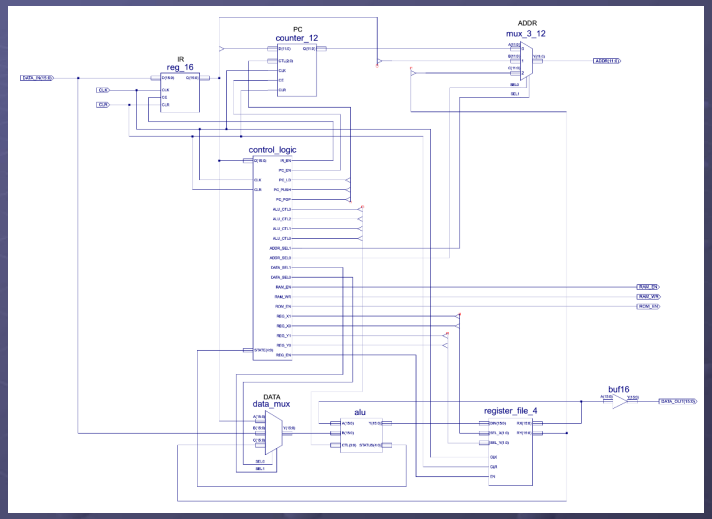
Systems and Devices 1
Lec 6a : CPU

Before we get started ...

- We now have a basic computer, but it has a number of limitations:
  - Functions : to simplify the design and construction of our software and to improve memory efficiency, the processor needs to support subroutines i.e. function calls.
  - Data types : depending on the application domain a processor will need to support different data types and structures e.g. small numbers, large numbers, arrays, lists, images etc.
  - Memory : need more than ACC + 256 storage locations.
- Note, solutions based around self modifying code are not a good idea, so time for an upgrade :)

SimpleCPU_v1d

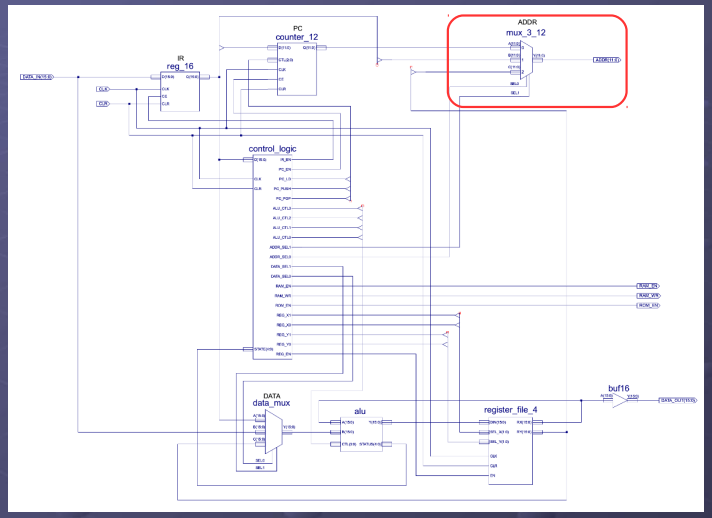


- Q: can you spot the differences :)

Improvement 1: Memory

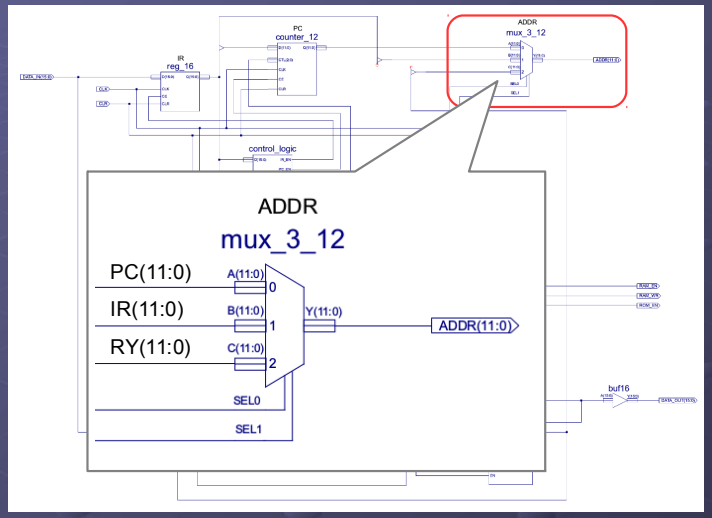
- Improvement 1 : increase external memory size
  - SimpleCPU_v1a could only address 256 x 16 bit memory locations.
    - LCD version of HELLO WORLD needed 221 instructions, doesn't leave a lot of room for other functionality.
  - SimpleCPU_v1d memory size increased to 4096 x 16 bit memory locations.
- Q :what architectural features do we need to change to implement this improvement?
  - Communication links: internal / external
  - Hardware : registers, multiplexers, logic …
  - How will this affect instruction format?

SimpleCPU_v1d



- A : internal/external address bus and multiplexer

SimpleCPU_v1d



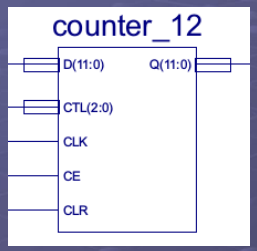
- A : internal/external address bus and multiplexer

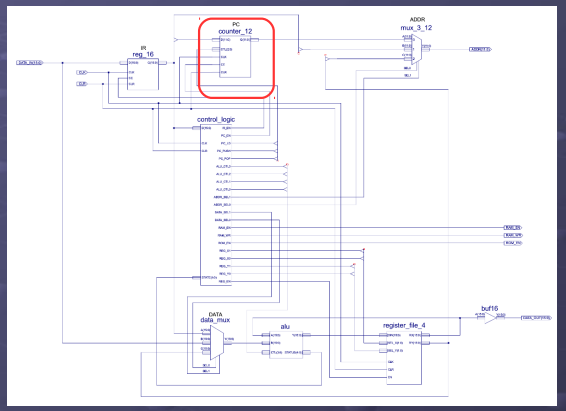
Improvement 1: Memory

	Opcode				Not used				Operand							
LOAD 0x9A	0	1	0	0	0	0	0	0	A	A	A	A	A	A	A	A
LOAD 0x19A	0	1	0	0	A	A	A	A	A	A	A	A	A	A	A	A

- Address bus increased to 12bits.
  - Q : why 4096 memory locations, why not 65536 (64K)?
  - A : allows the same instruction format to be used i.e. backwards compatibility, new CPU can run old code.
    - To address 65536 memory locations requires 16bit address i.e. no space for opcode, would need 20bit instructions
  - Address bus multiplexer hardware increased to 12 bits.
    - 33% increase in hardware requirements.

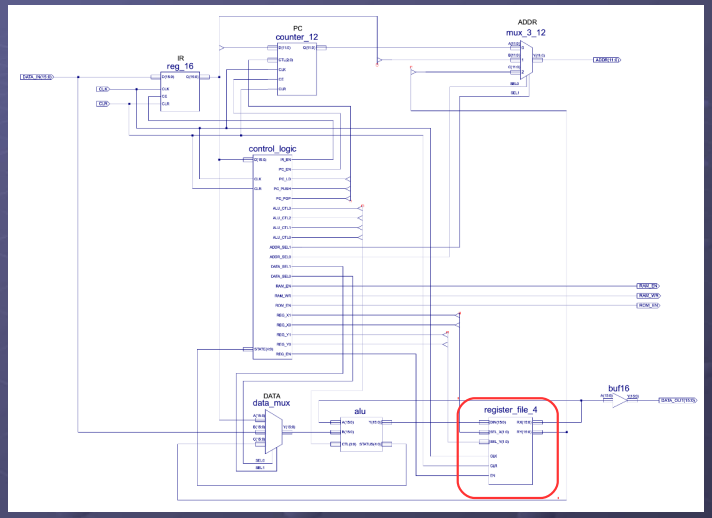
Improvement 1: Memory





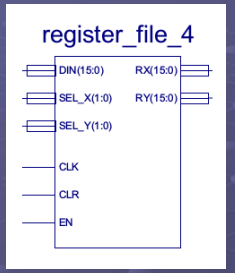
- 12bit program counter
  - Register, MUX, Adder
- Note, components are tightly coupled, small changes tend to ripple through the complete architecture, resulting in significant increases in hardware.

