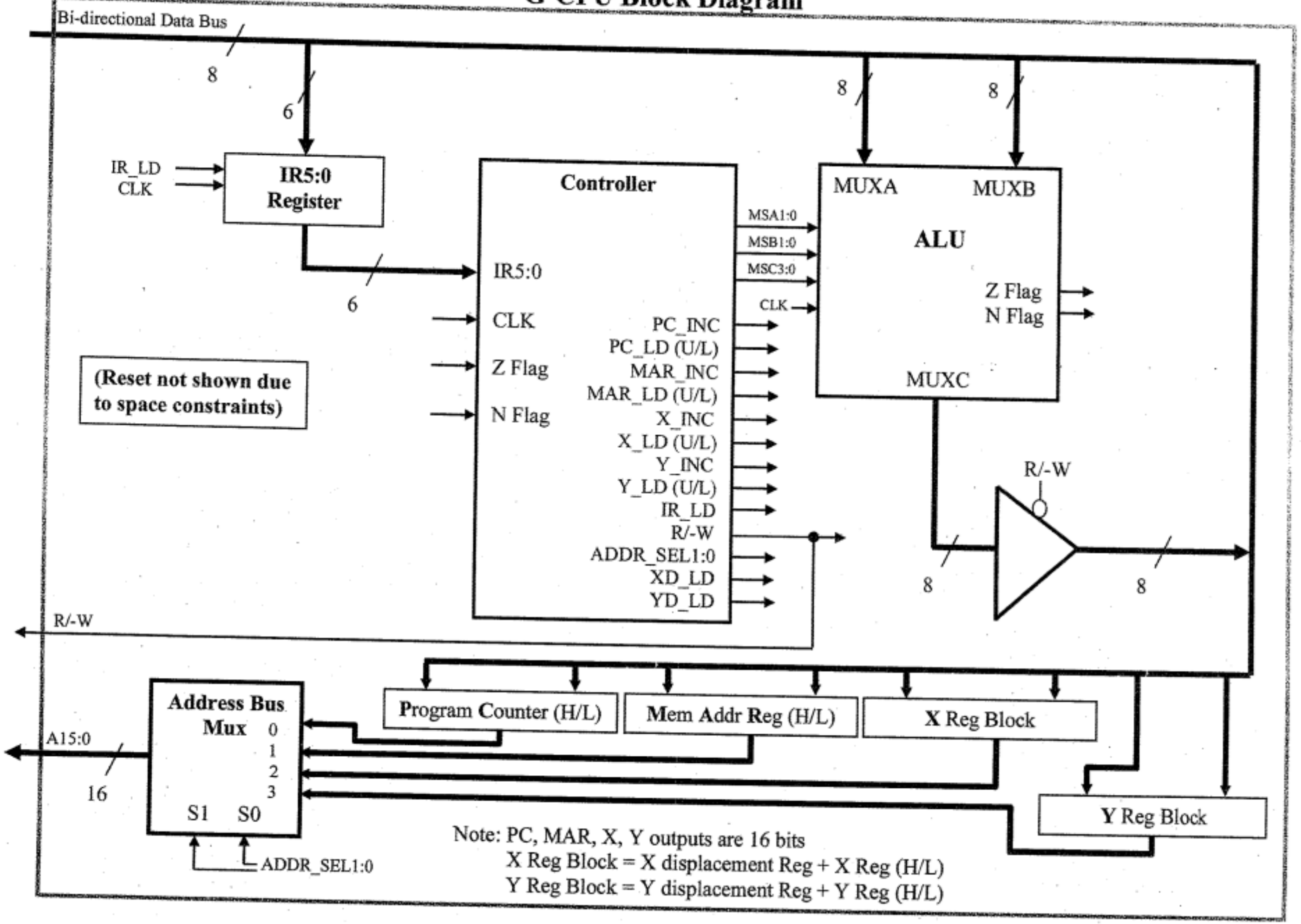
EEL3701 GCPU Review

by Matthew Benda

GCPU Hardware Design

Drs. Gugel and Schwartz
21-Nov-16

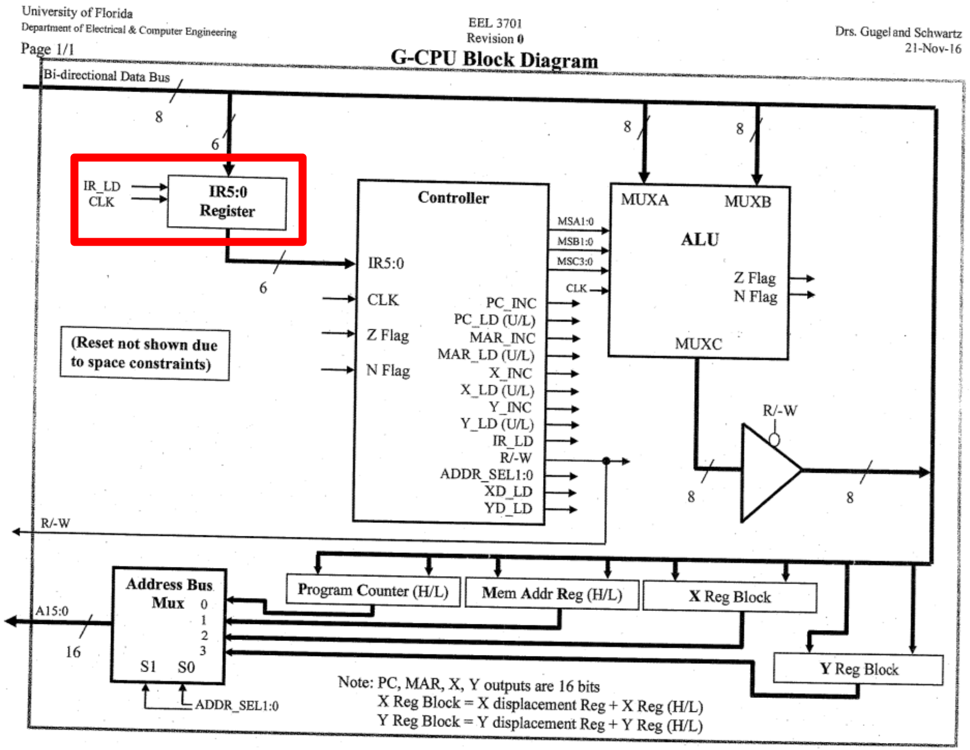
G-CPU Block Diagram



Note: PC, MAR, X, Y outputs are 16 bits
X Reg Block = X displacement Reg + X Reg (H/L)
Y Reg Block = Y displacement Reg + Y Reg (H/L)

