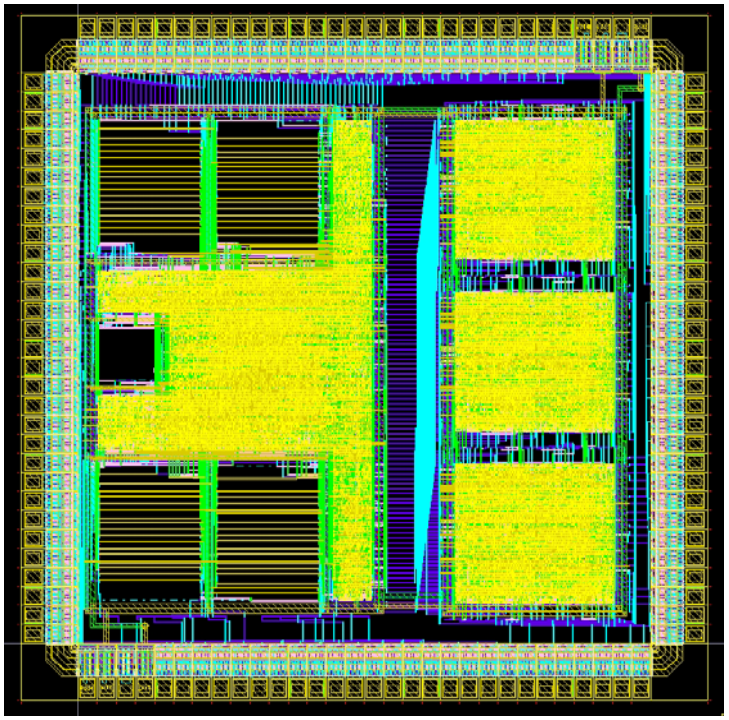
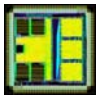
Digital Systems and Microprocessor Design (H7068)



9.3. Assembler memory access

Daniel Roggen
d.roggen@sussex.ac.uk



Content

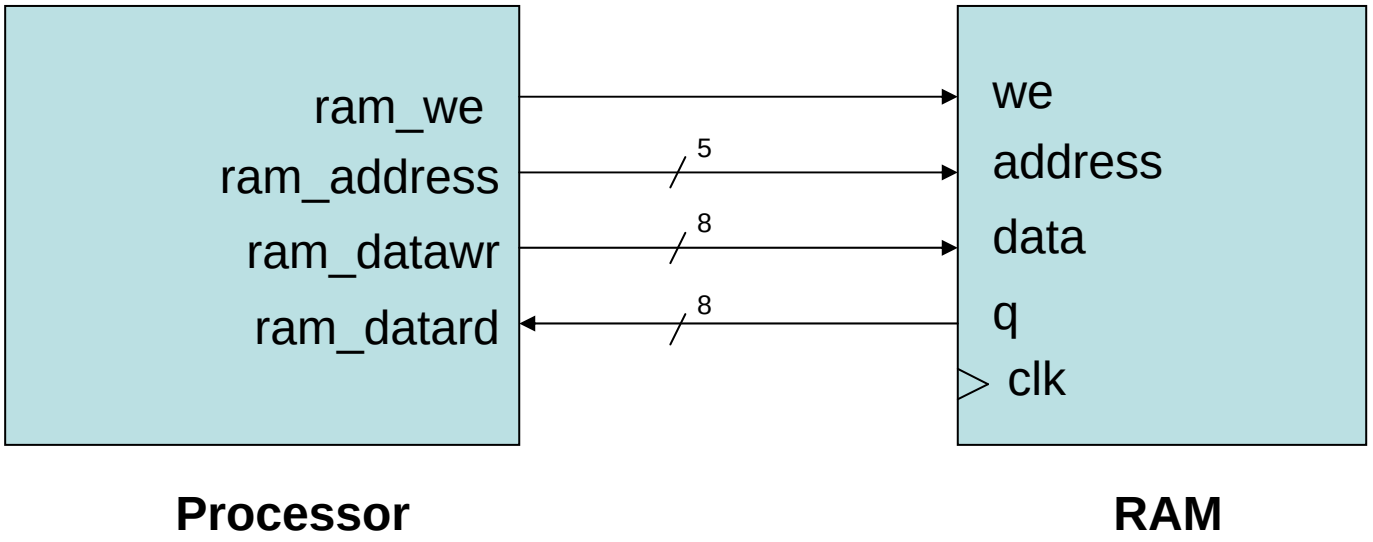
- Memory interface
- Memory read move instructions
- Memory write move instructions
- Mapping between C pointers and assembler instructions