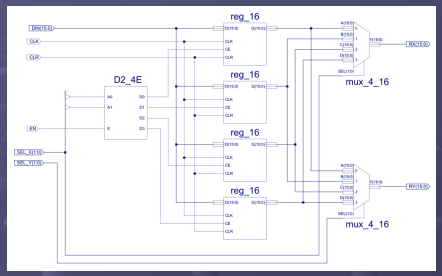
Improvement 1: Memory



- Register file : a bank of general purpose registers.

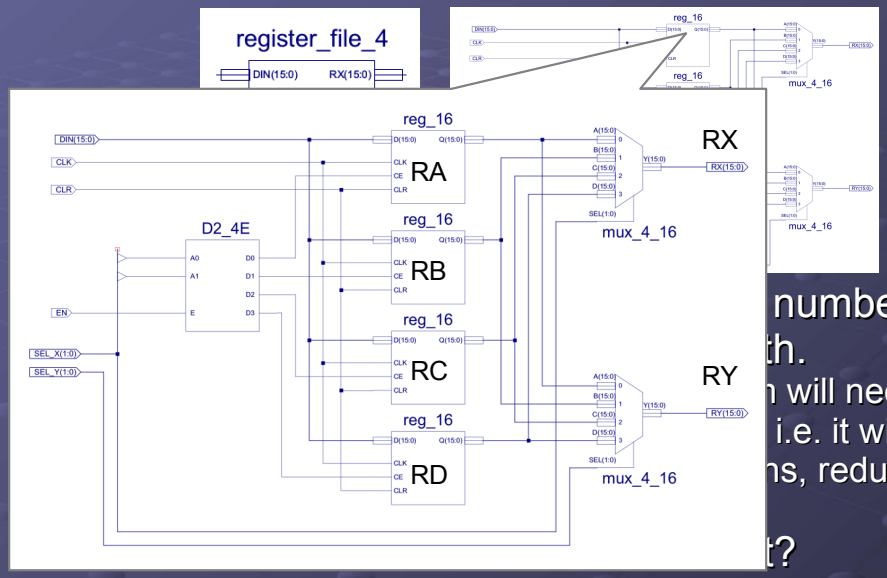
Improvement 1: Register file





- Increase internal “memory” size i.e. the number of general purpose registers and their width.
  - With only one internal register (ACC), a program will need to store all of its variables in external memory i.e. it will need to perform a lot of LOAD/STORE instructions, reducing processing performance.
- Q : how will this affect instruction format?

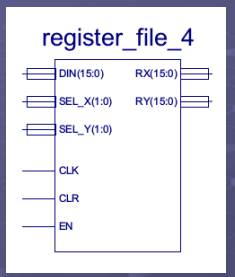
Improvement 1: Register file

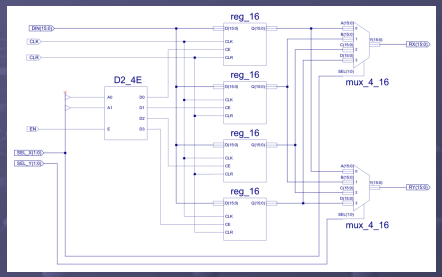


number of
th.
will need to
i.e. it will need
ns, reducing

t?

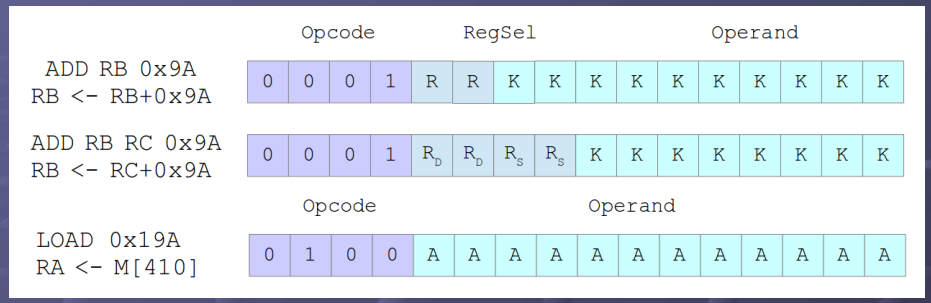
Improvement 1: Register file





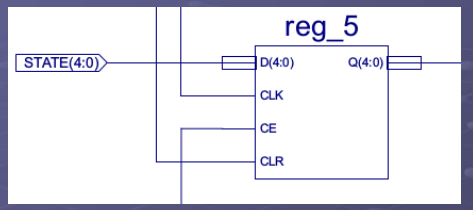
- Increase internal “memory” size i.e. the number of General Purpose Registers (GPR) and their width.
  - With one internal register (ACC), a program will need to store all of its variables in external memory i.e. it will need to perform a lot of LOAD/STORE instructions, reducing processing performance.
- Q : how will this affect instruction format?

Improvement 1: Register file



- A : operand source and result destination no longer implicit, therefore, need extra bit fields:
  - Register select bit-fields : R=Destination & Source R_D=Destination, R_S=Source
  - Choices to be made : implicit (fixed), two or three operand formats, should all instruction be supported?

Improvement 1: Register file



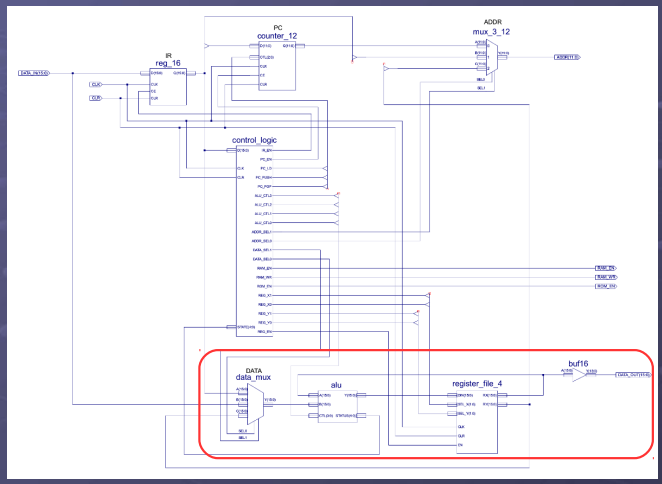
B0 : ZERO
B1 : CARRY
B2 : OVERFLOW
B3 : POSITIVE
B4 : NEGATIVE

- A : also affects the control instruction group i.e. previously conditional JUMP instructions were based on the ACC, but now which register should be tested?
  - Again that knock on, ripple affect :)
  - Solution : use a status register. This records the outcome of the last arithmetic or logical function performed.
  - To improve functionality we can add new status bits.
    - Will look at this again in the next lecture.

Improvement 2: Data types

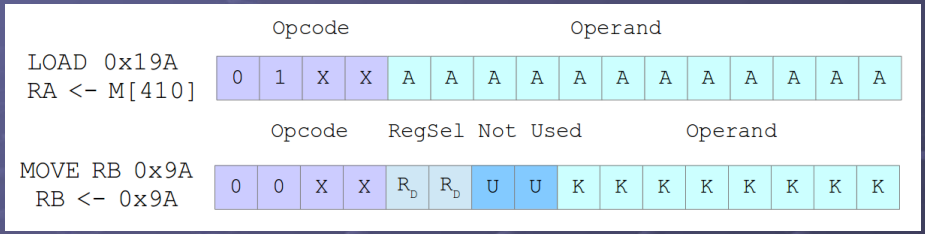
- Improvement 2 : increase data width
  - SimpleCPU_v1a could process 8bit data types
    - Unsigned : 0 to 255, Signed : -128 to +127
    - For some applications this more than sufficient, however, images commonly contain thousands of pixels, we need to process larger numbers.
      - RGB values: 255+255+255 = 765
  - SimpleCPU_v1d updated to support 16bit data types
    - Unsigned : 0 to 65535, Signed : -32768 to +32767
- Q : what architectural features do we need to change to implement this improvement?

