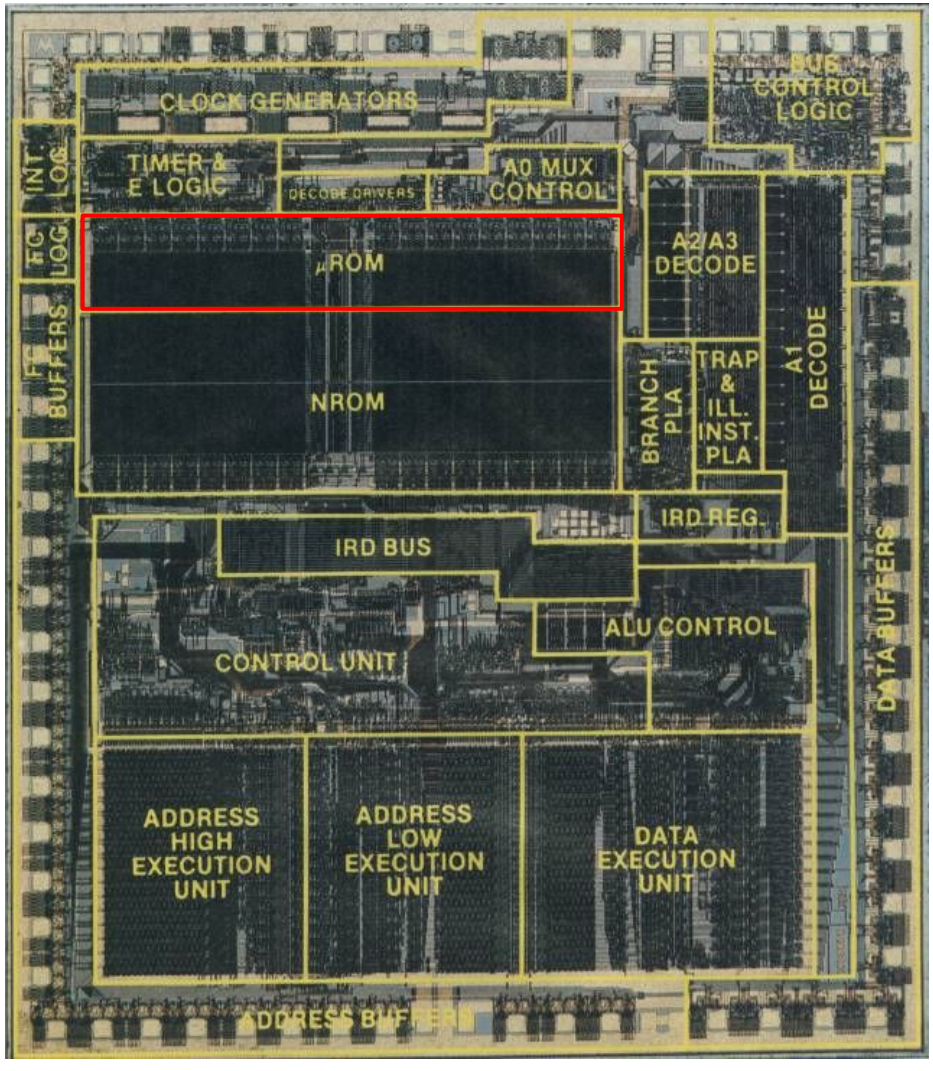
Microcode machine

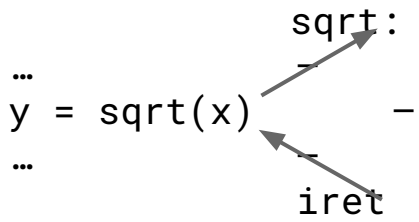
CSC 236

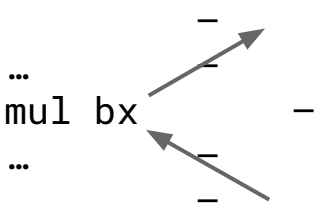

CLOCK GENERATORS
BUS CONTROL LOGIC
INT. LOG.
TIMER & E LOGIC
DECODE DRIVERS
A0 MUX CONTROL
FC LOG.
μROM
A2/A3 DECODE
I/F BUFFERS
NROM
BRANCH PLA
TRAP & ILL. INST. PLA
A1 DECODE
IRD REG.
IRD BUS
ALU CONTROL
CONTROL UNIT
DATA BUFFERS
ADDRESS HIGH EXECUTION UNIT
ADDRESS LOW EXECUTION UNIT
DATA EXECUTION UNIT
ADDRESS BUFFERS

Microprogramming

- A way to implement machine instructions
- Not visible to assembler programmer
  - Not the ISA -- instruction set architecture
  - Implements the ISA
- A machine instruction (ISA)
  - Consists of 1 or more micro-operations
- Each micro-operation will
  - Control data flow
  - Activate function units in the CPU

Analogous to a subroutine call … but invisible

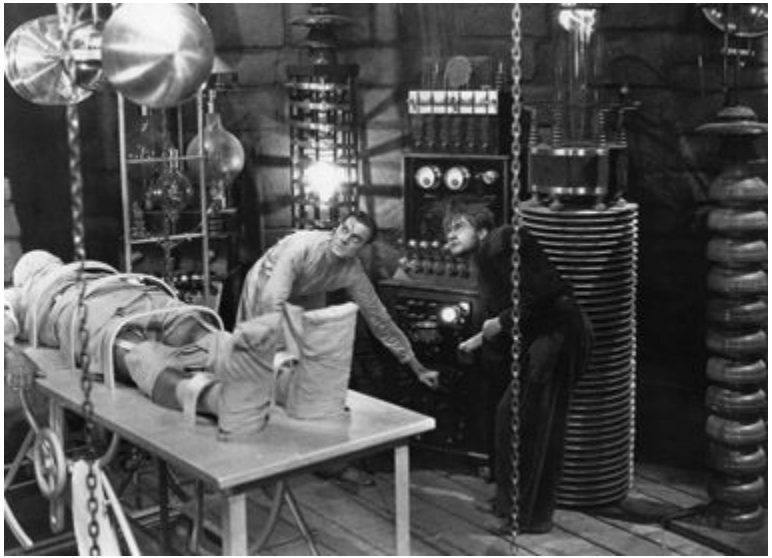




Power of microcode

- Microcode gives hardware its personality (defines what machine instructions do)
- Less expensive processors have the same instruction set as expensive processors
- One system can emulate another system
- Potential performance improvement

Let's build our machine



Machine characteristics

- Very simple embedded CPU
- 256 bytes of memory
- 2 programmer-accessible 8-bit registers
  - Accumulator (like ax on the 8086)
  - Index register (like si on the 8086)
- 2 machine code formats
  - 1 byte OPCODE
  - 2 byte OPCODE OPERAND

Machine Code Formats

1 byte OPCODE

- In OPCODE
  - Instruction description
  - Registers used (if any)
- Most operations w/ only regs

```
clear acc         ;acc=0
inc   index       ;index=index+1
dec   acc         ;acc=acc-1
add   index,acc   ;index=index+acc
sub   acc,index   ;acc=acc-index
```

All this will go in the opcode.

2 bytes OPCODE OPERAND

- OPCODE is same
- OPERAND
  - Memory address
  - Immediate data

```
load acc,[var]    ;acc=[var]
add  index,[var]  ;index=index+[var]
load index,25     ;index=25
sub  acc,[var]    ;acc=acc-[var]
```

This will require an extra byte.

Constants

- Consider
  - `clear acc`
  - Sets acc to 0
  - µop: `acc = 0`
- Where does the value 0 come from?
- Consider
  - `inc index`
  - µop: `index = index + 1`
- Where does the value 1 come from?

8 internal byte registers

000	program counter
001	instruction register
010	instruction register
011	constant 0
100	constant 1
101	accumulator
110	index register
111	μcode work register

8 internal byte registers

000	program counter
001	instruction register
010	instruction register
011	constant 0
100	constant 1
101	accumulator
110	index register
111	μcode work register

2 are programmer accessible

8 internal byte registers

000	program counter
001	instruction register
010	instruction register
011	constant 0
100	constant 1
101	accumulator
110	index register
111	μcode work register

6 are used internally by microcode

2 are programmer accessible

8 internal byte registers

000	program counter
001	instruction register
010	instruction register
011	constant 0
100	constant 1
101	accumulator
110	index register
111	μcode work register