Memory interface

- The educational processor has a memory interface allowing to read or write data from memory (or do nothing with it).
- Von Neumann architecture: the memory can contain code or data.
- When and how is the memory used?
- During fetch cycles:
  - The processor reads from memory two bytes (one byte during fetchh and one during fetchl cycles). This is the instruction to execute.
- During the execute cycle:
  - The memory is unused if the instruction is not a mov to/from memory
  - If the instruction is a mov to memory: one byte is written to the destination address
  - If the instruction is a mov from memory: one byte is read from the source address

Memory interface

- The UoS processor is connected to the RAM with the following lines:
  - ram_we (1 bit, processor output): indicates to write the data on ram_datawr to address ram_address at the next rising edge
  - ram_address (5 bits, processor output): indicates the RAM address to which to write and from which to read
  - ram_datawr (8 bits, processor output): indicates the byte to write
  - ram_datard (8 bits, processor input): the byte at address ram_address



Memory

- Remember a memory is akin to a table where data can be stored at an address

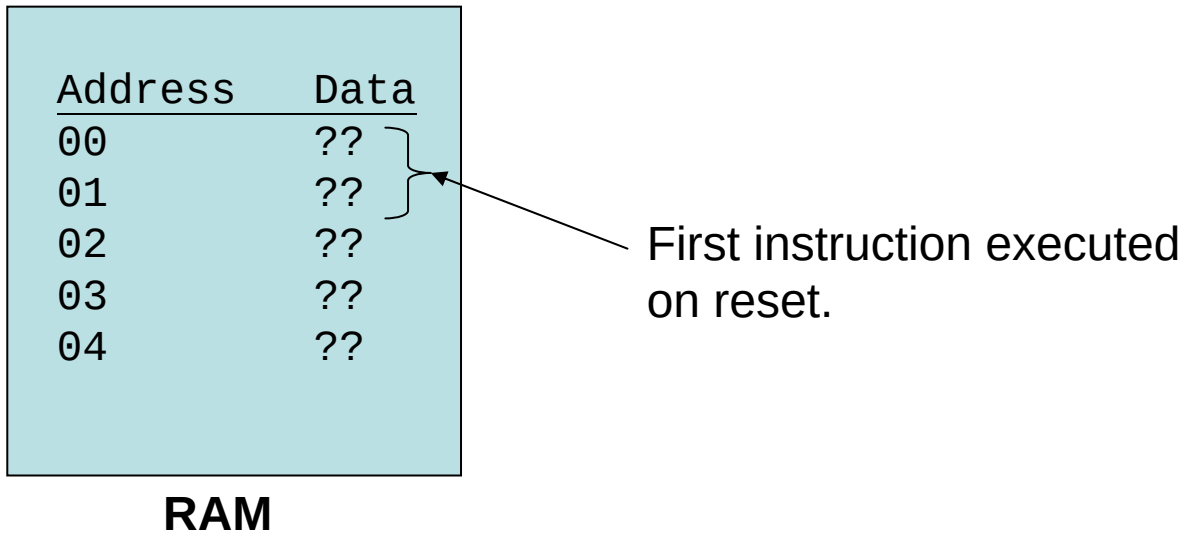
Address	Data
00	??
01	??
02	??
03	??
04	??

RAM

- The memory implemented alongside the UoS processor on the FPGA has the following characteristic:
  - Synchronous write: it writes “data” to “address” when “we=1” at the next clock rising edge
  - Asynchronous read: “q” is the data at “address”; changing the address gives immediately (after the propagation time) the data at this address

Memory

- On reset the program counter PC=0. Therefore the program starts at address 0, and the first instruction executed on reset is at address 0.



Accessing the memory

- In the UoS processor, the memory can only be accessed by “mov” instruction
- Reminder: **`mov dst, src`**

Instructions	instruction(15..8)								Instruction(7..0)							
Move	Opcode			$\overline{R}/I$	dd#m	sd#m	dreg (rn)		src (i or rm)							
mov rn, rm	0	0	0	0	0	0	r	r	-	-	-	-	-	-	r	r
mov rn, i	0	0	0	1	0	0	r	r	i	i	i	i	i	i	i	i
mov rn, [rm]	0	0	0	0	0	1	r	r	-	-	-	-	-	-	r	r
mov rn, [i]	0	0	0	1	0	1	r	r	i	i	i	i	i	i	i	i
mov [rn], rm	0	0	0	0	1	0	r	r	-	-	-	-	-	-	r	r
mov [rn], i	0	0	0	1	1	0	r	r	i	i	i	i	i	i	i	i