Improvement 2: Data types



- A : all data buses and processing hardware increased to 16 bits e.g. data MUX, ALU, registers etc.

Improvement 2: Data types



- This improvement has a significant impact on hardware requirements e.g. the move from an 8bit ripple adder to 16bits will double hardware requirements.
- Q : can you “initialise” a register with an 16bit value?
  - What are the different data sources? What are their data widths? What do we do if they are different?

Improvement 2: Data types

```
LOAD RA 0x64     RA <- M[100]                          16bit
MOVE RA 0xFF     RA <- (IR(7))^8 || IR(7:0)            8bit sign-extended
AND  RA 0xFF     RA <- RA and ((0)^8 || IR(7:0))       8bit unsigned
```

- Q : can you “initialise” a register with an 16bit value?
  A : maybe, common problem in most CPUs e.g. 64bit machines with 32bit instructions.
- Solution : create a new instruction to load upper byte, or move (shift) data within a register.
  - Note, processors are optimised for the most commonly used tasks. Infrequent operations can be implement using two (or more) simple machine-level instructions without having a significant impact on performance.

Quick Quizzz

```
MOVE RA 0xAA     RA <- (IR(7))^8 || IR(7:0)
SL0 RA           RA <- RA(14:0) || 0
OR RA 0xAA       RA <- RA or ((0)^8 || IR(7:0))
```

- Quick Quizzz
  - Using these instructions write a program to load the 16bit value 0xAAAA into a register.
  - Hint, will need ten instructions.
    - Load high byte, Move byte, Load low byte
  - What instructions would you need to add to perform this operation in two instructions?

Improvement 3: Addressing modes

```
ADD RA RB        RA <- RA + RB
LOAD RA (RB)     RA <- M[RB]
```

- With the addition of more general purpose registers we can now implement two new addressing modes.
  - Register : all operands are in registers.
  - Register indirect : external memory address stored in general purpose register rather than IR.
- Q : what bit fields are needed to define these instructions? Do we need to define a “number” i.e. a constant or an address?

Improvement 3: Addressing modes

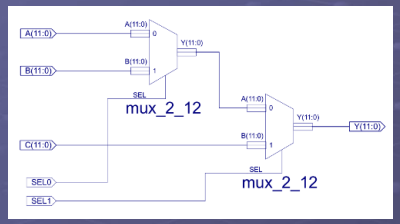
	Opcode				RegSel				Not Used				Opcode			
MOVE RA RB	1	1	1	1	R_D	R_D	R_S	R_S	0	0	0	0	0	0	0	1
LOAD RA (RB)	1	1	1	1	R_D	R_D	R_S	R_S	0	0	0	0	0	0	1	0
STORE RA (RB)	1	1	1	1	R_D	R_D	R_S	R_S	0	0	0	0	0	0	1	1

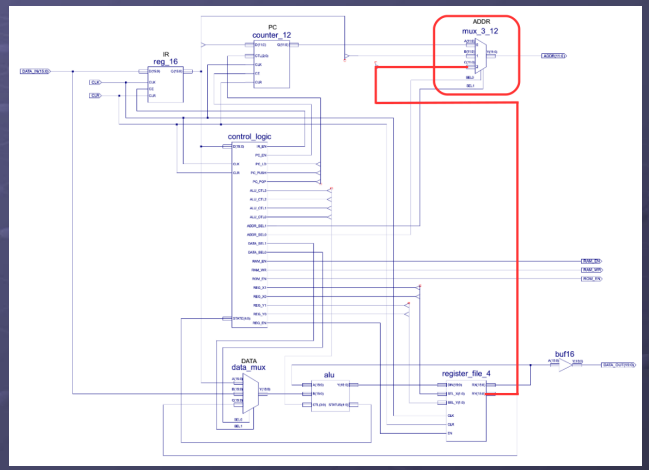
- Q : Do we need to define a “number” ?
- A : No, all operands are stored in registers. Therefore, we can use the same “opcode” for each instruction, then use the lower nibble to identify the type of register based instruction to perform.

Improvement 3: Addressing modes

- Note, register based addressing modes have two advantages
  - Reduces the need to store variables in memory i.e. removes LOAD/STORE instructions, increasing performance.
    - Less instruction in a program = quicker execution time.
    - Most processors have 8 to 32 registers (or more)
  - Removes the need to use self-modifying code as we can use registers as pointers i.e. to store the address in memory of the data we want to use, rather than using the absolute addressing mode.
- Q: what architectural features do we need to change to implement this improvement?

Improvement 3: Addressing modes





- A : need to add an additional input to the address MUX
  - PC
  - IR(11:0)
  - RY(11:0)

Improvement 3: Addressing modes

	Opcode				RegSel				Operand							
LOAD RX 100(RY) RX <- M[RY+100]	1	1	1	0	RX	RX	RY	RY	K	K	K	K	K	K	K	K

	Opcode				RegSel				Not used				Opcode			
ADD RX RY RX <- RX + RY	1	1	1	1	RX	RX	RY	RY	U	U	U	U	0	1	0	0
ADD RX RY RZ RX <- RY + RZ	1	1	1	1	RX	RX	RY	RY	RZ	RZ	U	U	0	1	0	1

- Q: what architectural features do we need to change?
- A: dependents on functionality, addressing modes used e.g. displacement, 2 / 3 operand formats, orthogonal instruction sets?
  - An instruction-set architecture where all instruction types can use all addressing modes and storage elements.

Improvement 4: Macros/Subroutines

- To simplify coding the processor needs to support the software programming constructs used by the programmer i.e. functions.
- This can be implemented using either:
  - Macros : pre-processor used as a text replacement tool, allowing the programmer to define constants and more “powerful” macro-instructions from multiple machine-level instructions
  - Subroutines : commonly referred to as routines, procedures, functions, or methods. This is a block of code that can be written once and then “called” from anywhere in a program.

Macros

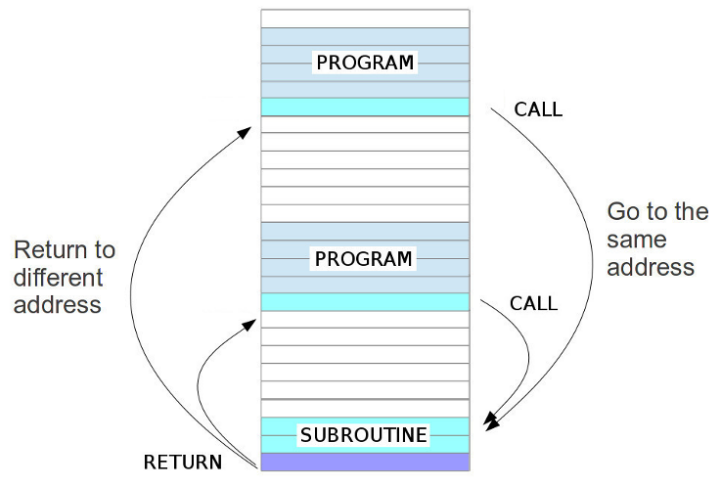
```
define( clr,`move 0x00')
define( set,`move 0xFF')

define( ora,`and eval($1 ^ 255)
             add $1')
```

- Example: simpleCPU_v1a macro
  - Pre-processing cut and paste.
  - Reduces coding time, but does not improve memory efficiency i.e. same code replicated each time a macro is used.