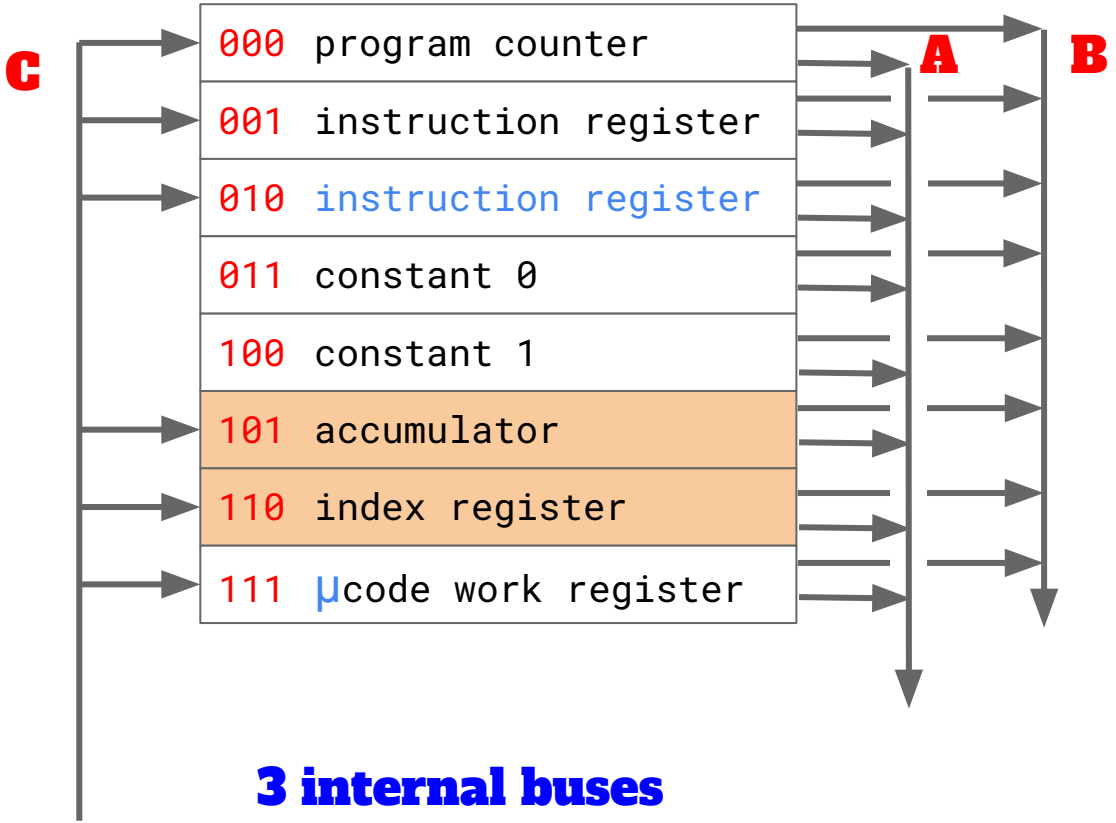
next sequential instruction

OPCODE

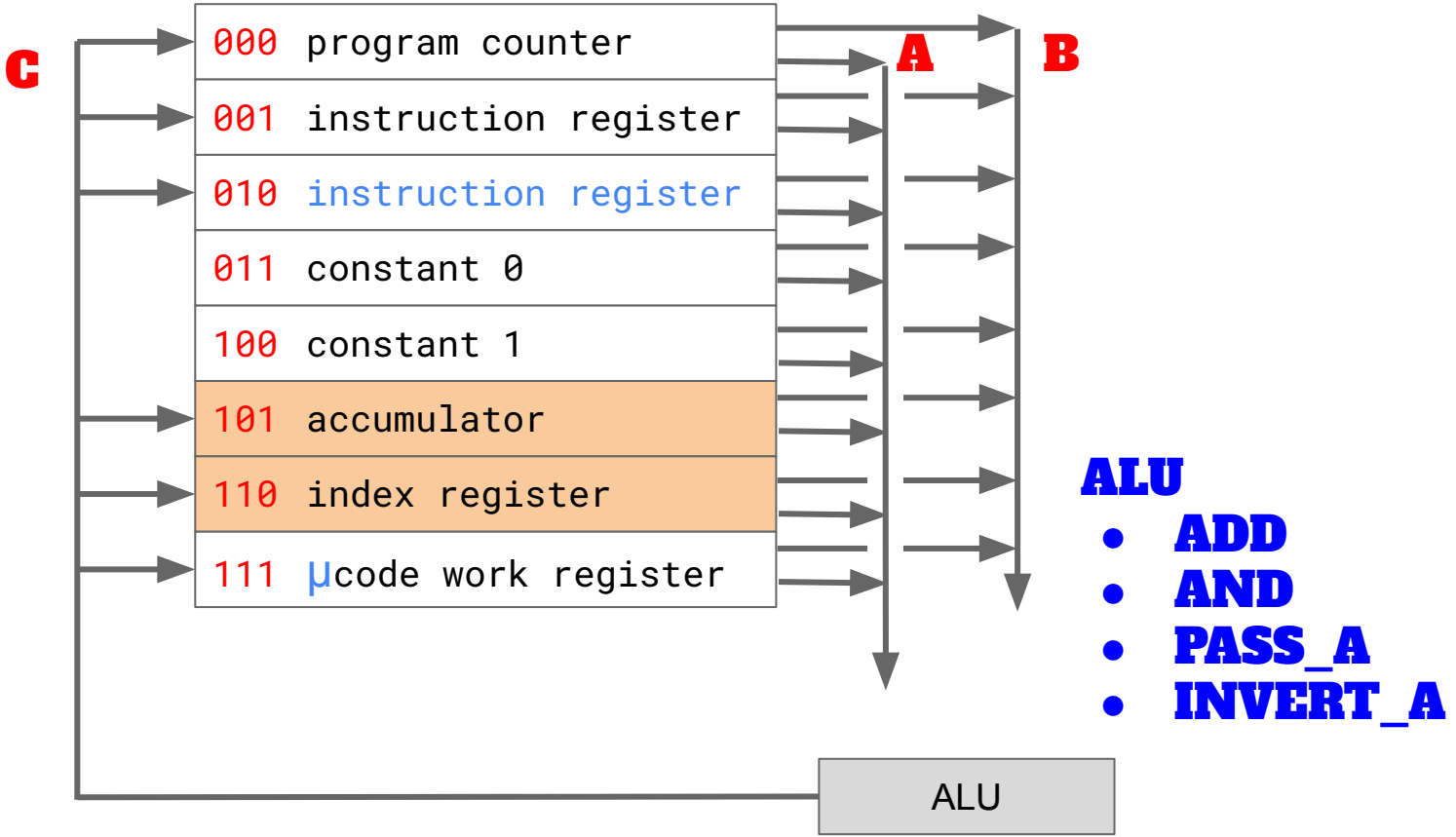
OPERAND

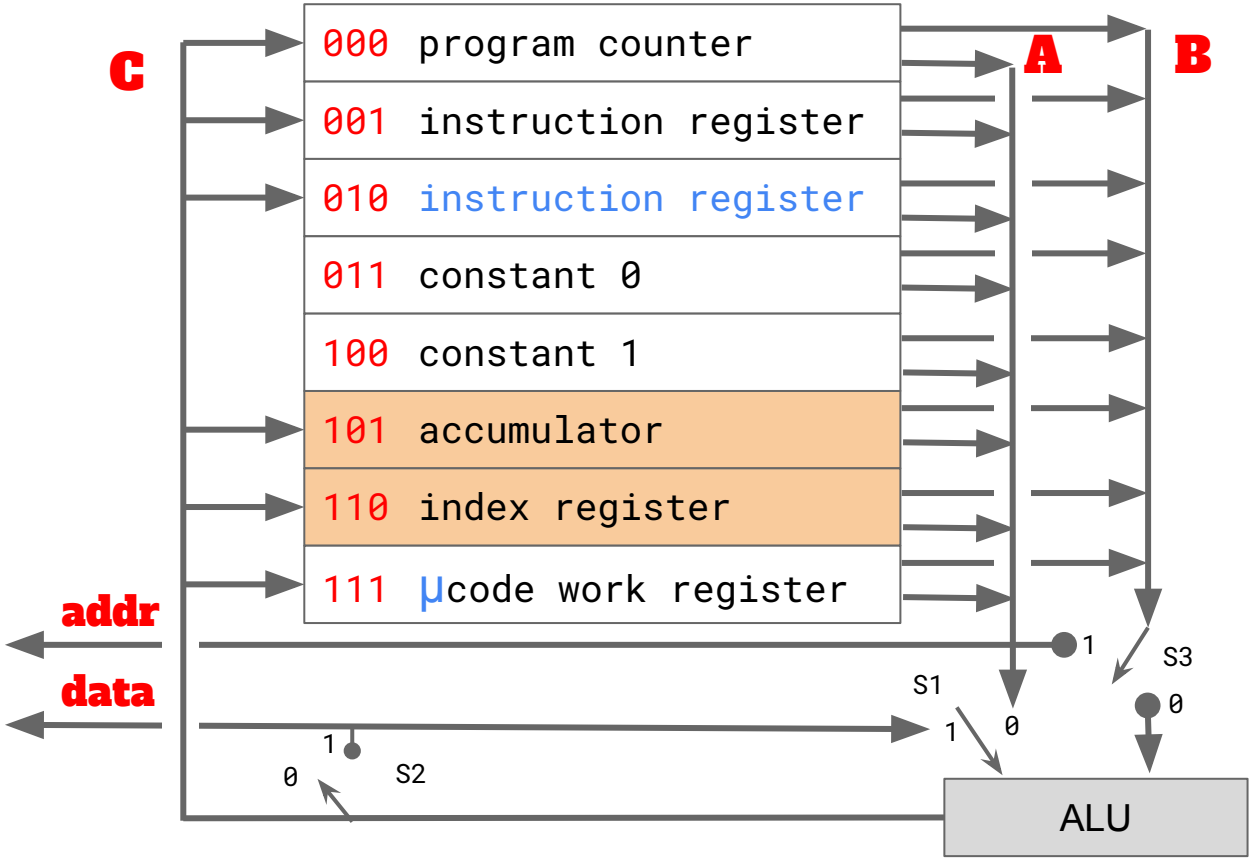
25% of registers are just constants!

temporary storage

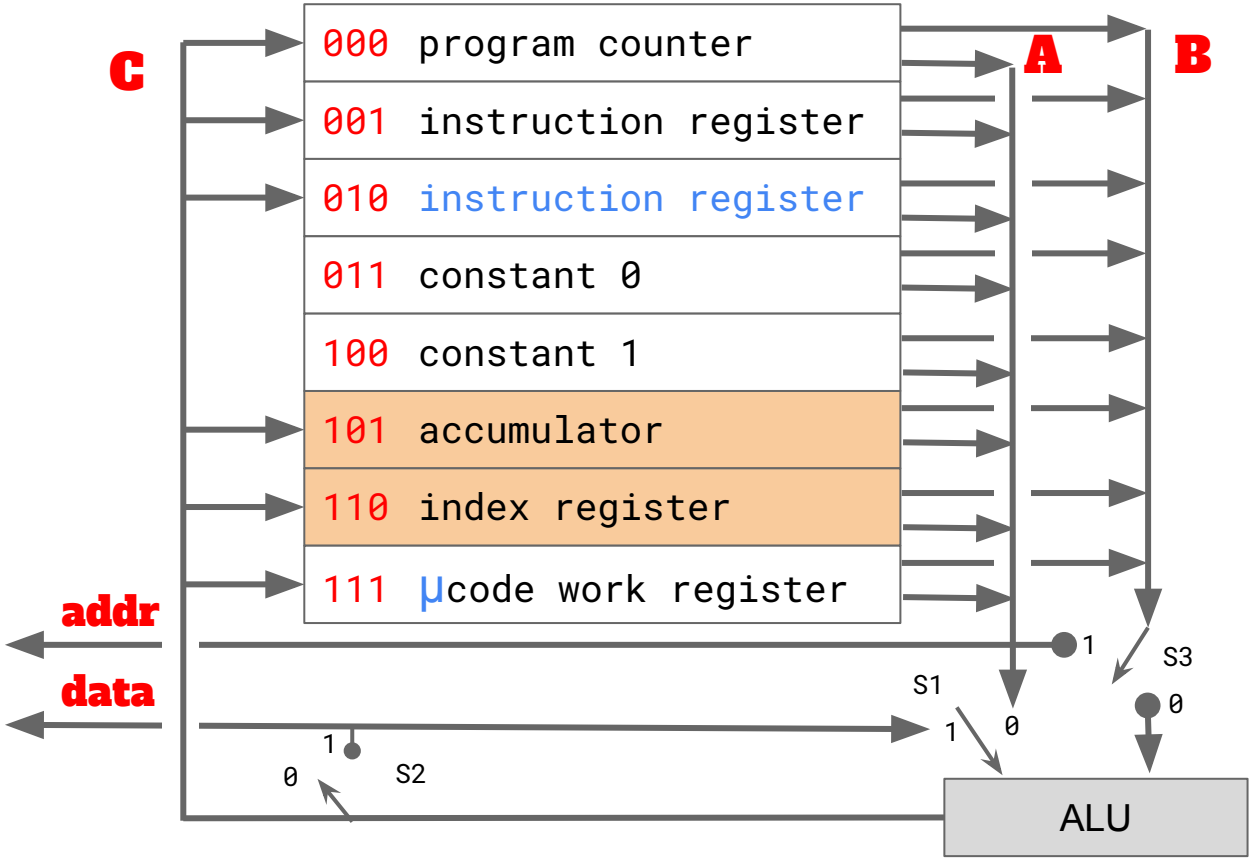


3 internal buses
All registers can feed A & B
C bus can feed any of 6 registers


C
000 program counter
001 instruction register
010 instruction register
011 constant 0
100 constant 1
101 accumulator
110 index register
111 μcode work register
A
B
ALU
ALU
ADD
AND
PASS_A
INVERT_A