Reading data from a memory location

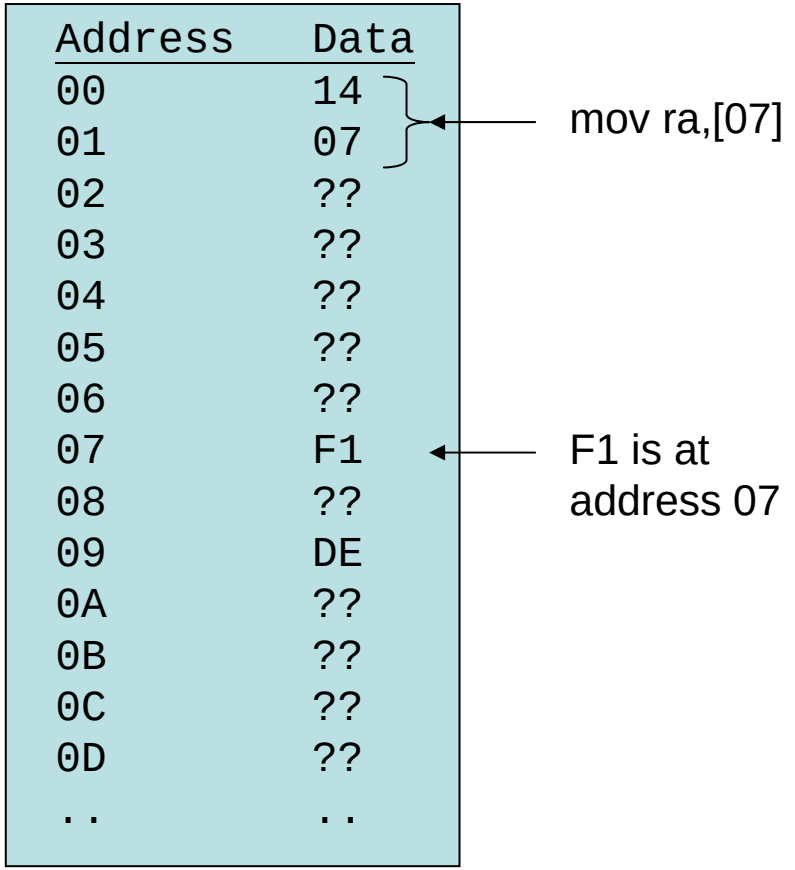
- Move from memory location can be done with immediate or register address.
- Immediate address: **`mov ra,[07h]`**
  - Reads the data at address 07h and puts it into register ra
- Register address: **`mov ra,[rb]`**
  - Reads the data at address rb and puts it into register ra.
  - For example, if rb=09, it reads data at address 09 and puts it into Ra

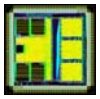
Instructions	**instruction(15..8)**								**Instruction(7..0)**							
Move	**Opcode**			$\overline{R}/I$	**dd#m**	**sd#m**	**dreg (rn)**		**src (i or rm)**							
`mov rn, [rm]`	**0**	**0**	**0**	**0**	**0**	**1**	**r**	**r**	**-**	**-**	**-**	**-**	**-**	**-**	**r**	**r**
`mov rn, [i]`	**0**	**0**	**0**	**1**	**0**	**1**	**r**	**r**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**

Reading data from a memory location

`mov ra,[07h]`

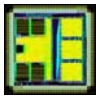
- When executing this instruction (exec cycle):
  - Control unit has put address (07) on “ram_address”
  - The memory is asynchronous for reads and the value F1h is placed on the memory output (ram_datard)
  - Control unit enables a register file write. It selects register a for write. It selects ram_datard as the data to write.
  - On the rising edge in the exec cycle, the value F1 (coming from the RAM) is thus stored in RA.





Reading data from a memory location

```
PC    Adr   Data Instr            RA RB RC RD
                                  0  0  0  0
->    00    1407 mov  ra,[07h]
      02    ???? ??
      04    ???? ??
      06    ??F1 ??
      08    ??DE ??
      0A    ???? ??
```