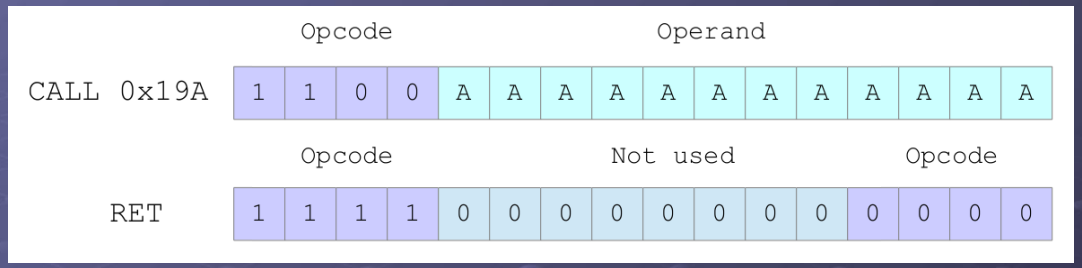
```
User Program
clr
ora 0xAA
move 0x0F
ora 0xAA

Final Code
move 0x00     (00)
and  0x55     (00)
add  0xAA     (AA)
move 0x0F     (0F)
and  0x55     (05)
add  0xAA     (AF)
```


Subroutines

- Write once use many.
- Two new instructions required
  - CALL
  - RET



Subroutines

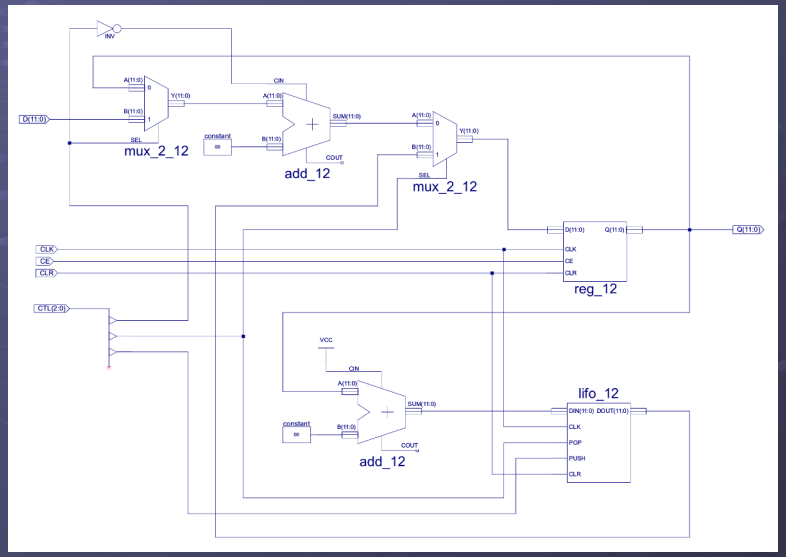


- CALL instruction comparable to JUMP, but also pushes PC+1 onto subroutine return stack.
- RET instruction pops return address off stack and updated PC

- CALL AA
  - `M[SP] <- PC+1`
  - `SP    <- SP+1`
  - `PC    <- AA`
- RET
  - `SP    <- SP-1`
  - `PC    <- M[SP]`

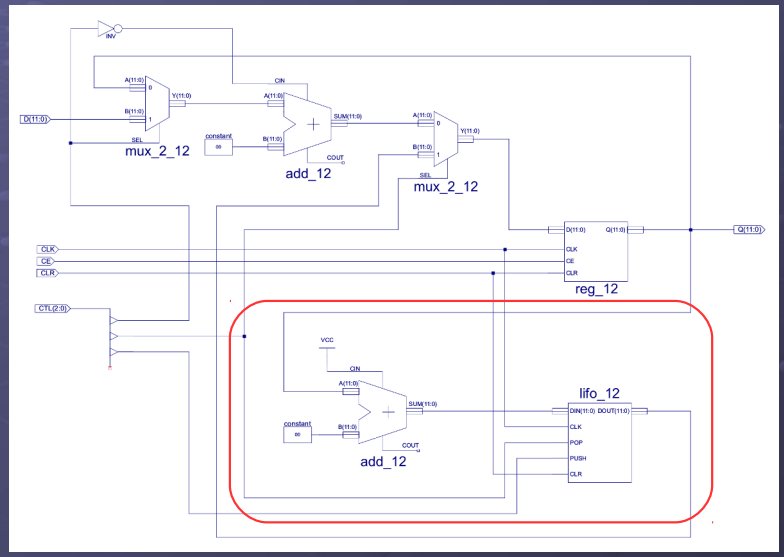
Subroutines

- New PC additional:
  - LIFO buffer
  - Adder (inc)



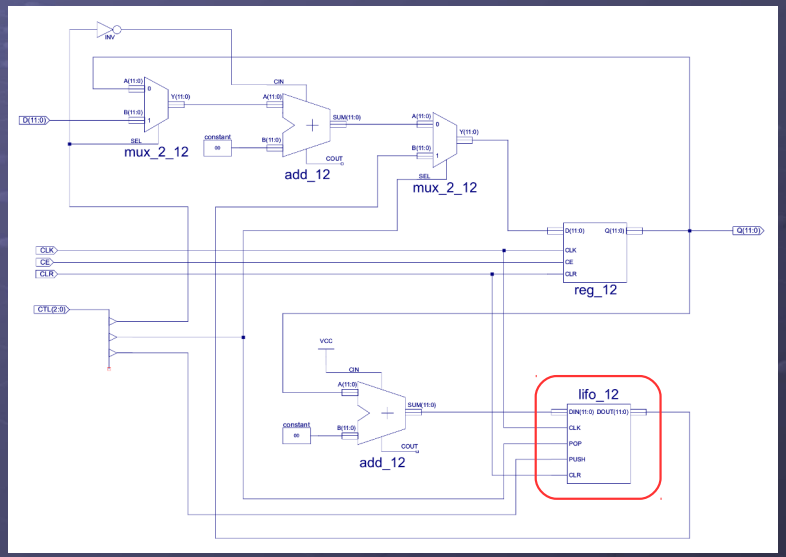
Subroutines

- New PC additional:
  - LIFO buffer
  - Adder (inc)



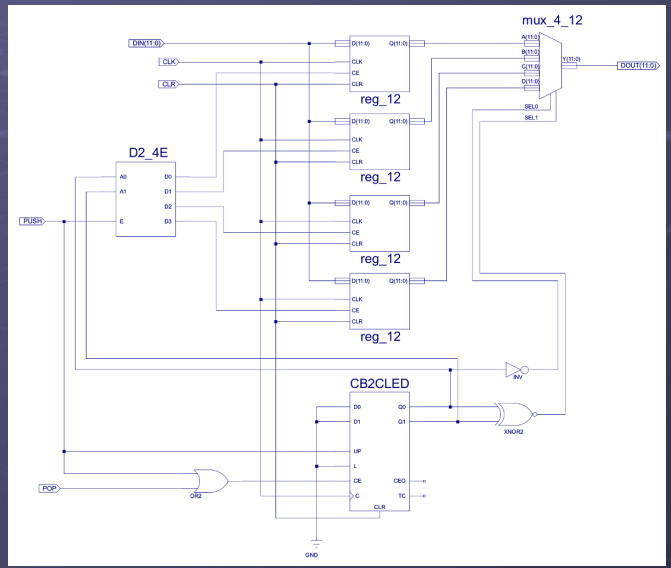
Subroutines

- New PC additional:
  - LIFO buffer
  - Adder (inc)