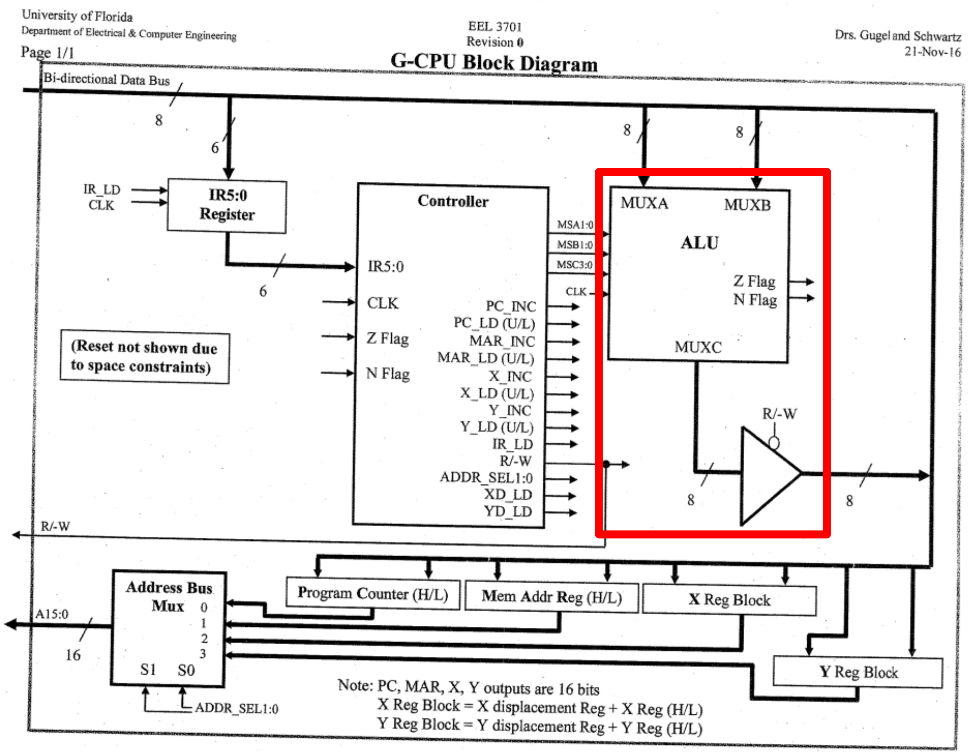
Instruction Register

- 6 bits wide -> up to 64 instructions
- Stores input on a rising edge if IR_LD is true
- Only loads in state 0 - all other states are decode/execute states
- Basically same design as Lab 6



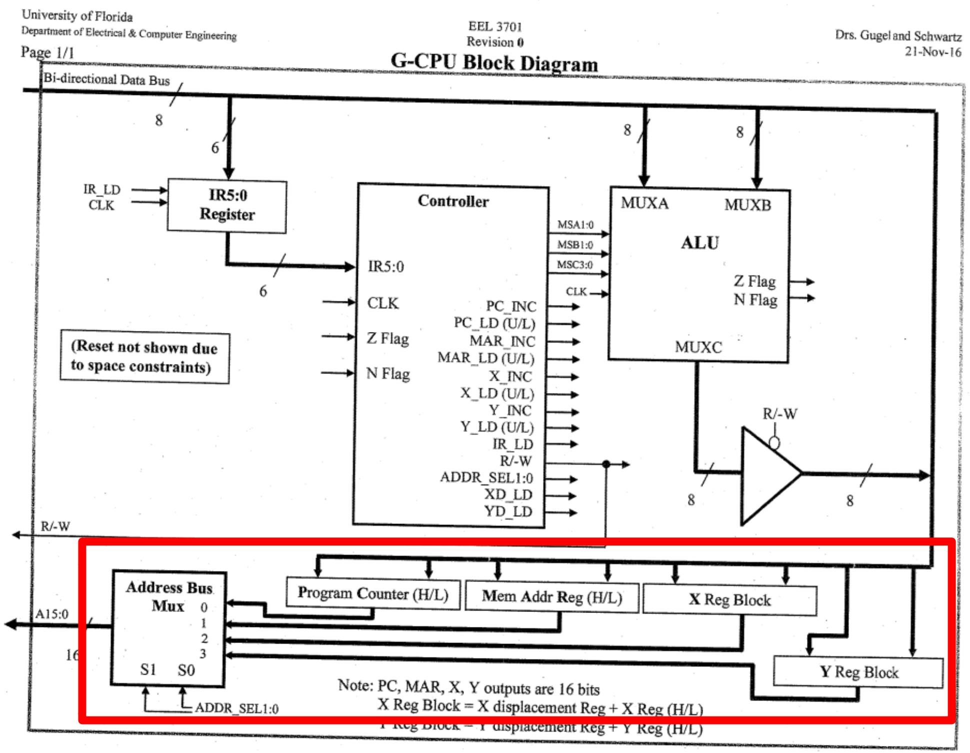
ALU

- Up to 16 functions (up from 8 in Lab 6)
- MSA/MSB work the same
- Z Flag- true if REGA == 0
- N Flag - true if REGA < 0 (when interpreted as a 2s complement number)
- Basically same design as Lab 4
- Output bus is connected to data bus with a tri-state buffer (since it is sometimes driven by memory not the CPU).