Reading data from a memory location

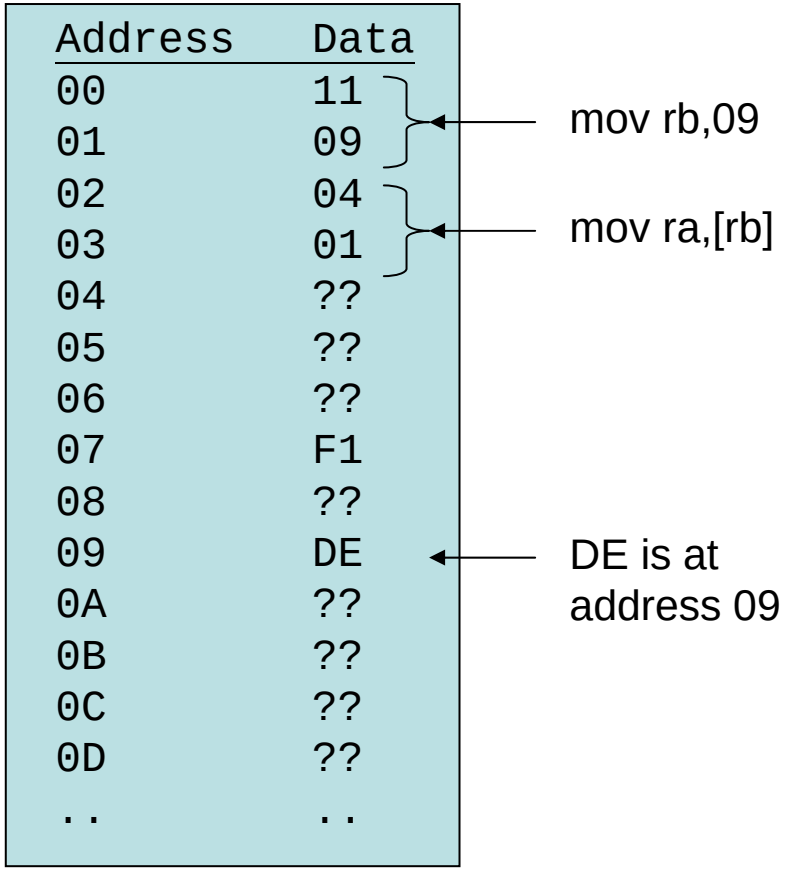
```
PC    Adr   Data Instr           RA RB RC RD
                                 F1 0  0  0
      00    1407 mov  ra,[07h]
->    02    ???? ??
      04    ???? ??
      06    ??F1 ??
      08    ??DE ??
      0A    ???? ??
```

Reading data from a memory location

mov rb,09

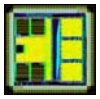
mov ra,[rb]

- When executing the memory mov instruction (exec cycle):
  - Control unit has put address, which is in register ra (09) on “ram_address”
  - The memory is asynchronous for reads and the value DEh is placed on the memory output (ram_datard)
  - Control unit enables a register file write. It selects register a for write. It selects ram_datard as the data to write.
  - On the rising edge in the exec cycle, the value F1 (coming from the RAM) is thus stored in RA.



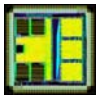
Reading data from a memory location

```
PC    Adr   Data Instr              RA RB RC RD
                                    0  0  0  0
->    00    1109 mov  rb,09h
      02    0401 mov  ra,[rb]
      04    ???? ??
      06    ??F1 ??
      08    ??DE ??
      0A    ???? ??
```



Reading data from a memory location

```
PC    Adr   Data Instr              RA RB RC RD
                                     0  09 0  0
      00    1109 mov  rb,09h
->    02    0401 mov  ra,[rb]
      04    ???? ??
      06    ??F1 ??
      08    ??DE ??
      0A    ???? ??
```



Reading data from a memory location

```
PC    Adr   Data Instr              RA RB RC RD
                                    DE 09 0  0
      00    1109 mov   rb,09h
      02    0401 mov   ra,[rb]
->    04    ???? ??
      06    ??F1 ??
      08    ??DE ??
      0A    ???? ??
```

Writing data to a memory location

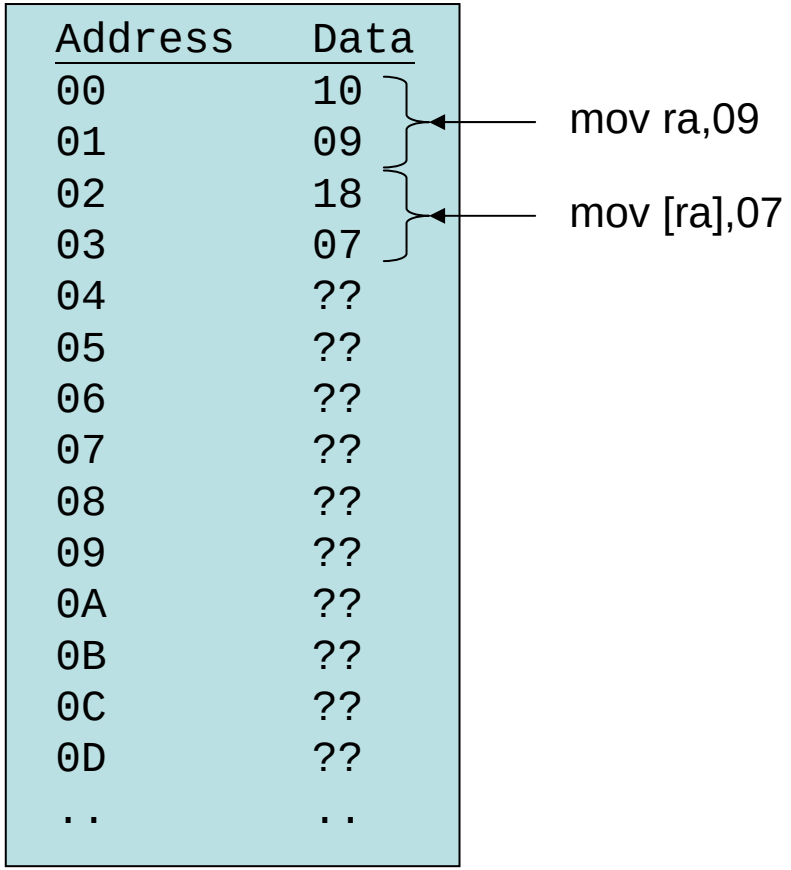
- Move to memory location can be done only with register address. The source can be a register or immediate.
  - This is due to the choice of instruction encoding; with 16 bit instructions the UoS processor cannot have immediate memory destination and immediate source! A different encoding would be required.
- Immediate source: **`mov [ra],07h`**
  - Writes 07h to the address ra.
  - For example, if ra=09, then 07h is written to addres 09h.
- Register source: **`mov [ra],rb`**
  - Writes the data rb to the addres ra.