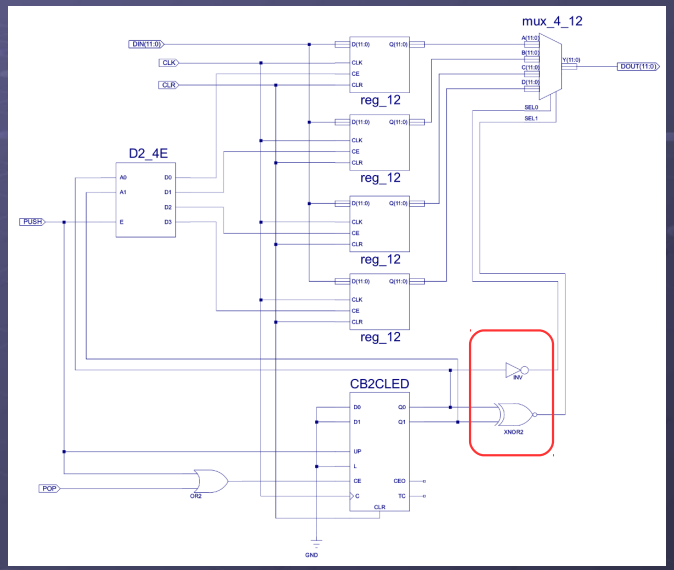
Subroutines

- LIFO 12
  - Four 12bit registers
  - Stack counter
  - Decrementer
  - 2-4 onehot decoder
  - Four input 8bit MUX

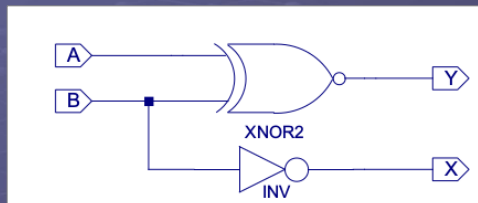


Subroutines

- LIFO 12
  - Four 12bit registers
  - Stack counter
  - Decrementer
  - 2-4 onehot decoder
  - Four input 8bit MUX



Quick Quizzz

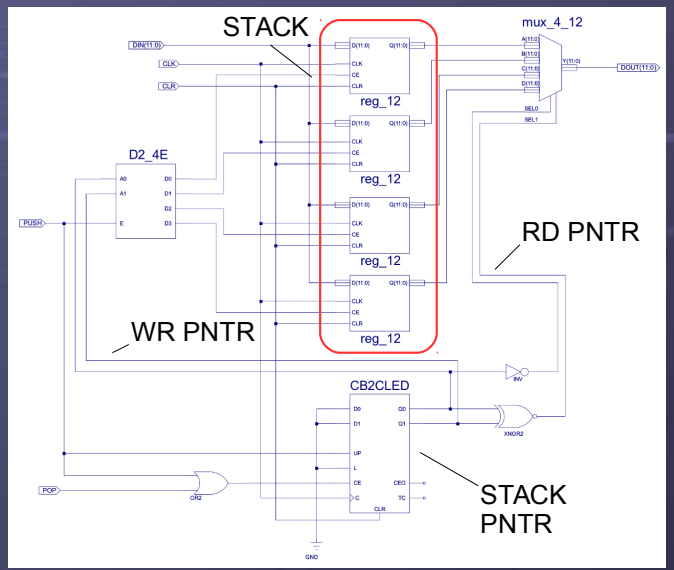


A	B	Y	X
0	0	1	1
0	1	0	0
1	0	0	1
1	1	1	0

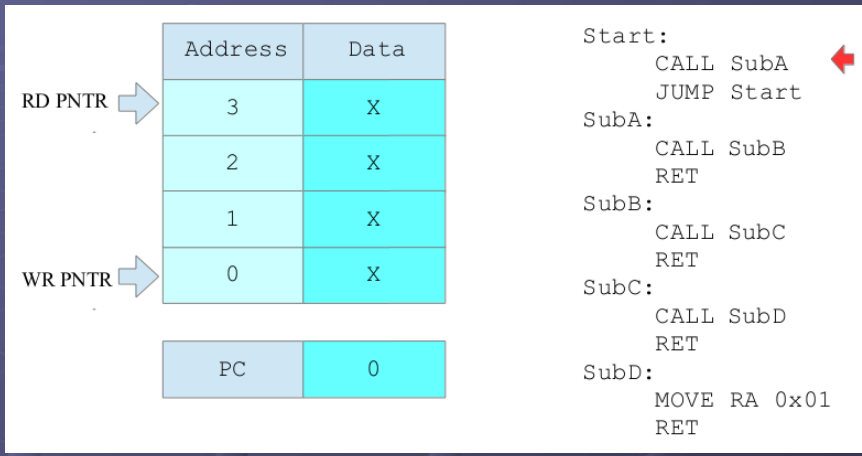
- Q: what does this circuit do?
  - Hint, inputs A and B are a two bit number.

Subroutines

- LIFO 12
  - Four 12bit registers
  - Stack counter
  - Decrementer
  - 2-4 onehot decoder
  - Four input 8bit MUX

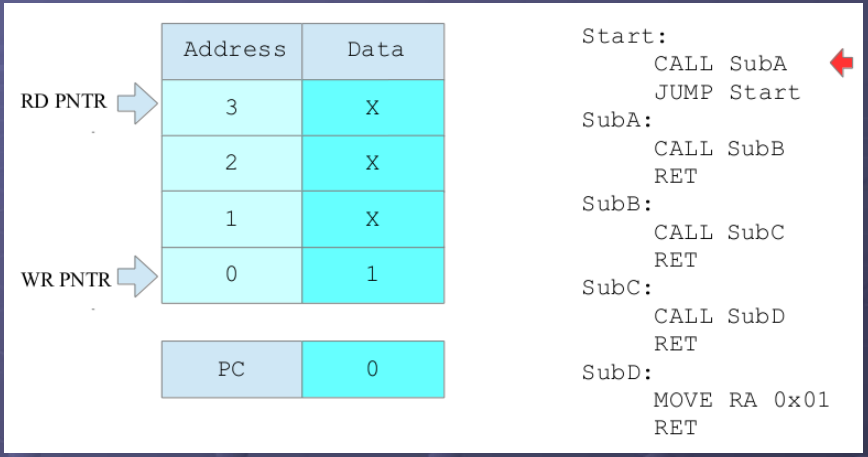


Subroutines



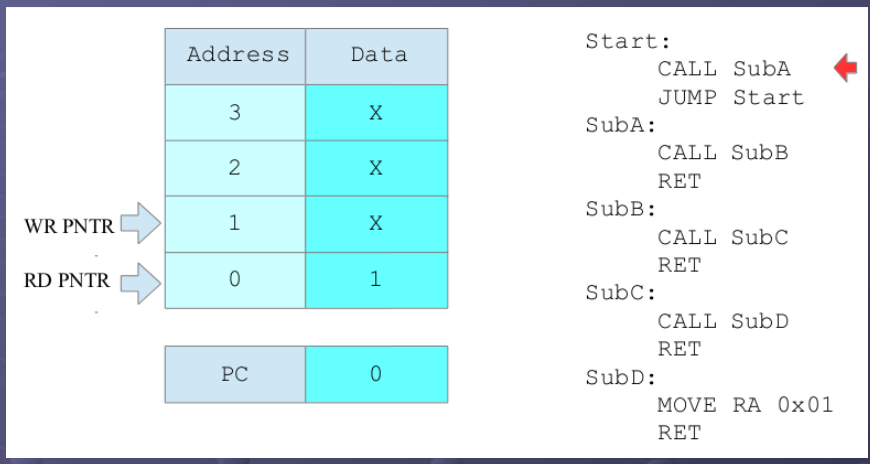
- Nested subroutine calls : reset
  - Execute instruction at address 0.