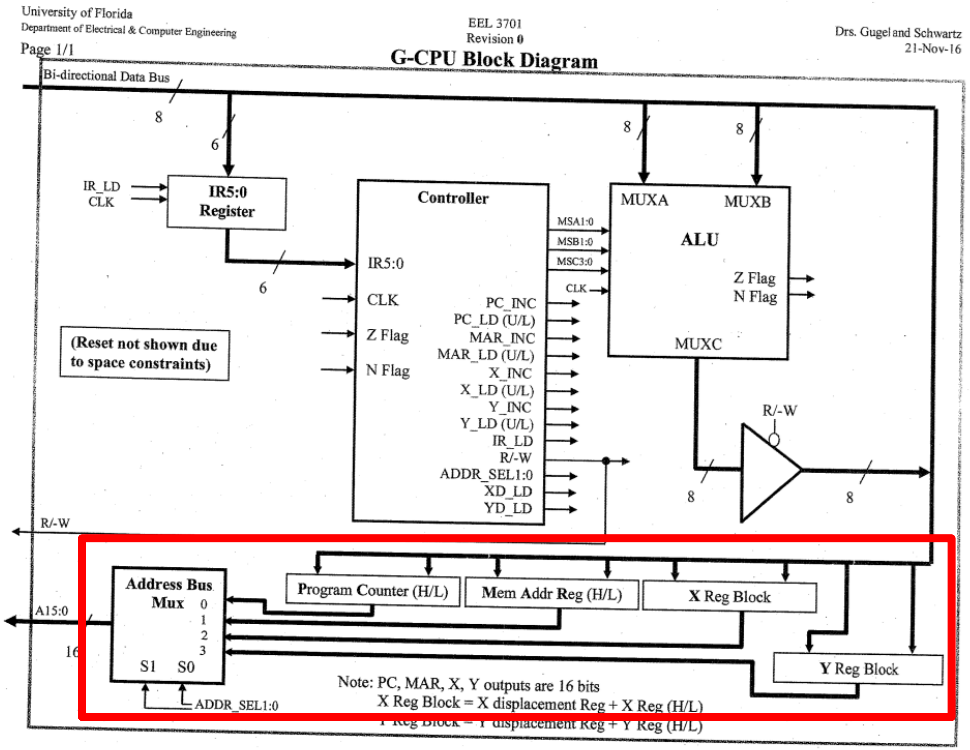
Address Control Unit

- Only 4 possible sources for an address
  - PC, MAR, X, Y
- PC used to keep track of program execution
  - Only loaded by branch instructions or Incremented by all instructions
- X,Y used as pointers to data- treat them as variables
  - Used for indexed addressing mode instructions