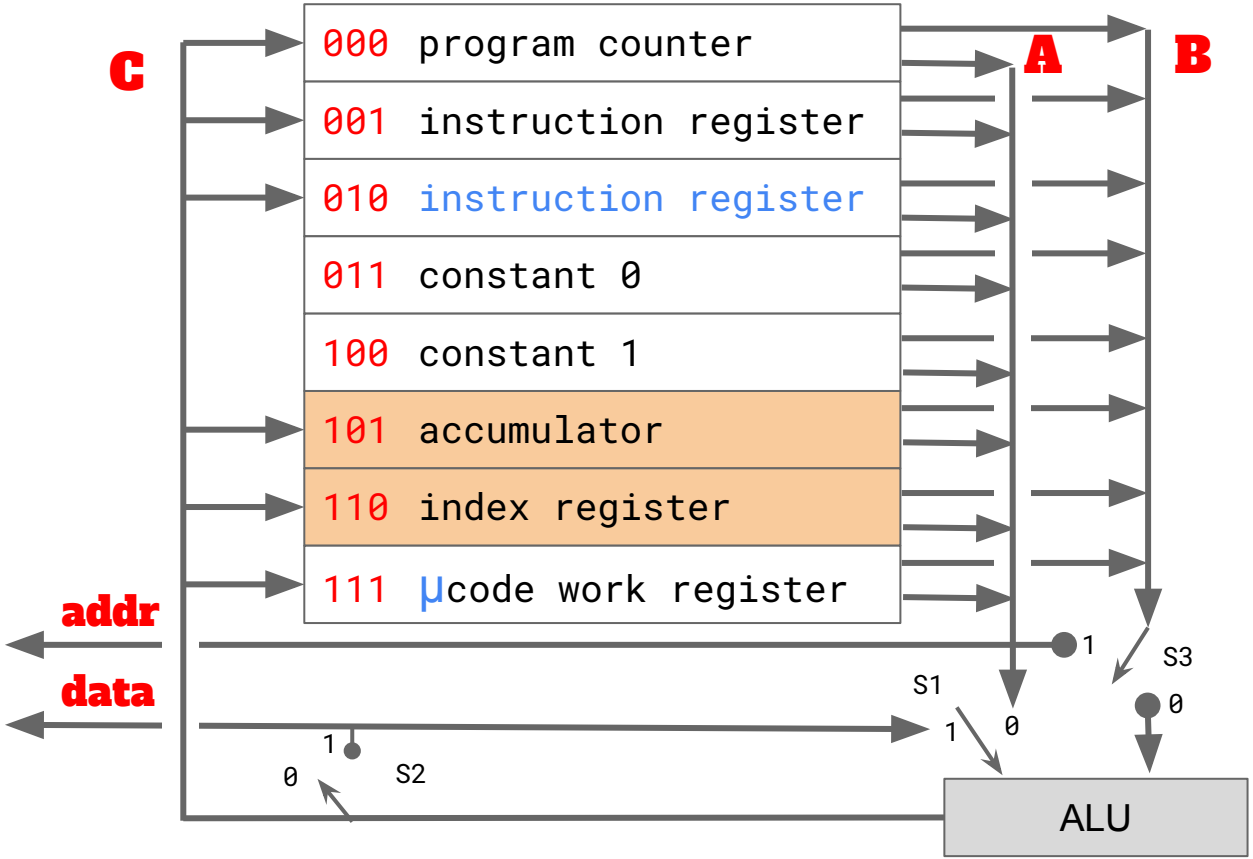


3 switches that control bus routing
Addr bus and data bus going to memory
addr = memory location (out) ; data = value (in/out)



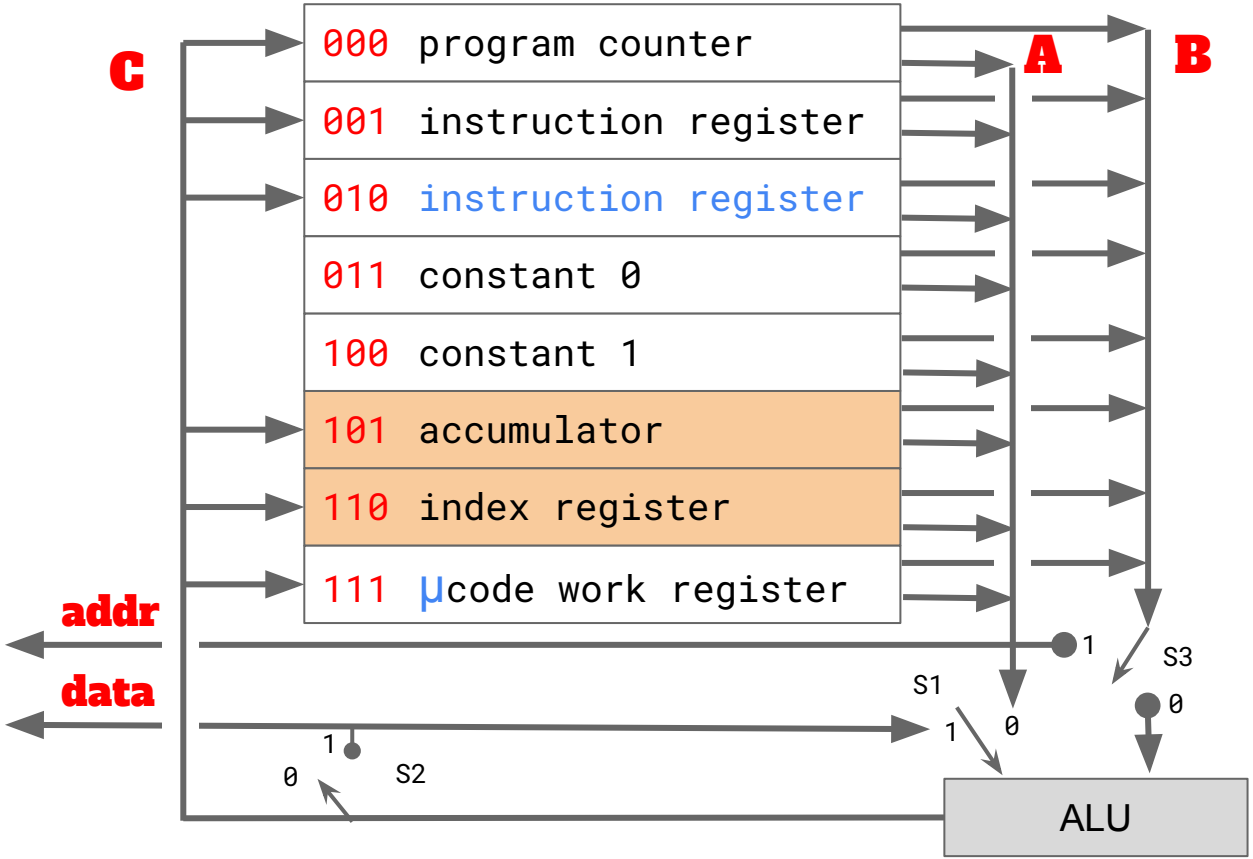
0	1-2	3	4	5	6	7-9	10-12	13-15
S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus

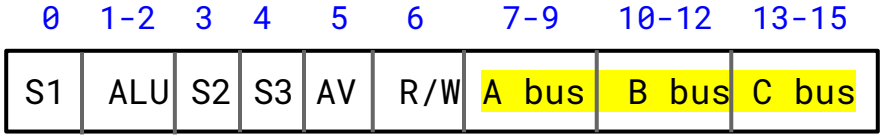
This is what a micro-instruction looks like.



00 add
01 and
10 pass A
11 not A

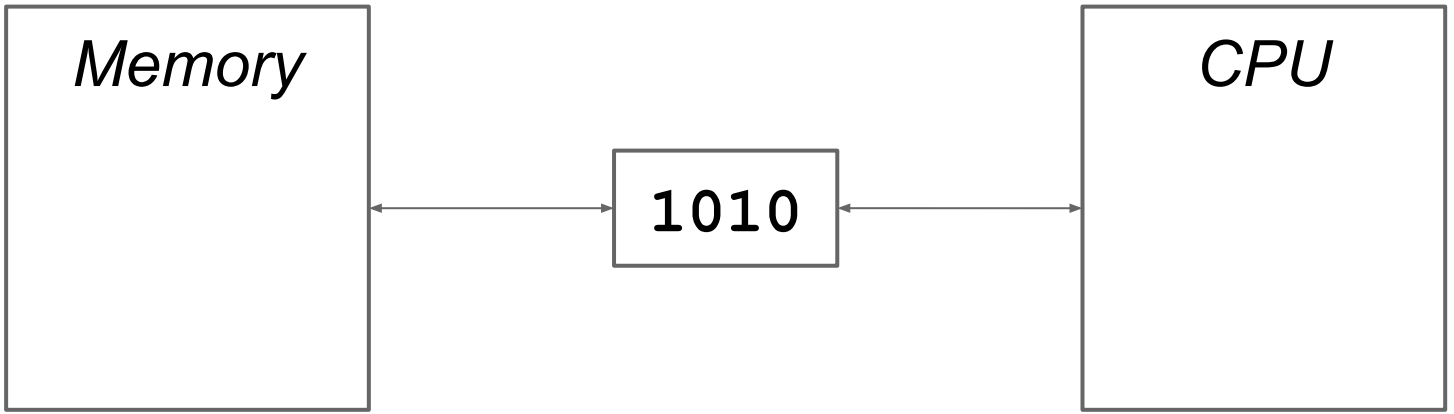
0	1-2	3	4	5	6	7-9	10-12	13-15
S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus





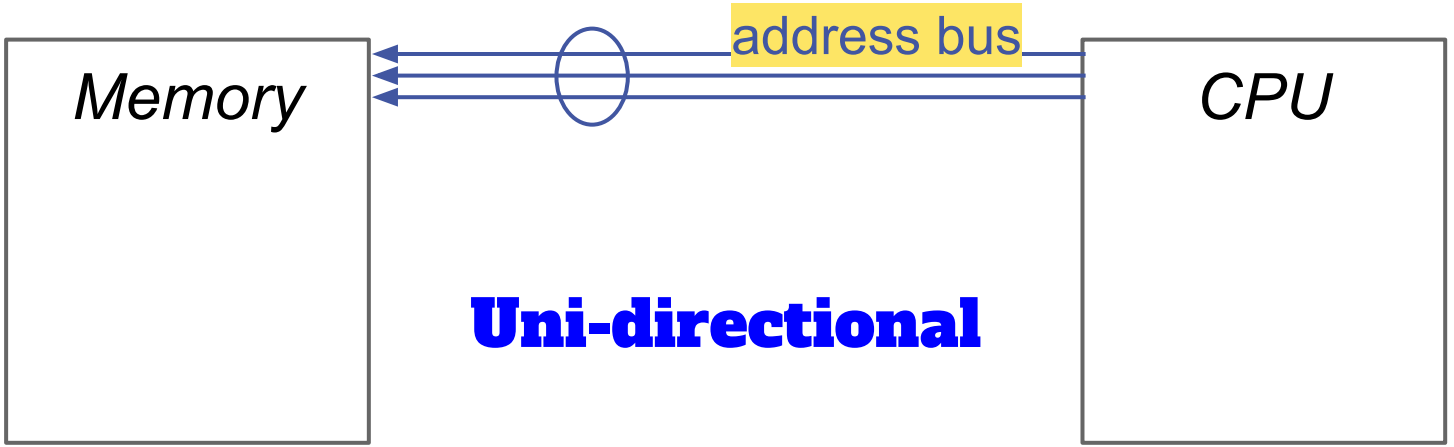
bus fields select fields
8 registers = 3 bits