Subroutines



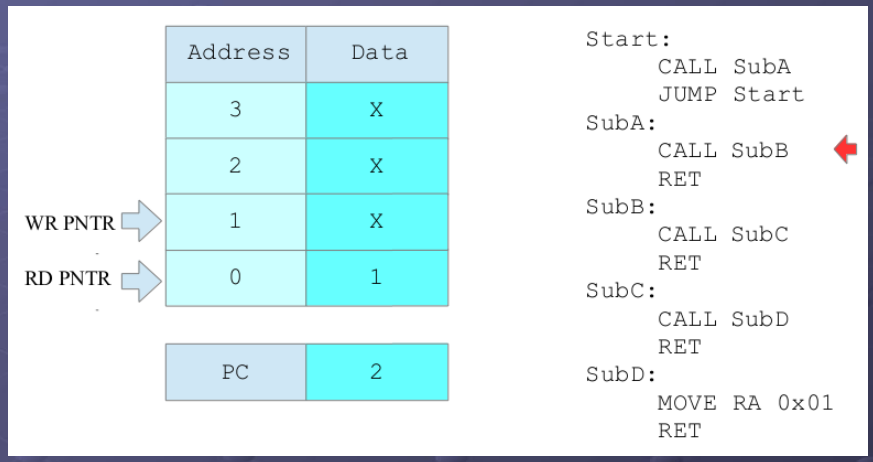
- Nested subroutine calls : CALL SubA
  - Push return address onto stack

Subroutines



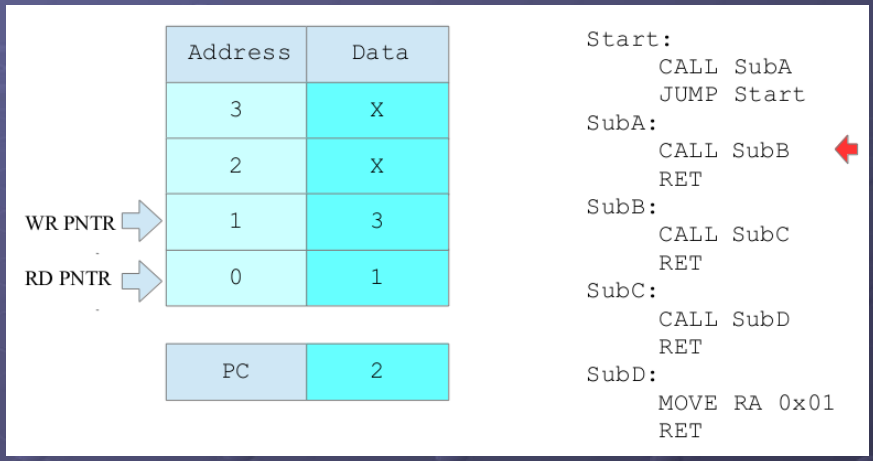
- Nested subroutine calls : CALL SubA
  - Update write and read pointers

Subroutines



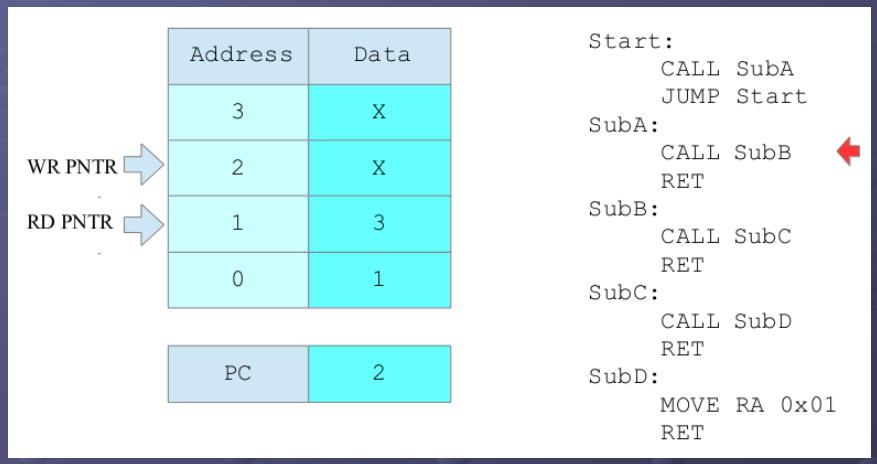
- Nested subroutine calls : CALL SubA
  - Update PC with address of SubA

Subroutines



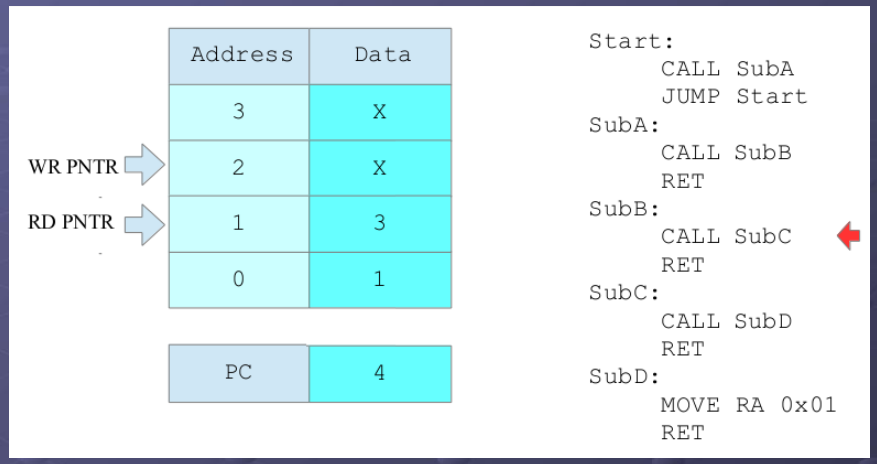
- Nested subroutine calls : CALL SubB
  - Push return address onto stack

Subroutines



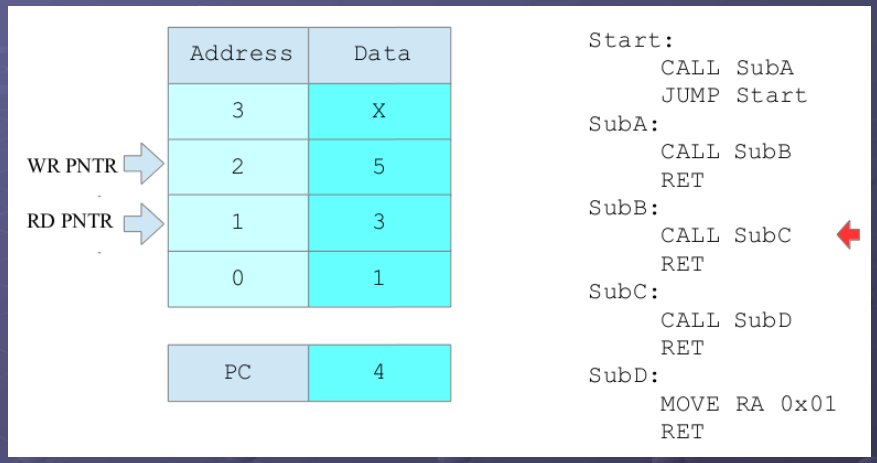
- Nested subroutine calls : CALL SubB
  - Update write and read pointers

Subroutines



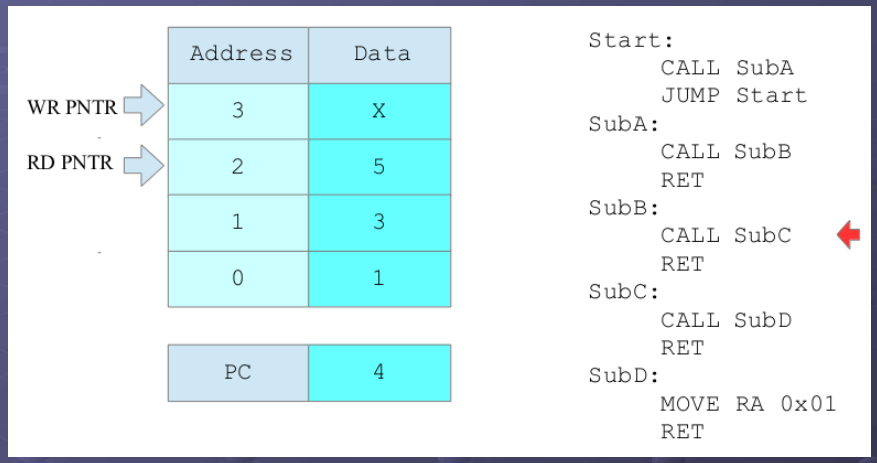
- Nested subroutine calls : CALL SubB
  - Update PC with address of SubB