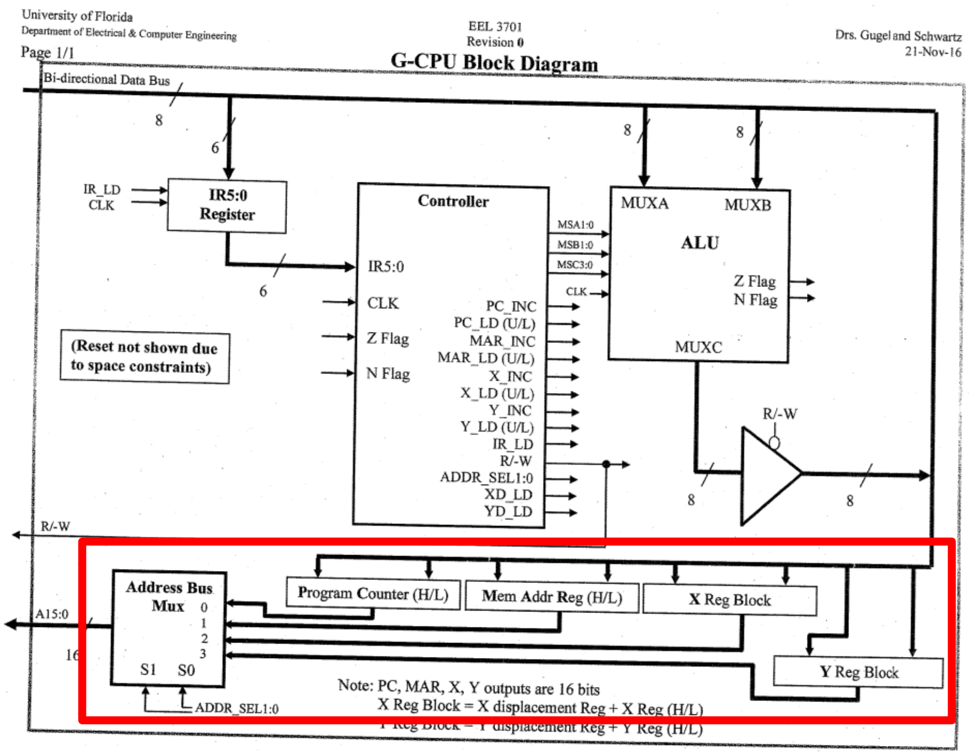
Address Control Unit

- MAR used for “random access” instructions
  - Extended addressing mode instructions use the MAR to load in whatever address they reference.
- Since we can’t change the PC, X, or Y to get to random addresses, we use the MAR.
- Ex: LDAA $1370 loads the address into the MAR so that it can be accessed



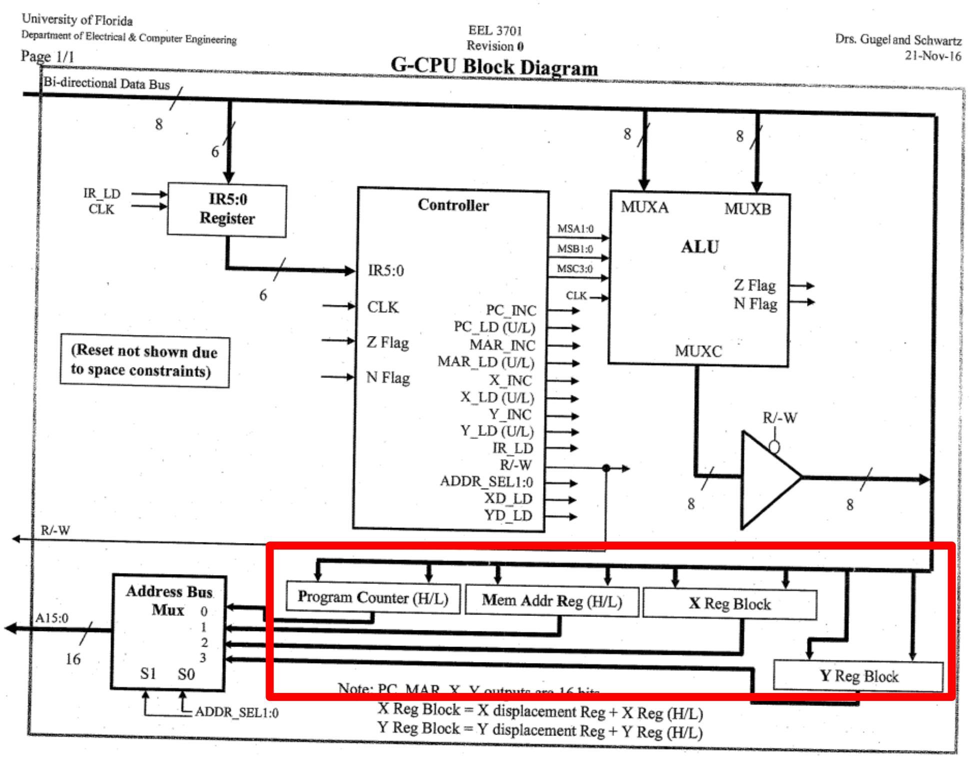
Address Control Unit

- Address Bus Mux
  - Actually selects what source we are using for an instruction
  - Extended will use the MAR
  - Indexed will use X, Y
  - Absolute/Immediate will use PC
  - Inherent doesn't use memory -> default to PC



