the intended destination
  + If prediction is correct, just get the results from the ROB and store to the intended destination instead of executing
  + If prediction is wrong, flip the prediction bit, flush the results, and execute the correct instructions as normal
* **How to determine the prediction?**
  + *Initially assume branch will NOT be taken (bit  = 0)*
    - Most instructions are for if statements, which are usually used for edge cases
    - Loop instruction would be wrong at most twice (entering and exiting the loop)
  + Update prediction based on whether that prediction was correct
* **How many instructions to execute based on the prediction?**
  + Can’t really figure out when instructions for branch taken ends and normal instructions begins
  + *Keep it simple and just execute only the 1st instruction of the path predicted*
* **When does branch prediction happen?**
  + When the jump instruction is fetched
    - JZ, JNE, JCC, SOB, JGE
* **Where are the results of the speculatively executed instructions stored?**
  + Reorder Buffer (ROB)
    - Instruction Type = branch, store (to memory), or register
    - Value = the result of the instruction execution
    - Destination = where the result will be stored (register number or memory address)
      * Branch instructions  PC
      * Store (to memory) instruction  memory
      * Register instruction  register (R0-3, X1-3)
    - Ready = ready to be stored  to destination or not
* **How do we make sure singleStep() doesn’t try to re-execute the speculatively executed instruction?**
  + *Either add a flag getResultFromROB, and check the flag inside singleStep() before trying to execute that instruction*
    - Simpler than the option below
    - If flag is true, store the value to the destination
      * This will also handle nested jump instructions
  + Or inside the instruction method, after determining the PC, actually commit then update the PC to what it would’ve done after executing
    - What if there are nested jump instructions?