Subroutines



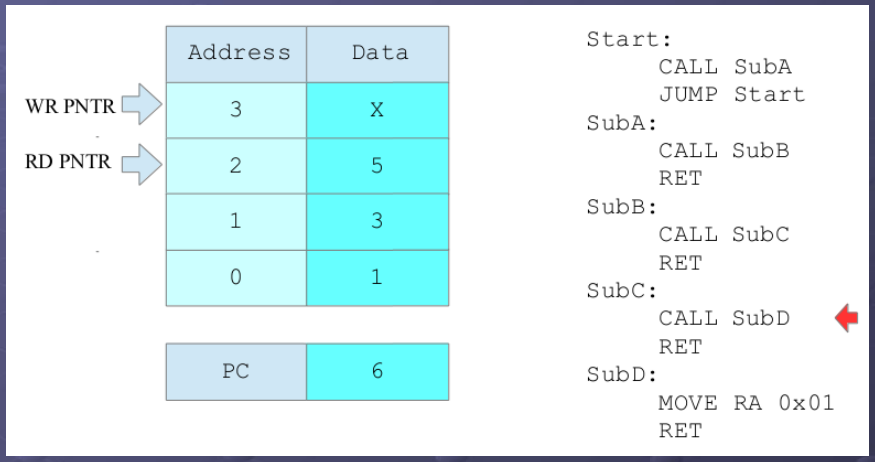
- Nested subroutine calls : CALL SubC
  - Push return address onto stack

Subroutines



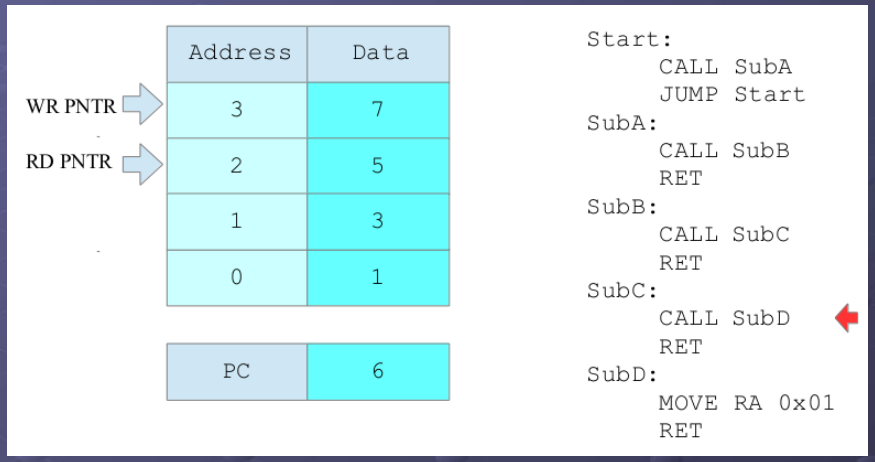
- Nested subroutine calls : CALL SubC
  - Update write and read pointers

Subroutines



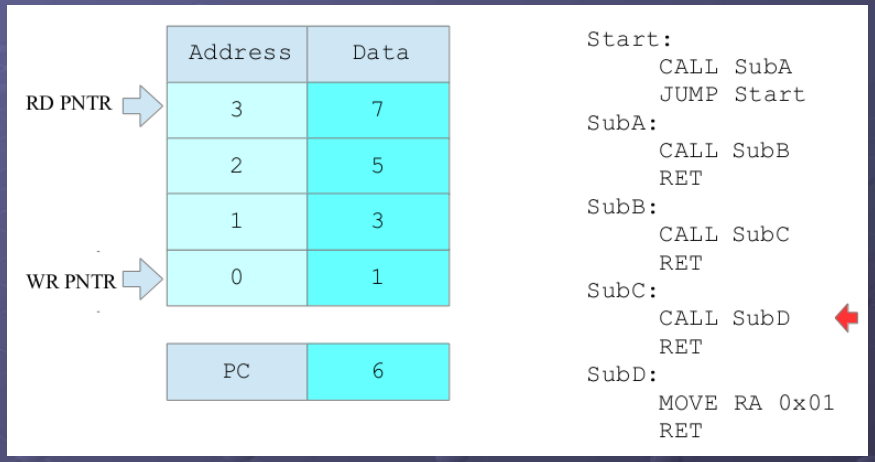
- Nested subroutine calls : CALL SubC
  - Update PC with address of SubC

Subroutines



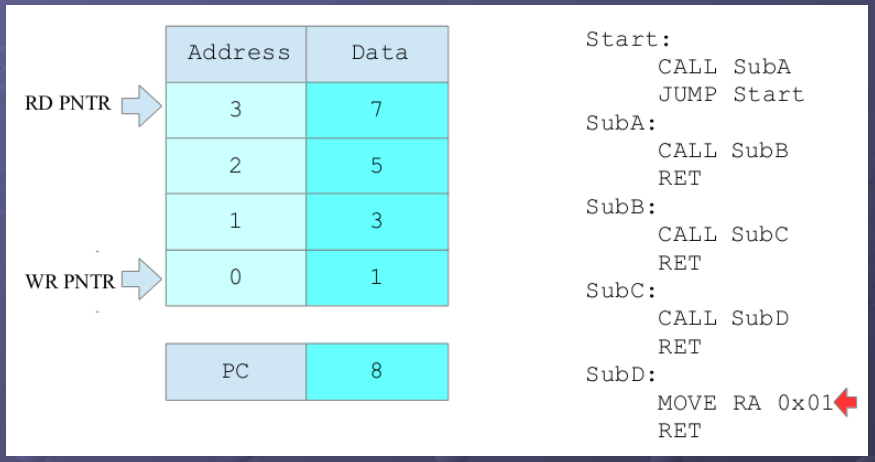
- Nested subroutine calls : CALL SubD
  - Push return address onto stack

Subroutines



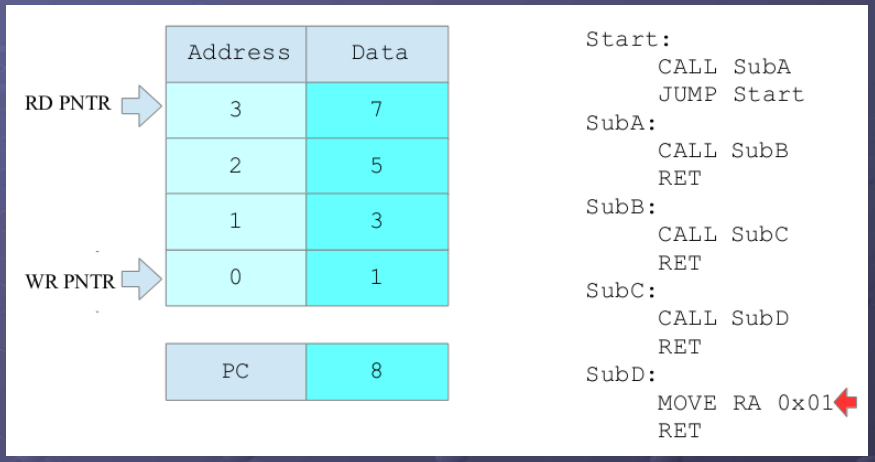
- Nested subroutine calls : CALL SubD
  - Update write and read pointers