Instructions	**instruction(15..8)**								**Instruction(7..0)**							
Move	**Opcode**			$\overline{R}/I$	**dd#m**	**sd#m**	**dreg (rn)**		**src (i or rm)**							
`mov [rn], rm`	0	0	0	0	1	0	r	r	-	-	-	-	-	-	r	r
`mov [rn], i`	0	0	0	1	1	0	r	r	i	i	i	i	i	i	i	i

Writing data to a memory location

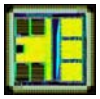
```
mov ra,09h
mov [ra],07
```

- When executing this instruction (exec cycle):
  - Control unit has put address (09) on “ram_address”
  - Control unit has put data (07) on ram_datawr.
  - Control unit has enabled ram write “ram_we=1”
  - The memory is synchronous for writes. On the rising edge in the exec cycle, the value 07 is stored to address 09.



Writing data to a memory location

```
PC    Adr   Data Instr              RA RB RC RD
                                     0  0  0  0
->    00    1000 mov  ra,09h
      02    1807 mov  [ra],07
      04    ???? ??
      06    ???? ??
      08    ???? ??
      0A    ???? ??
```



Writing data to a memory location

```
PC    Adr   Data Instr              RA RB RC RD
                                    09 0  0  0
      00    1000 mov  ra,09h
->    02    1807 mov  [ra],07
      04    ????
      06    ????
      08    ????
      0A    ????
```

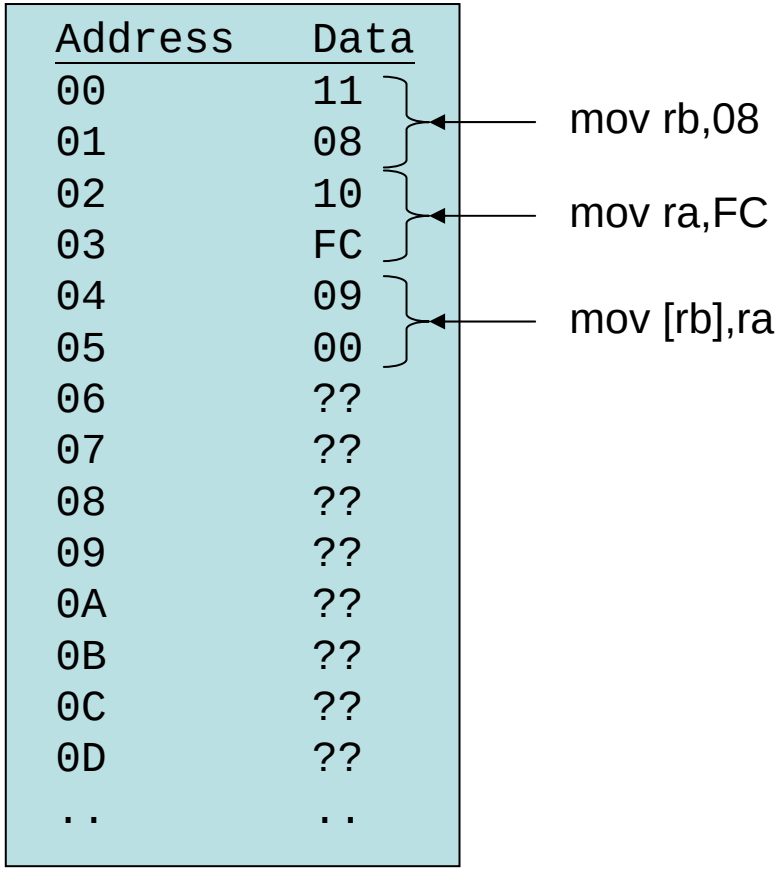
Writing data to a memory location

```
PC    Adr   Data Instr              RA RB RC RD
                                     09 0  0  0
      00    1000 mov  ra,09h
      02    1807 mov  [ra],07
->    04    ????
      06    ????
      08    ??07
      0A    ????
```

Writing data to a memory location

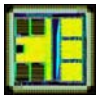
```
mov rb,08h
mov ra,FCh
mov [rb],ra
```

- When executing this instruction (exec cycle):
  - Control unit has put address (08) on “ram_address”
  - Control unit has put data (FC) on ram_datawr.
  - Control unit has enabled ram write “ram_we=1”
  - The memory is synchronous for writes. On the rising edge in the exec cycle, the value FC is stored to address 08.