Buses & Protocols



Rules for moving data between the CPU and memory

Buses & Protocols

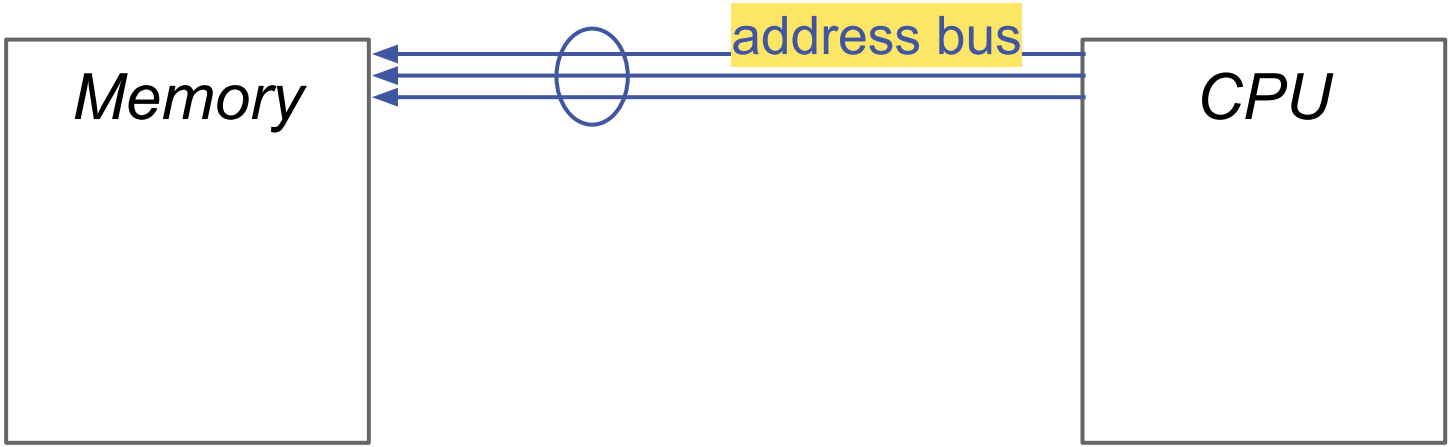


Uni-directional

Carries the desired address from the CPU to memory

For example:
20 lines address 2^{20} bytes

Buses & Protocols



32-bit system

Can address

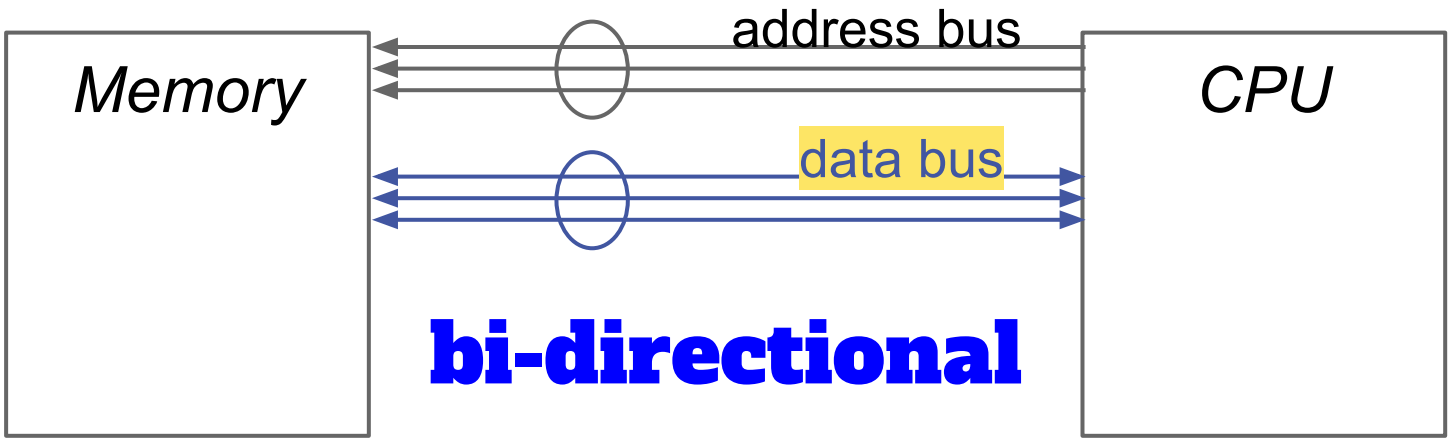
2^{32} bytes = 4 GB

16-bit system

Can address

2^{16} bytes = 64 KB

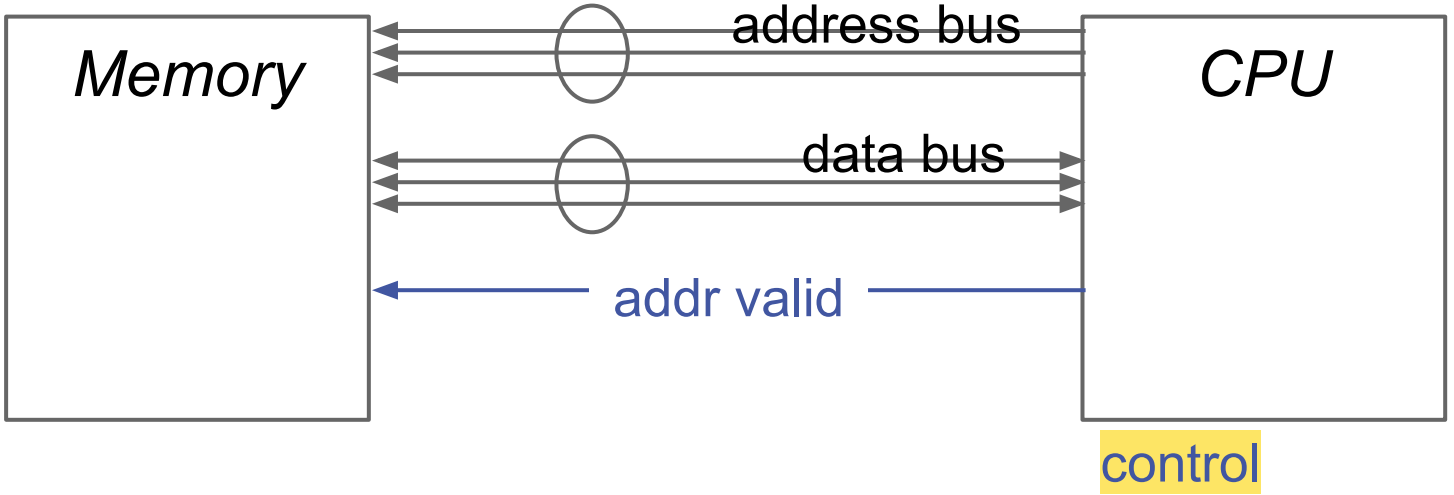
Buses & Protocols



Carries data between the CPU and memory

8 lines = 1 byte

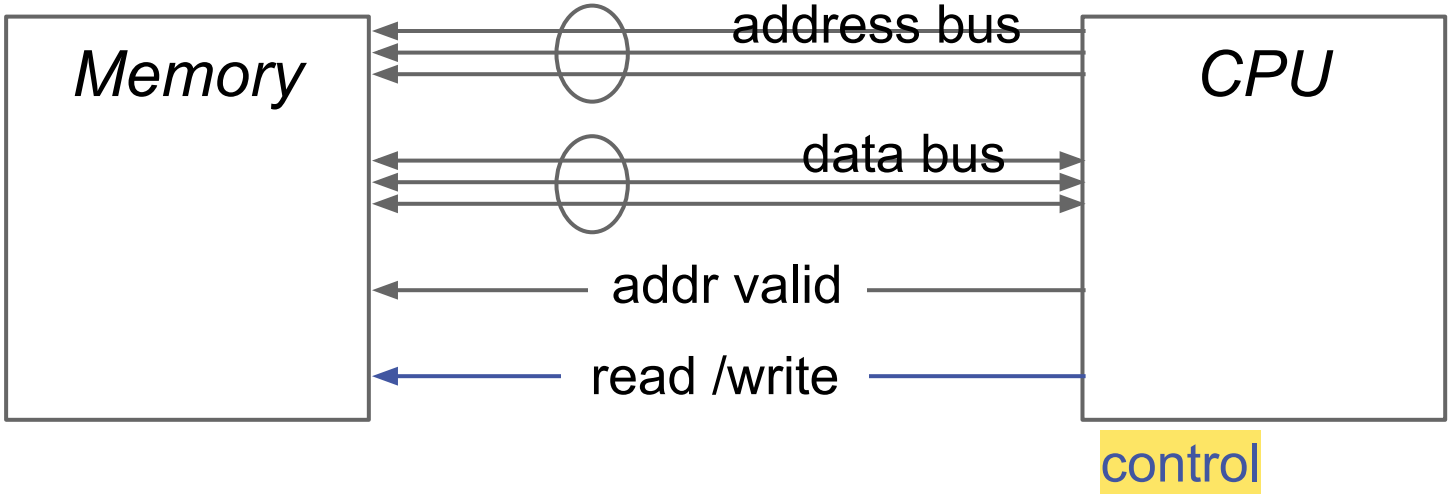
Buses & Protocols



0 = no work for memory

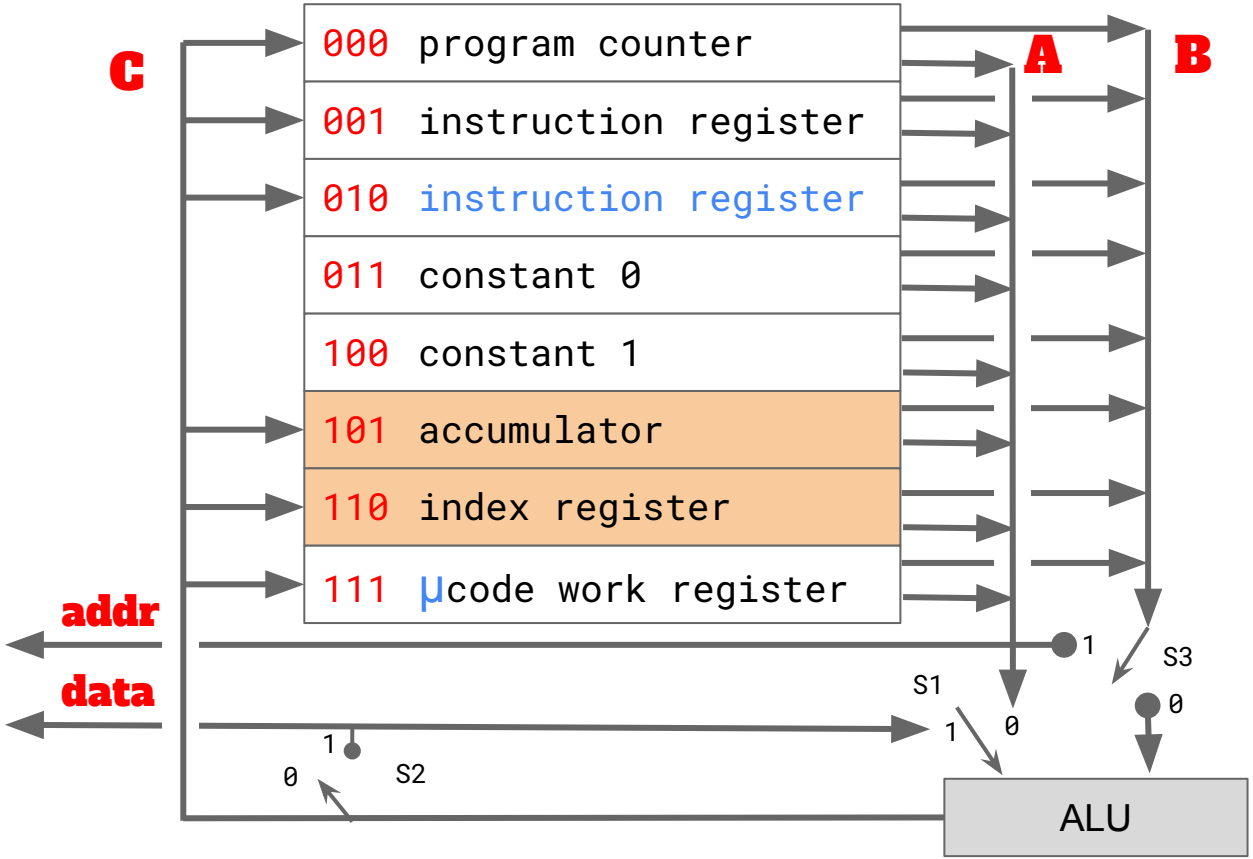
1 = work for memory

Buses & Protocols



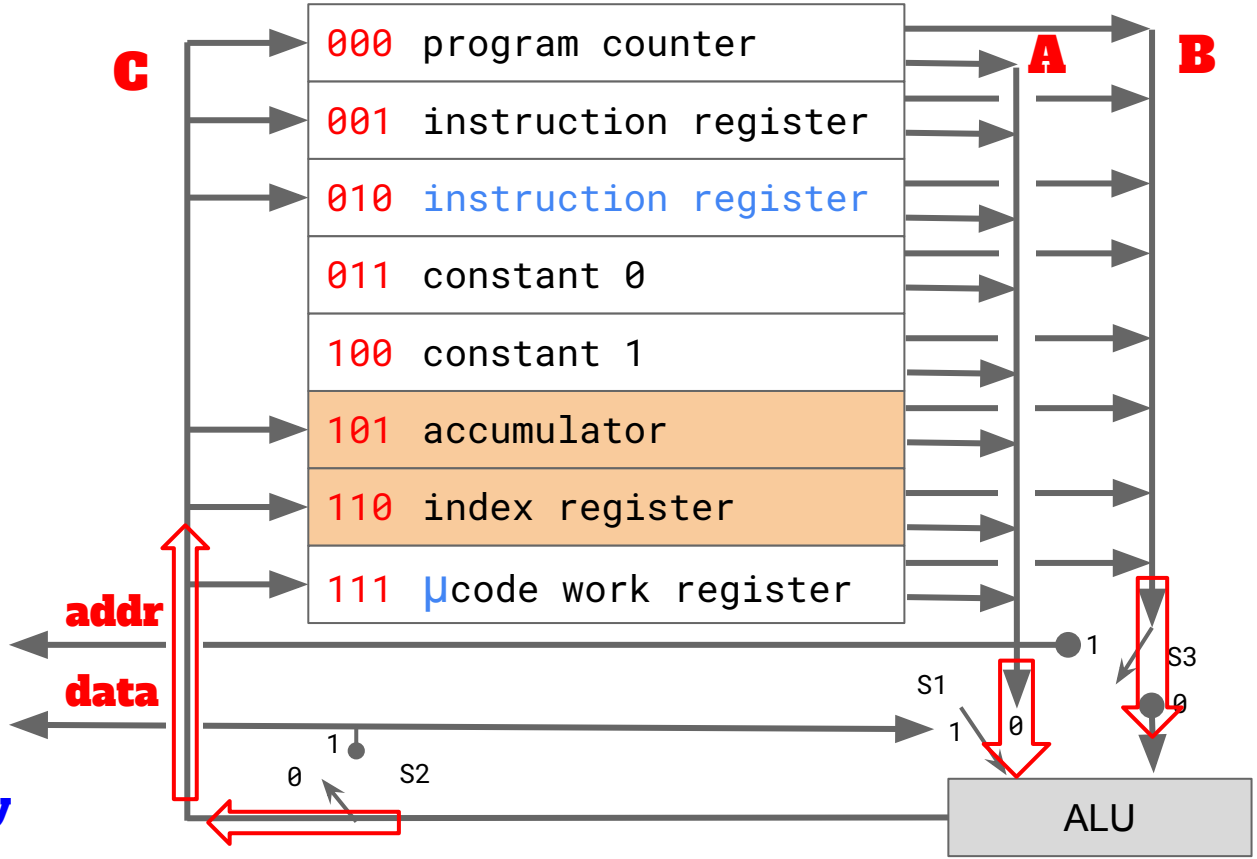
0 = read operation

1 = write operation



0	1-2	3	4	5	6	7-9	10-12	13-15
S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus

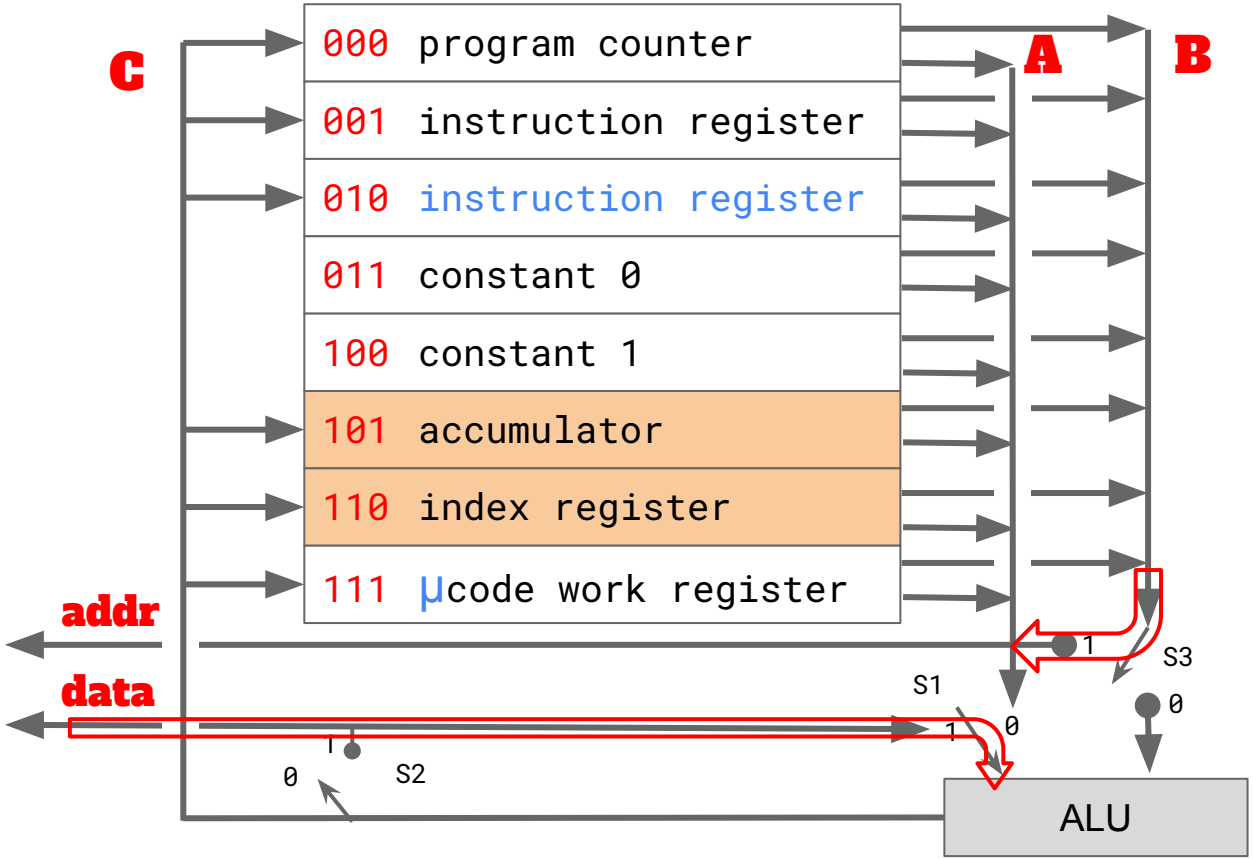
AV addr valid
R/W direction
0=R, 1=W



No memory access

S1 = 0
S2 = 0
S3 = 0

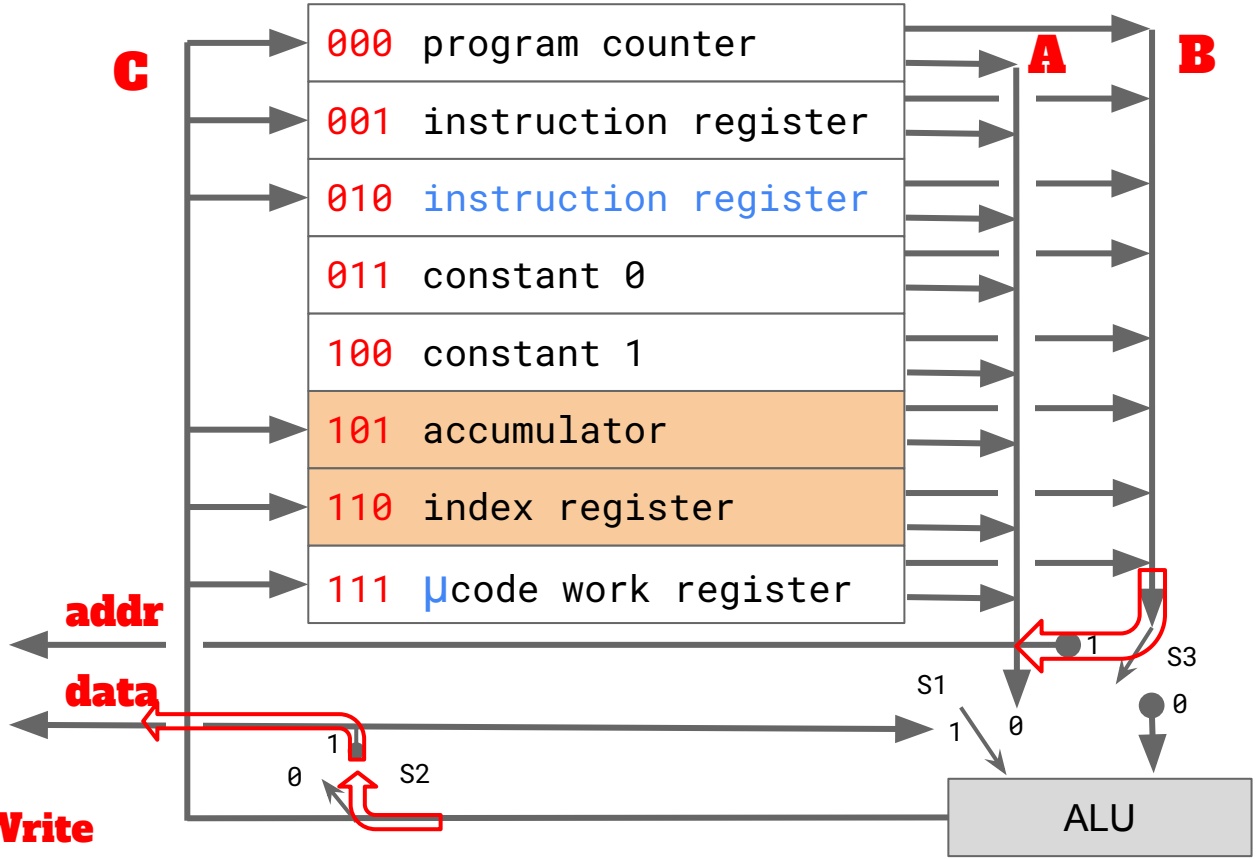
0	1-2	3	4	5	6	7-9	10-12	13-15
S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus



Read access

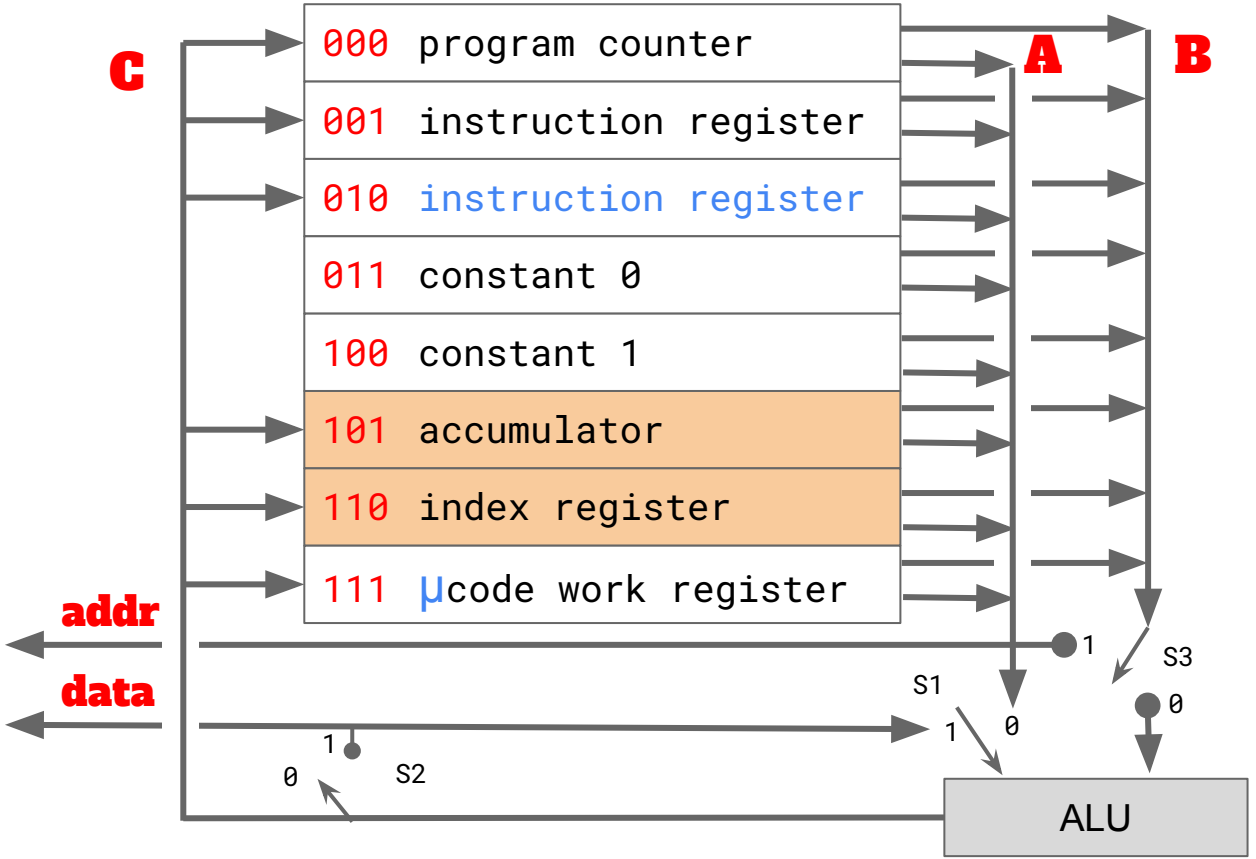
S1 = 1
S2 = 0
S3 = 1

0	1-2	3	4	5	6	7-9	10-12	13-15
S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus



Read access	Write access
S1 = 1	0
S2 = 0	1
S3 = 1	1

0	1-2	3	4	5	6	7-9	10-12	13-15
S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus

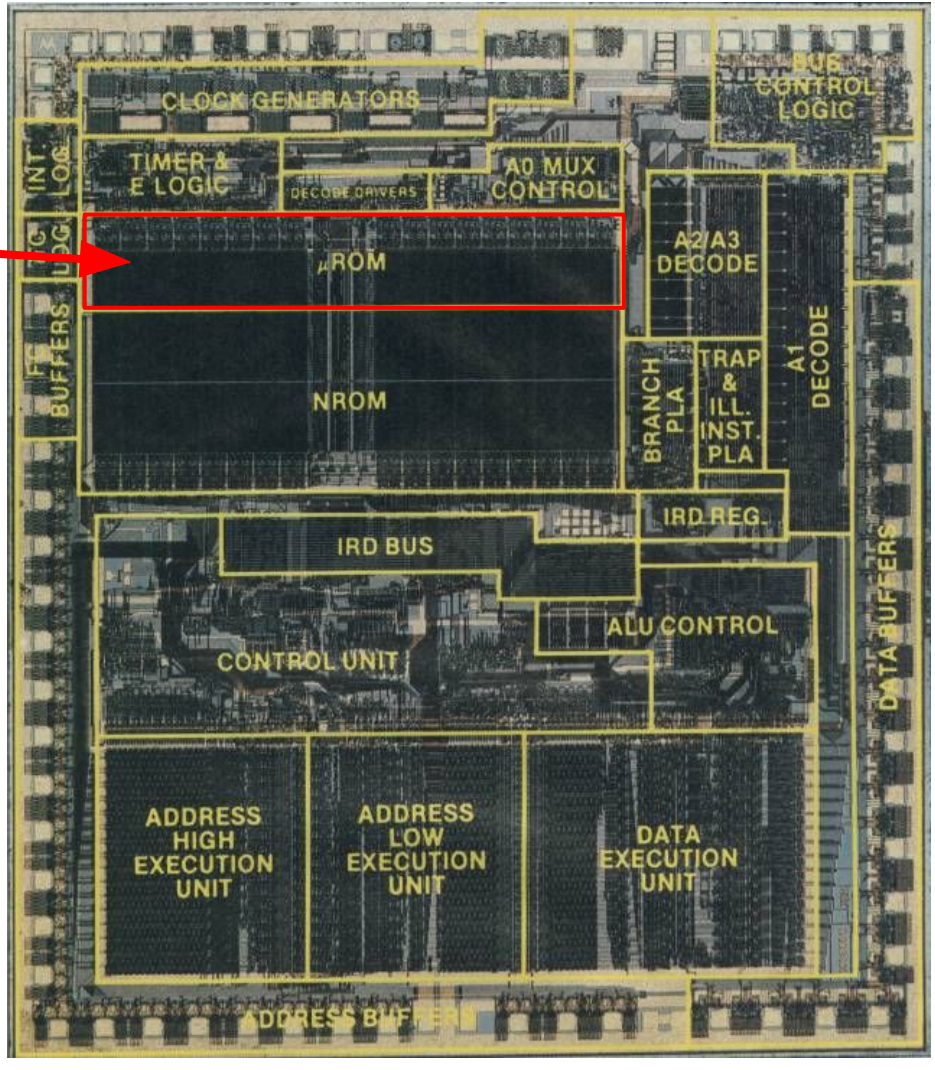


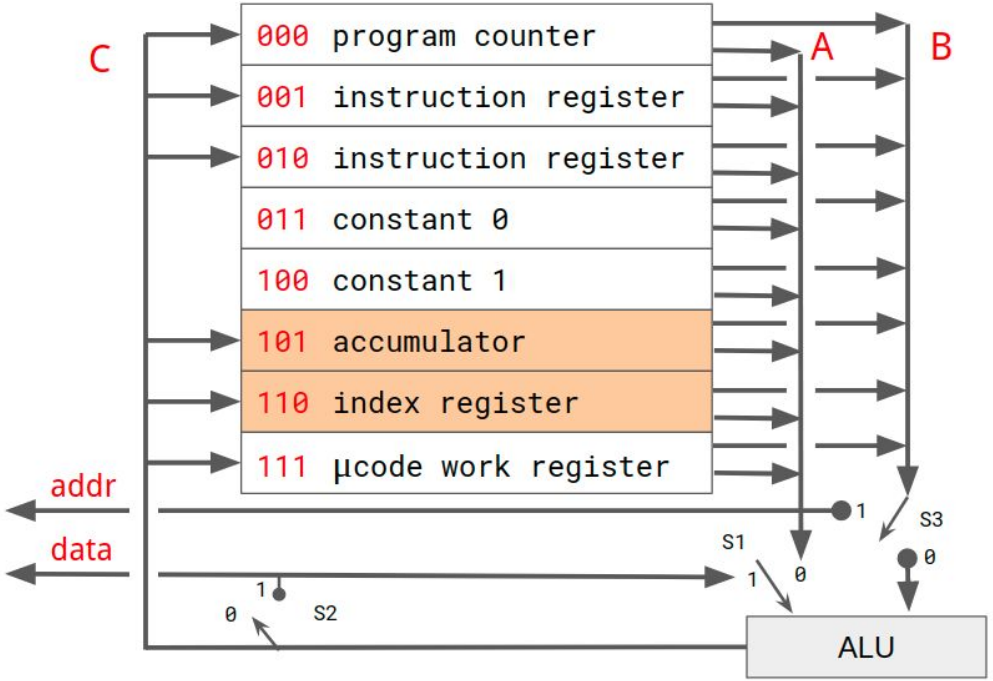
0	1-2	3	4	5	6	7-9	10-12	13-15
S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus

Operations located in μROM

Internal, read-only memory

Not part of main memory.



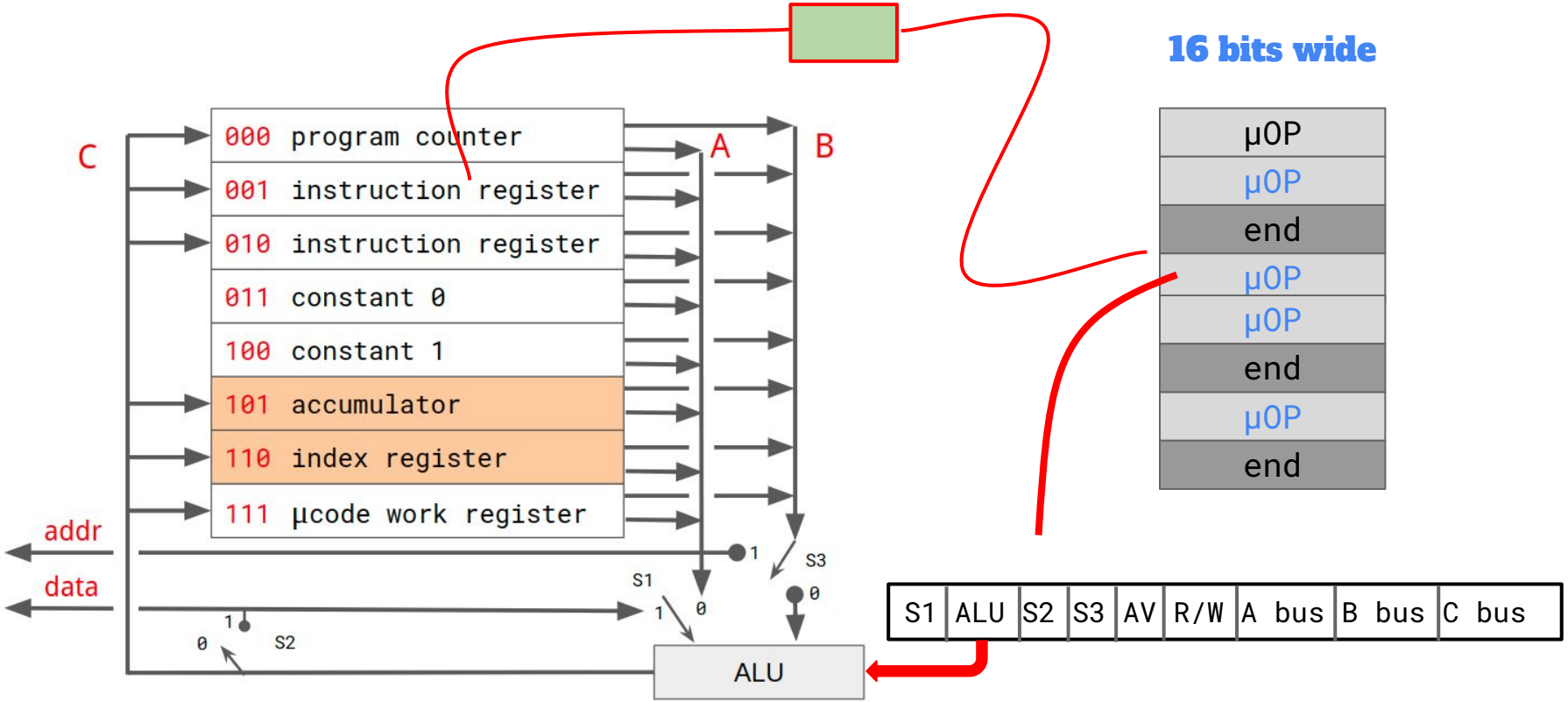


µOP store (in µROM)

16 bits wide

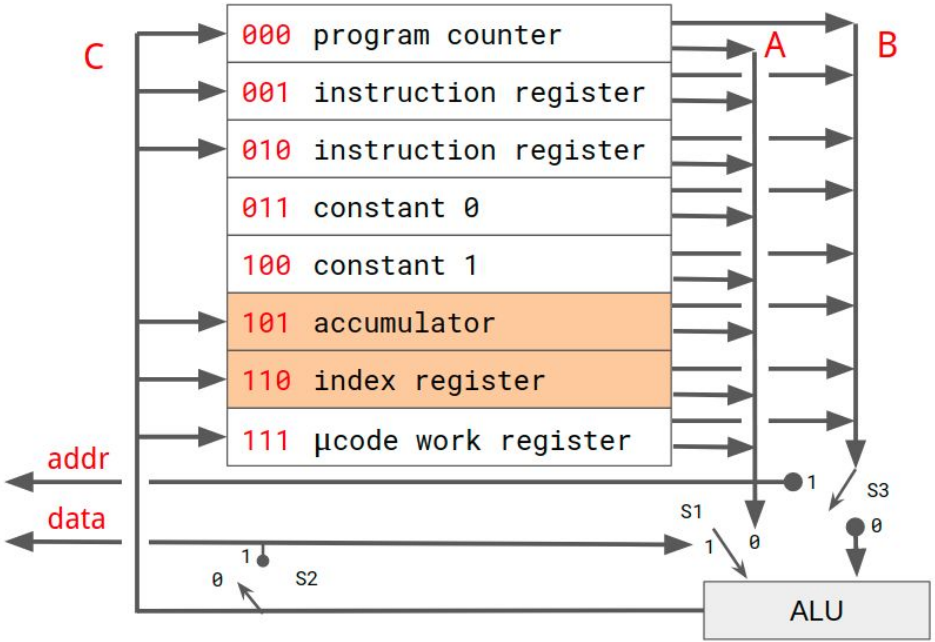
µOP
µOP
end
µOP
µOP
end
µOP
end

Instruction register contains the code (ISA)
Decoding instruction provides offset into µROM