Subroutines



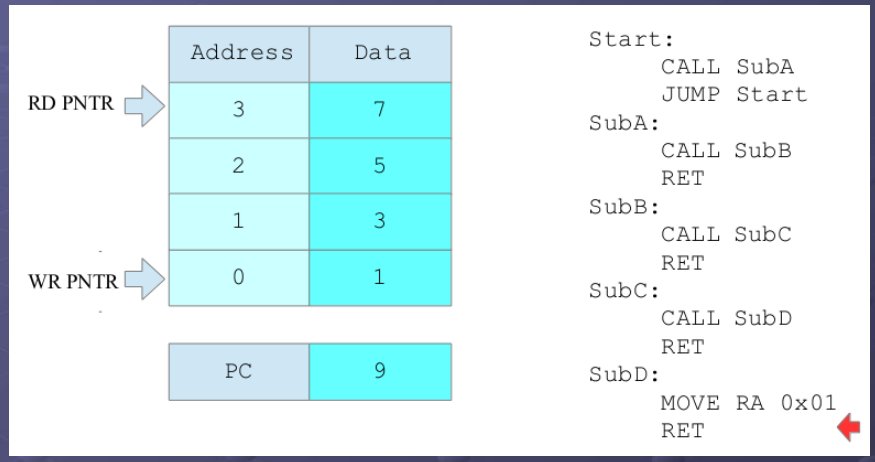
- Nested subroutine calls : CALL SubD
  - Update PC with address of SubD

Subroutines



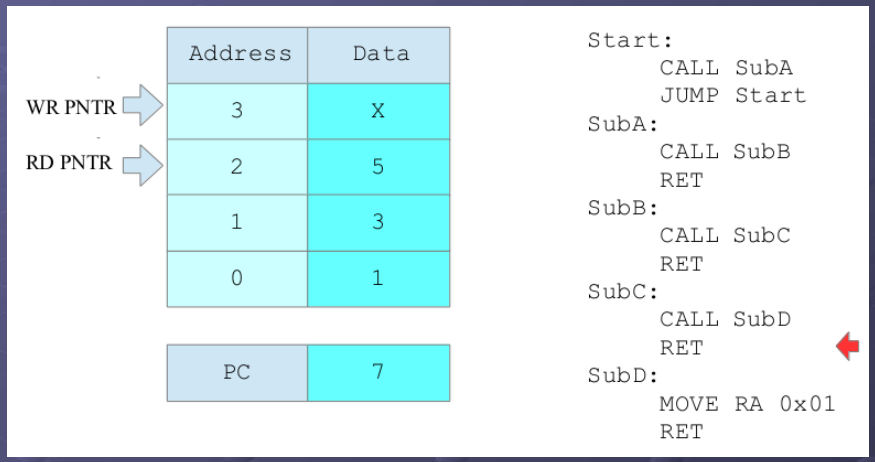
- Nested subroutine calls : Execute subroutine
  - MOVE RA 0x01

Subroutines



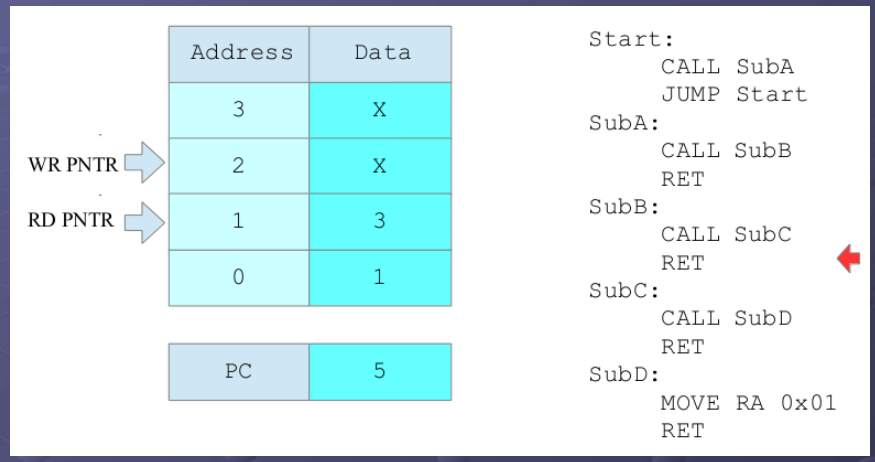
- Nested subroutine calls : RET
  - Pop return address off stack, update PC

Subroutines



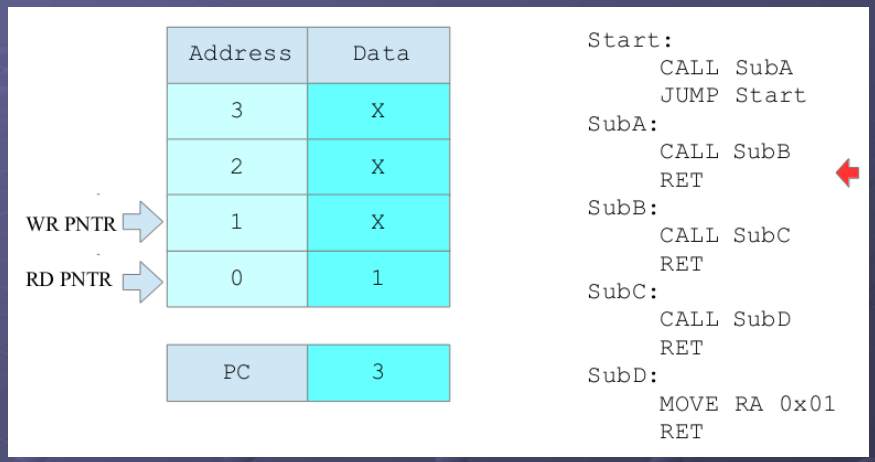
- Nested subroutine calls : RET
  - Update read and write pointers

Subroutines



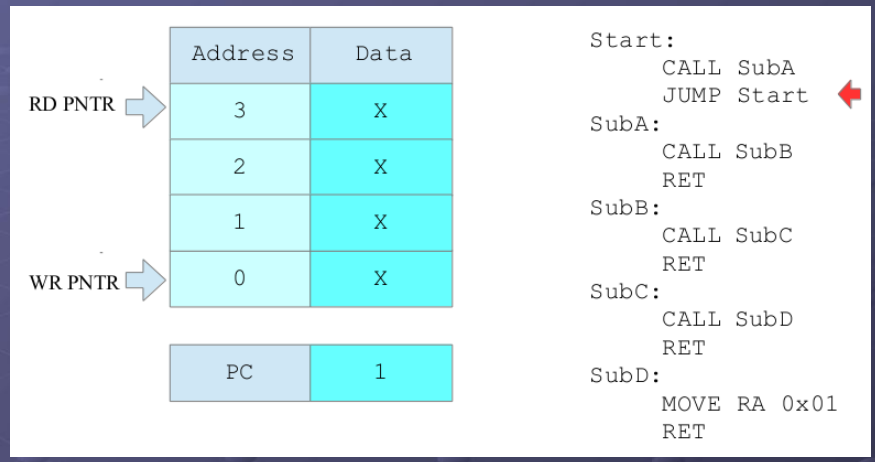
- Nested subroutine calls : RET
  - Pop return address off stack, update PC & pointers

Subroutines



- Nested subroutine calls : RET
  - Pop return address off stack, update PC & pointers

Subroutines



- Nested subroutine calls : RET
  - Pop return address off stack, update PC & pointers

Subroutines

- There are a lot of different solutions to implementing the CALL / RET stack, with varying levels of hardware and software support (next lecture).
  - Typically, rather than using a separate hardware component (LILO), the stack is implemented in external memory i.e. alongside instructions and data.
  - When a CALL instruction is executed the PC is automatically saved into one of the general purpose registers, or a specific address: stack pointer register (SP).
    - The return address is then calculated and written to memory, either under software or hardware control.
  - Calling parameters / local variables used by the subroutine also stored on the stack. Separate data stacks.

Summary

- The more complicated the problem the more instructions and data a program will need.
  - Always need more memory :)
  - Increasing internal storage (register file) helps reduce LOAD/STORE instructions, improving processing performance.
  - Improve memory efficiency (code density) by using subroutines.
- We would like to have an orthogonal instruction-set, however, we are limited by processor's architecture i.e. memory width, internal architecture etc.
  - When designing a CPU compromises have to be made :(
- Match the processor to the application domain i.e. in terms of instruction functionality and addressing modes.