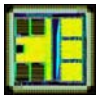
Writing data to a memory location

```
PC    Adr   Data Instr              RA RB RC RD
                                     0  0  0  0
->    00    1108 mov  rb,08h
      02    10FC mov  ra,FCh
      04    0900 mov  [rb],ra
      06    ???? ??
      08    ???? ??
      0A    ???? ??
```



Writing data to a memory location

```
PC    Adr   Data Instr              RA RB RC RD
                                     0  08 0  0
      00    1108 mov  rb,08h
->    02    10FC mov  ra,FCh
      04    0900 mov  [rb],ra
      06    ???? ??
      08    ???? ??
      0A    ???? ??
```



Writing data to a memory location

```
PC    Adr   Data Instr          RA RB RC RD
                                 FC 08 0  0
      00    1108 mov  rb,08h
      02    10FC mov  ra,FCh
->    04    0900 mov  [rb],ra
      06    ???? ??
      08    ???? ??
      0A    ???? ??
```

Writing data to a memory location

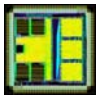
```
PC    Adr   Data Instr          RA RB RC RD
                                 FC 08 0  0
      00    1108 mov  rb,08h
      02    10FC mov  ra,FCh
      04    0900 mov  [rb],ra
->    06    ???? ??
      08    FC?? ??
      0A    ???? ??
```

ALU operations with memory operands

- In the UoS processor, ALU operations cannot be directly performed on memory data. Instead:
  - Read the operands from memory and put them in registers
  - Perform the ALU operation on the registers
  - Write the result register to the destination memory

- Example: read data at address 1C and 1D, add them together, and store the result at address 1C

```
mov ra,1Ch
mov rb,[ra]
mov rc,[1Dh]
add rb,rc
mov [ra],rb
```