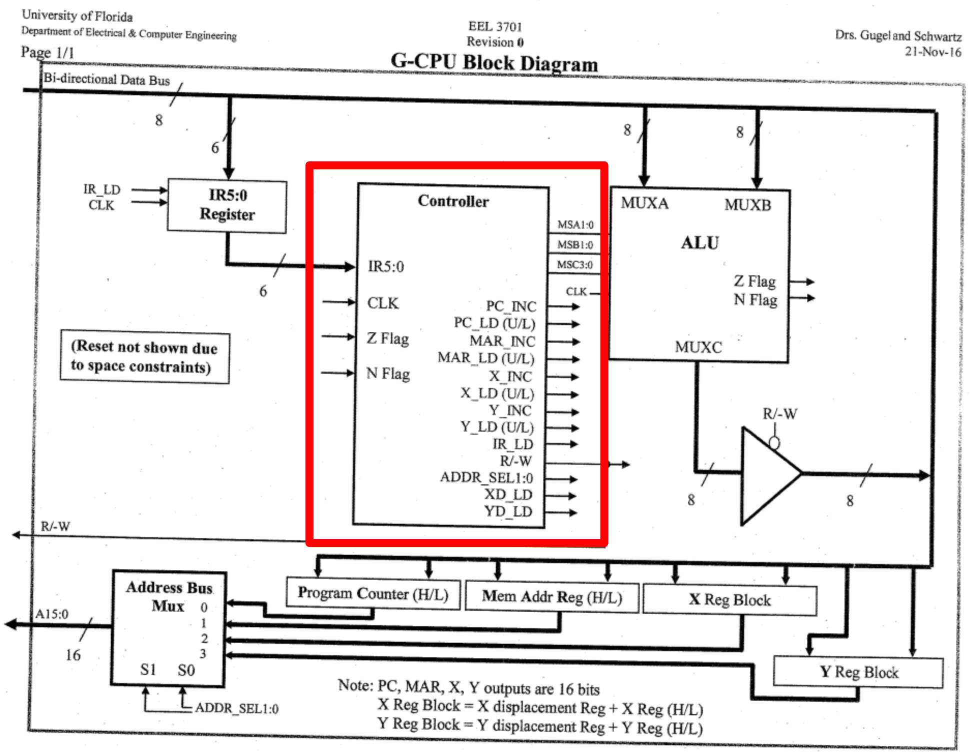
Address Control Unit

- Address Source Registers Structure
  - Since our data is 8 bits, and the address bus is 16 bits wide, we actually need 2x8-bit registers for each source
  - Each source is divided into a upper/lower (U/L) register
  - Each sub-register is loaded independently
  - X,Y also have a displacement register (used to load the displacement for indexed instructions)



Controller

- Generates the necessary control signals to execute each instruction.
  - INC signals for all address sources
  - LD_U/LD_L for all address sources
  - IR_LD to load IR
  - R/~W to control direction of data on data bus
  - Address source select signals
  - X,Y displacement load signals
  - ALU controls



Instruction Set

Instruction Anatomy

- Every instruction has at least one byte for its associated machine codes, but there can be up to three depending on the instruction.
- GCPU document has a key for what each machine code placeholder represents
- Ex:
    - LDX #data has machine codes 08 ii jj
    - ii is the low byte of the data
    - jj is the high byte of the data
    - LDX #$1370 has machine codes 08 70 13