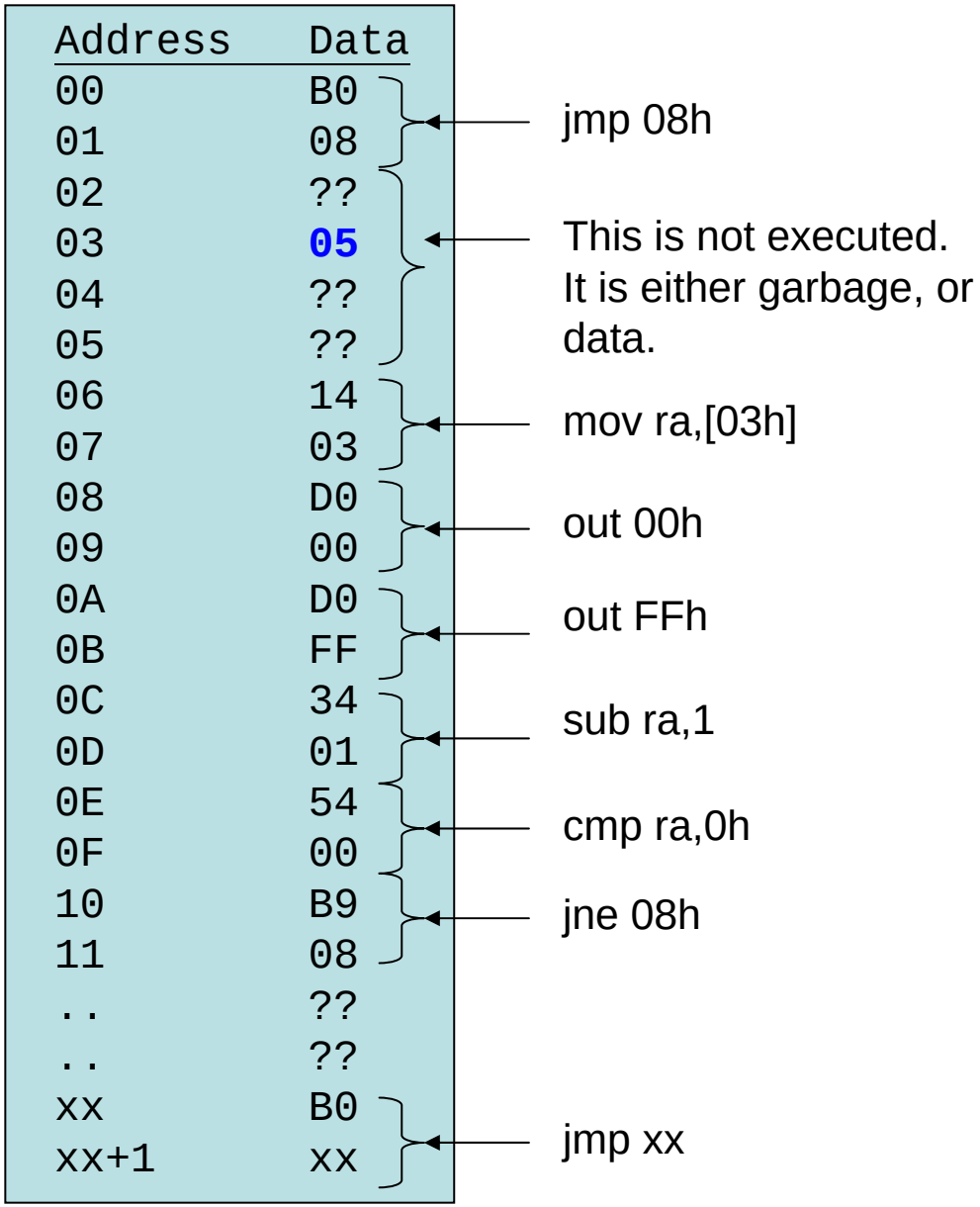
ALU operations with memory operands

- Other processor architectures may have ALU operations allowing memory operands.

- This is the case with Intel/AMD x86. The following are some of the available possibilities:

```
add reg,reg
add reg,[mem]
add [mem],reg
add reg,imm
add [mem],imm
```

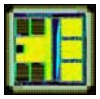
Mixing code and data in memory



- Von Neumann: code and data can be mixed!
- It is up to the programmer (or the compiler) to know where to place data and code in the memory.
- Only constraint: the first instruction is at address 0
- This program iterates between 06-10 according to the value at address 3

Memory move and C pointers

- C pointers directly translate to memory move instructions
  - C was designed as a language that easily maps to typical processor architectures
- C has variables and pointers.
  - Pointers are also variables; they allow in addition access to memory with '*'.
- Direct analogy to assembler instructions we have seen!



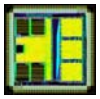
C variables and assembler

- **`char v1;`**
  - v1 contains a value. May map to a register.
- **`register char v2;`**
  - v2 contains a value. The keyword “register” indicates the compiler we wish v2 to be in a register.

- The compiler decides where to store a variable: in memory or in registers. If registers are available then the compiler will use them for variables, as this leads to more compact code.
- The “register” keyword indicates the compiler we wish to have the variable in a register, but it is non-binding.
- Variables can be operated on: incremented, decremented, etc.
- Eg:

```
register char v1,v2,v3;
v1 = 0x23;
v2 = 0x12;
v3 = v1 + v2;
```

v3 is 0x35 here



C variables and assembler

Indicates we would like the variables in registers, but it is non binding.

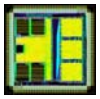
- C:

```
register char v1,v2,v3;
v1 = 0x23;
v2 = 0x12;
v3 = v1 + v2;
```

- Let's assume the human/compiler assigns v1,v2,v3 to ra,rb,rc respectively.

- Equivalent in assembler:

```
mov ra,23h
mov rb,12h
mov rc,ra
add rc,rb
```



C pointers and assembler

- Pointers are variables that contain a value.... the subtlety is that this value indicates a memory location that can be read from / written to.
- **`char *v1;`**
  - v1 contains a value. May map to a register.
- **`register char *v2;`**
  - v2 contains a value. The keyword “register” indicates the compiler we wish v2 to be in a register.
- v1 and v2: they contain a value, but this value indicates a memory location we can read from or write to.
- Eg:

```
register char *v1,*v2,*v3;
v1 = 0x23;
v2 = 0x12;
v3 = v1 + v2;
```

v3 is 0x35 here

No difference until here!

C pointers and assembler

- C:

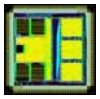
```
register char *v1,*v2,*v3;
v1 = 0x23;
v2 = 0x12;
v3 = v1 + v2;
```

At this point, v3 is 0x35

- Pointers are variables that can be modified by arithmetic operations, just like normal variables.
- Let's assume the compiler assigns v1,v2,v3 to ra,rb,rc respectively.

- Equivalent in assembler:

```
mov ra,23h
mov rb,12h
mov rc,ra
add rc,rb
```

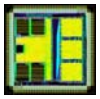


C pointers and assembler

- In addition, pointers can be used to read or write memory locations.

- Writing byte to address v: **`*v = x;`**
  - Maps to instructions: **`mov [rn],i`** or **`mov [rn],rm`**

- Reading byte from address v: **`x = *v;`**
  - Maps to instruction: **`mov rn,[i]`** or **`mov rn,[rm]`**



C pointers and assembler: writing memory

- Example:

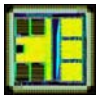
```
register char *v1,*v2;
v1 = 0x0B;
v2 = 0x0D;
*v1 = 0xFE;
*v2 = 0x3F;
```

Here writing to memory location 0B and 0D

- Assembler:

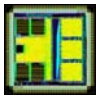
```
mov ra,0Bh
mov rb,0Dh
mov [ra],FEh
mov [rb],3Fh
```

Here writing to memory location 0B and 0D



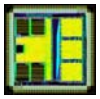
C pointers and assembler: writing memory

```
PC   Adr  Data Instr          RA RB RC RD
                              0  0  0  0
->   00   100B mov  ra,0Bh
     02   110D mov  rb,0Dh
     04   18FE mov  [ra],FEh
     06   193F mov  [rb],3Fh
     08   ???? ??
     0A   ???? ??
     0C   ???? ??
     0E   ???? ??
```



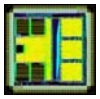
C pointers and assembler: writing memory

```
PC    Adr   Data Instr              RA RB RC RD
                                     0B 0  0  0
      00    100B mov   ra,0Bh
->    02    110D mov   rb,0Dh
      04    18FE mov   [ra],FEh
      06    193F mov   [rb],3Fh
      08    ???? ??
      0A    ???? ??
      0C    ???? ??
      0E    ???? ??
```



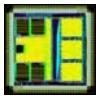
C pointers and assembler: writing memory

```
PC    Adr   Data Instr              RA RB RC RD
                                     0B 0D 0  0
      00    100B mov   ra,0Bh
      02    110D mov   rb,0Dh
->    04    18FE mov   [ra],FEh
      06    193F mov   [rb],3Fh
      08    ???? ??
      0A    ???? ??
      0C    ???? ??
      0E    ???? ??
```



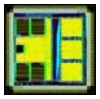
C pointers and assembler: writing memory

```
PC    Adr   Data Instr              RA RB RC RD
                                     0B 0D 0  0
      00    100B mov   ra,0Bh
      02    110D mov   rb,0Dh
      04    18FE mov   [ra],FEh
->    06    193F mov   [rb],3Fh
      08    ???? ??
      0A    ??FE ??
      0C    ???? ??
      0E    ???? ??
```



C pointers and assembler: writing memory

```
PC    Adr   Data Instr              RA RB RC RD
                                     0B 0D 0  0
      00    100B mov   ra,0Bh
      02    110D mov   rb,0Dh
      04    18FE mov   [ra],FEh
      06    193F mov   [rb],3Fh
->    08    ???? ??
      0A    ??FE ??
      0C    ??3F ??
      0E    ???? ??
```



C pointers and assembler: reading memory

- Example:

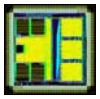
```
register char *v1;
register char v2;
v1 = 0x0B;
v2 = *v1;
```

} **Here reading from memory location 0B**

- Assembler:

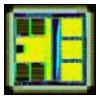
```
mov ra,0Bh
mov rb,[ra]
```

} **Here reading from memory location 0B**



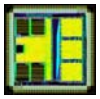
C pointers and assembler: reading memory

```
PC    Adr   Data Instr              RA RB RC RD
                                     0  0  0  0
->    00    100B mov  ra,0Bh
      02    0500 mov  rb,[ra]
      04    ???? ??
      06    ???? ??
      08    ???? ??
      0A    ??DB ??
      0C    ???? ??
      0E    ???? ??
```



C pointers and assembler: reading memory

```
PC    Adr   Data Instr              RA RB RC RD
                                     0B 0  0  0
      00    100B mov   ra,0Bh
->    02    0500 mov   rb,[ra]
      04    ???? ??
      06    ???? ??
      08    ???? ??
      0A    ??DB ??
      0C    ???? ??
      0E    ???? ??
```



C pointers and assembler: reading memory

```
PC    Adr   Data Instr              RA RB RC RD
                                     0B DB 0  0
      00    100B mov  ra,0Bh
      02    0500 mov  rb,[ra]
->    04    ???? ??
      06    ???? ??
      08    ???? ??
      0A    ??DB ??
      0C    ???? ??
      0E    ???? ??
```

Summary

- Assembler instruction “mov” allows to read/write data to memory
- This allows to perform computations on much more data than there are registers available.
- Memory can contain data or code (Von Neumann architecture)
- Direct mapping between memory move operations and C pointers