- mm — 8-bit immediate data value
- ii — Low-order byte of a 16-bit data
- jj — High-order byte of a 16-bit data
- ll — Low-order byte of a 16-bit address
- hh — High-order byte of a 16-bit address
- dd — 8-bit displacement value
- bb — Low-order byte of a 16-bit address for a branch instruction

Addressing Modes & Effective Addresses

- Five addressing modes:
  - Inherent Addressing
  - Immediate Addressing
  - Extended Addressing
  - Indexed Addressing
  - Absolute Addressing
- The GCPU document tells you what mode each instruction uses!
- Effective address = address of location data is fetched from or sent to
  - Some instructions don’t access the memory -> no fetch/send -> no EA

Inherent Addressing

- Used by “ALU-level” instructions
- SUM_BA, SHFA_L, etc.
- No effective address for these instructions (no memory access)

Immediate Addressing

- Used by instructions that put a given value into a register
- Examples:
  - LDAA #$37
  - LDX #$3701
- These instruction use the **exact** data that is provided in the instruction
- The data is embedded in the machine codes, so it **immediately** follows the instruction opcode.
- EA = Address of the instruction itself + 1 (next address after opcode).
  - Need to assemble the program to find this, can’t find it otherwise

Extended Addressing

- Used by instructions that fetch/store data to a particular address
- Examples:
  - LDX $1000
  - LDAA $FF
- Notice no ‘#’ before the number in the above examples.
- These instruction **go to the given address** and either store or fetch data from there.
  - We cannot assume anything about the data there unless we put it there ourselves.
- EA = the address given in the instruction
  - $1000, $00FF for the above examples

Indexed Addressing

- Used by instructions that fetch/store data to a particular address **relative to the address in X,Y**
- Examples:
  - LDAA 0,X
  - STAB 3,Y
- Like extended addressing, these instructions **go to an address** and either store or fetch data from there.
- EA = X/Y + displacement value
- Commonly used to point to tables/arrays (as you have seen/will see in Lab 7)

Absolute Addressing

- Used by branch instructions
- BEQ $08, BP LOOP
- Evaluates the branch condition, it is is met, it loads PC_L with the given address
  - Does not affect PC_H -> there is a limited range for branches
- EA = none! No data is moved with this instruction, like inherent addressing.

Labels and Assembler Directives

Labels!

- Labels are placed all the way to the left of instructions or assembler directives
- Used to reference the **address** of the line at which they are placed
- Very useful for organizing and writing clean assembly code

- To be explored later

Assembler Directives

- EQU
    - Similar to #define in C-like programming languages
    - Equates a string to some value
    - COUNT EQU 15 - When assembling the code, replace every instance of count with 15 before converting to machine codes
    - Does not actually get put anywhere in memory since it is not code- just a tool for the assember (YOU!) when writing complex programs.

Assembler Directives

- ORG
  - Tells the assembler at what addresses to put the code that follows it
  - Used to establish some frame of reference for where t put the program
  - Example:

```
ORG $0000
LDAA #3
```

  - In this case, ORG tells the assembler that the LDAA instruction should be placed at address 0.
  - Useful for if we want to write programs at different addresses, or to initialize data somewhere separate from programs.
  - Does not actually get put anywhere in memory since it is not code- just a tool for the assember (YOU!) when writing complex programs.

Assembler Directives

- DC.B
  - **D**efines a **c**onstant **b**yte at some address
  - Used to initialize some data in memory
  - Example:

```
        ORG $1FF0
DATA    DC.B 1,2,3,4,5,6
```

  - In this case, DATA is a label for the address $1FF0. Starting there, the values 1,2,3,4,5,6 are initialized in memory
  - Useful for if we want to provide a program with some starting data
  - Can be used for either ROM or RAM address spaces

Assembler Directives

- DS.B
  - **D**efines a **s**torage **b**yte at some address
  - Example:

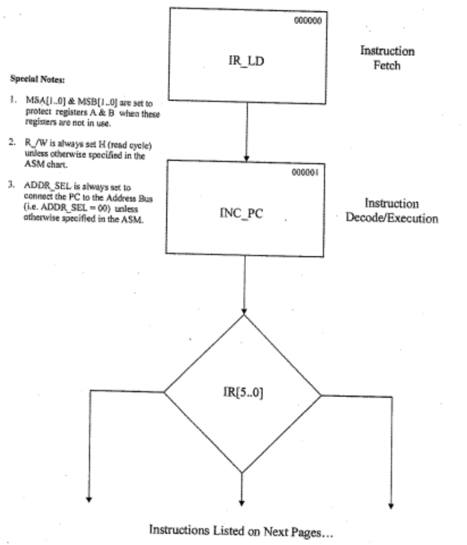
```
        ORG $1FF0
DATA    DS.B 3
DATA2   DS.B 1
```

  - In this case, DATA is a label for the address $1FF0. Since DS.B allocates 3 bytes, DATA2 is a label for address $1FF3.
  - Useful for allocating memory for variables that we will generate during program execution (loop counters, running totals, etc)
  - Should be used in RAM address range, since we will want to write to it (that is the whole point)

Lets Look At Some Flowcharts

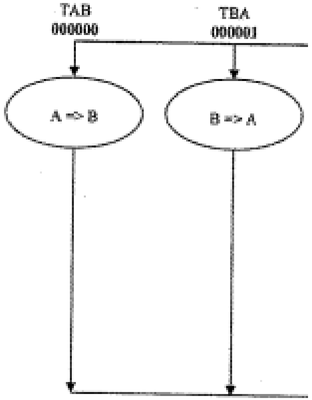
In the beginning...

- Every instruction starts with an instruction fetch and a decode/execute state.
- Depending on the instruction, more execute states will be used



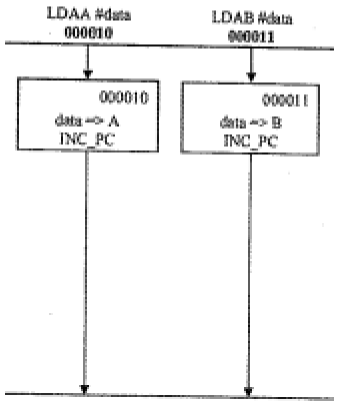
Inherent Addressing Example

- These instructions just use the one base decode/execute state
- No memory access requires so no need for extra states.
- We’ve already seen that computations in the ALU only require 1 cycles, so this works.



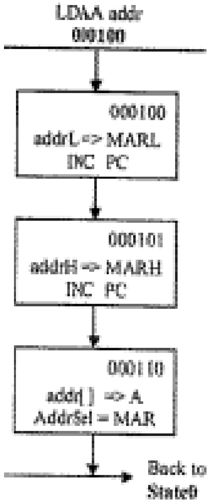
Immediate Addressing Example

- These instruction use one extra execute state
  - Have to increment PC to point to the immediate value
  - Then one cycle to store the input



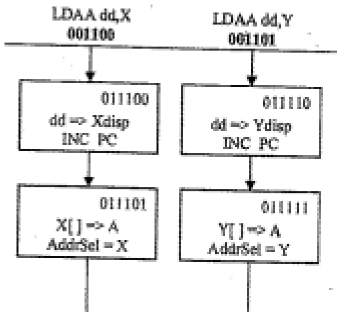
Extended Addressing Example

- This instruction uses three extra execute states
  - One to load MARL
  - One to load MARL (MAR now points to the given address)
  - One to load the value and store.



Indexed Addressing Example

- These instructions uses two extra execute states
  - One to load the displacement value
  - One to load the value and store.



Absolute Addressing Example

- These instructions uses one extra execute state
  - One mealy decision based on the condition
  - Then it either continues by incrementing the PC, or it loads the branch address to the PC.