Example 1: clear acc

acc ← 0

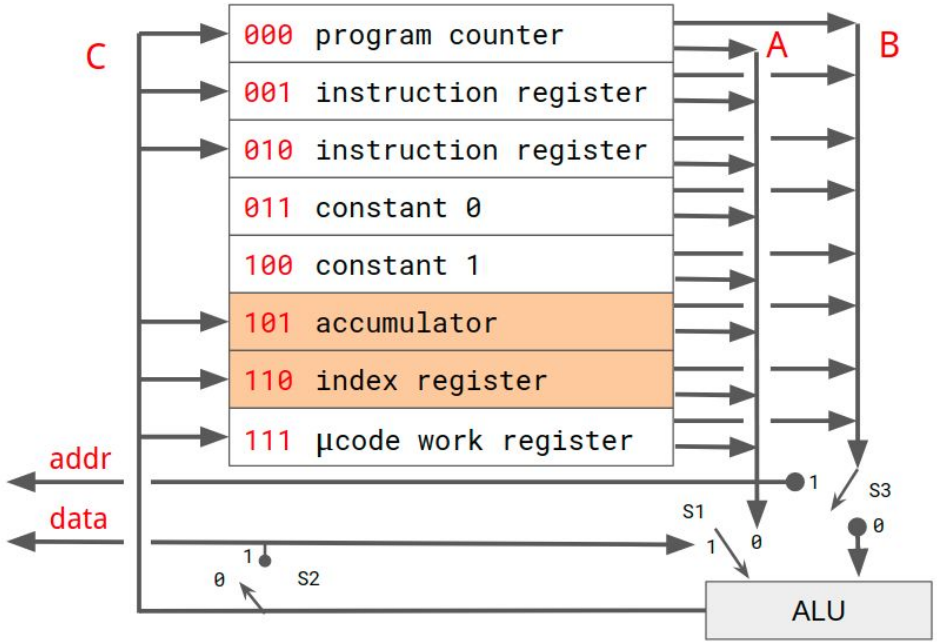


S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus

Example 1: clear acc

register 5 **register 3**

acc ← 0

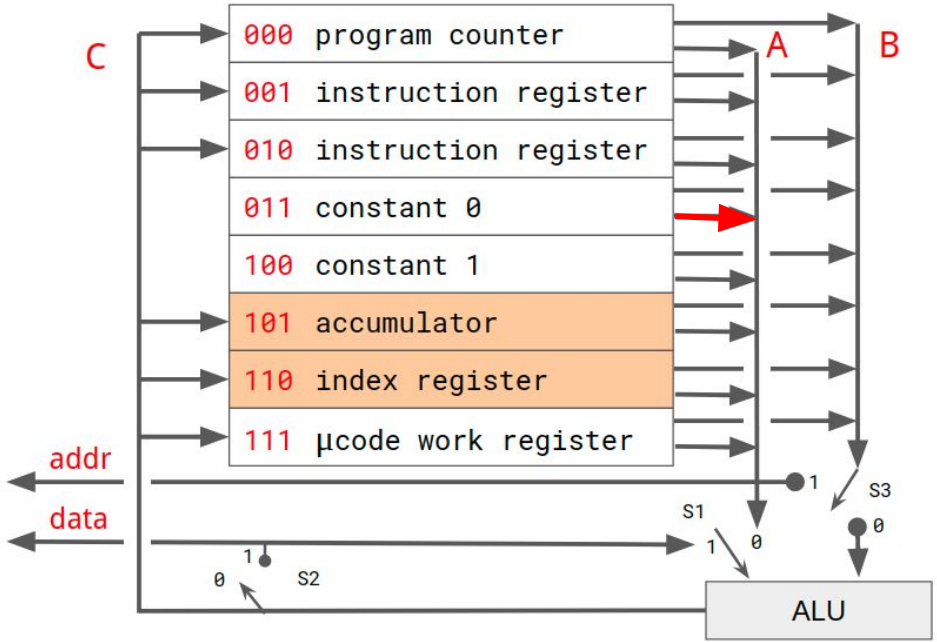


S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus

Example 1: clear acc

register 5 **register 3**

acc ← 0

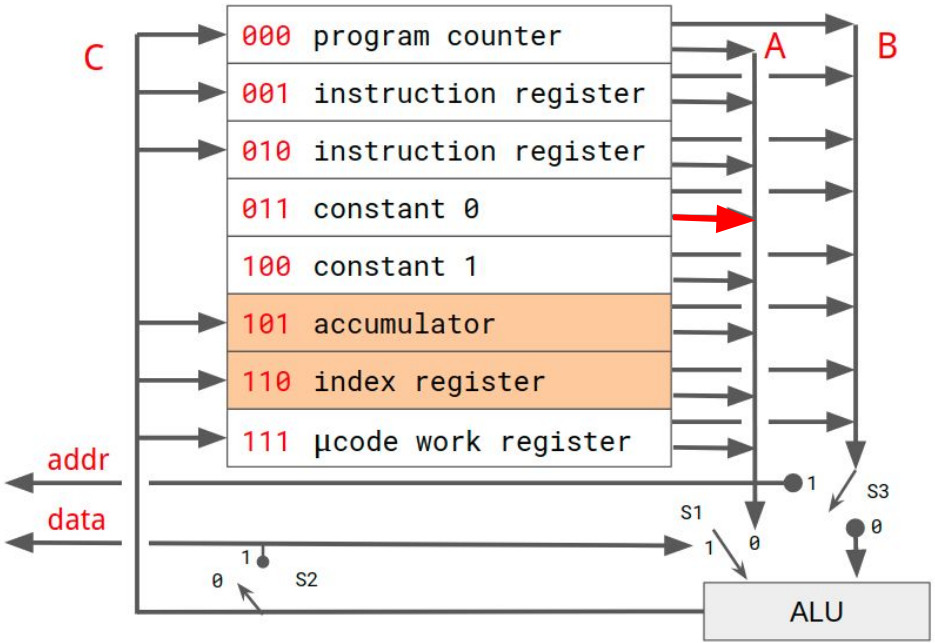


S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus

Example 1: clear acc

register 5 **register 3**

acc ← 0

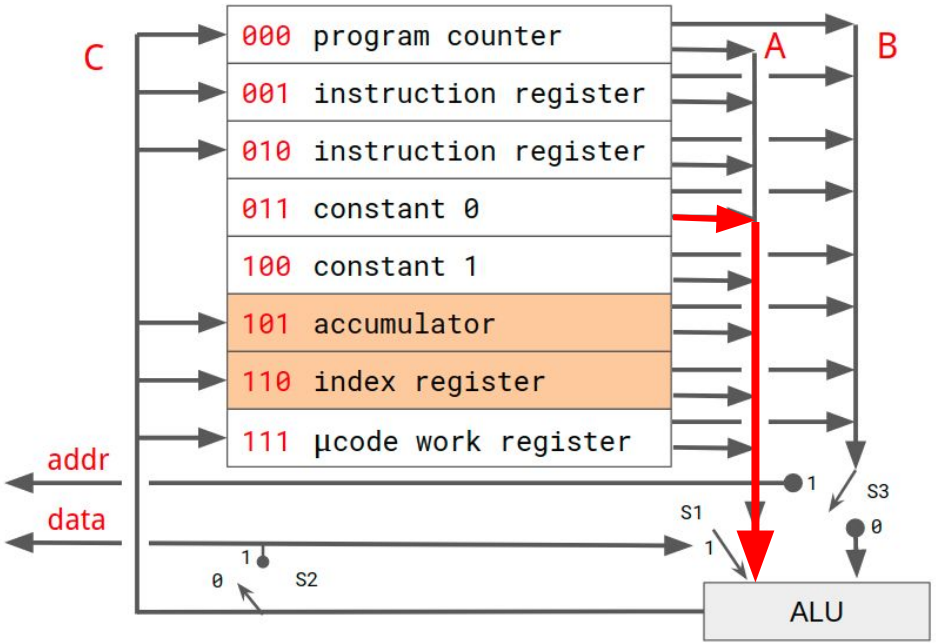


S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus
						011		

Example 1: clear acc

register 5 **register 3**

acc ← 0

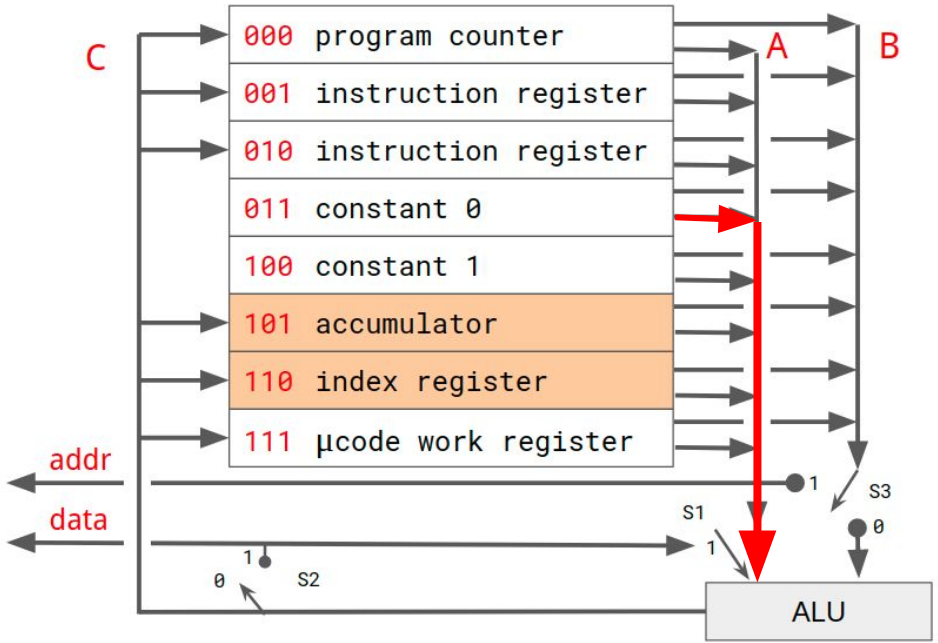


S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus
0						011		

Example 1: clear acc

register 5 **register 3**

acc ← 0



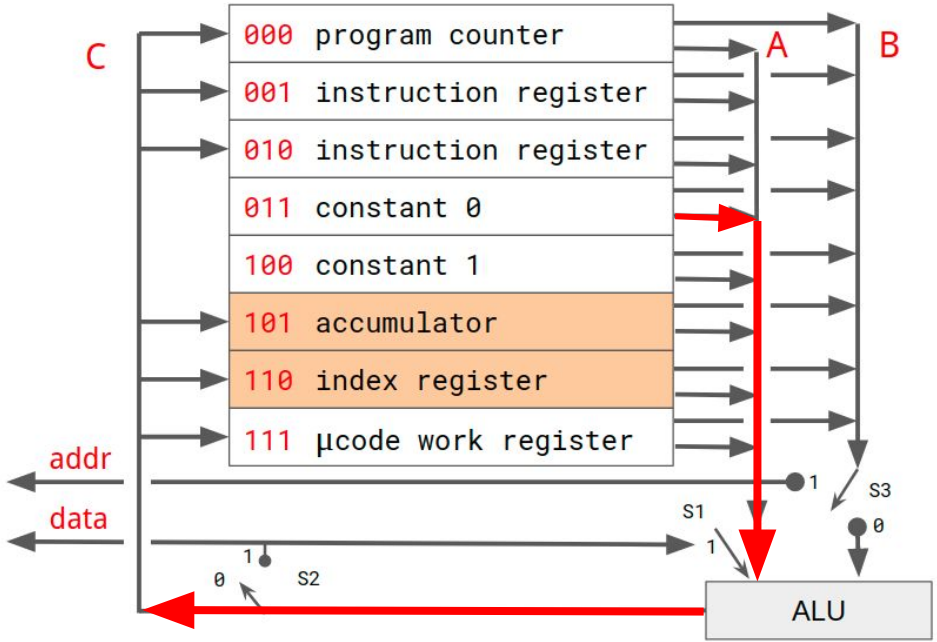
S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus
0	10					011		

pass A

Example 1: clear acc

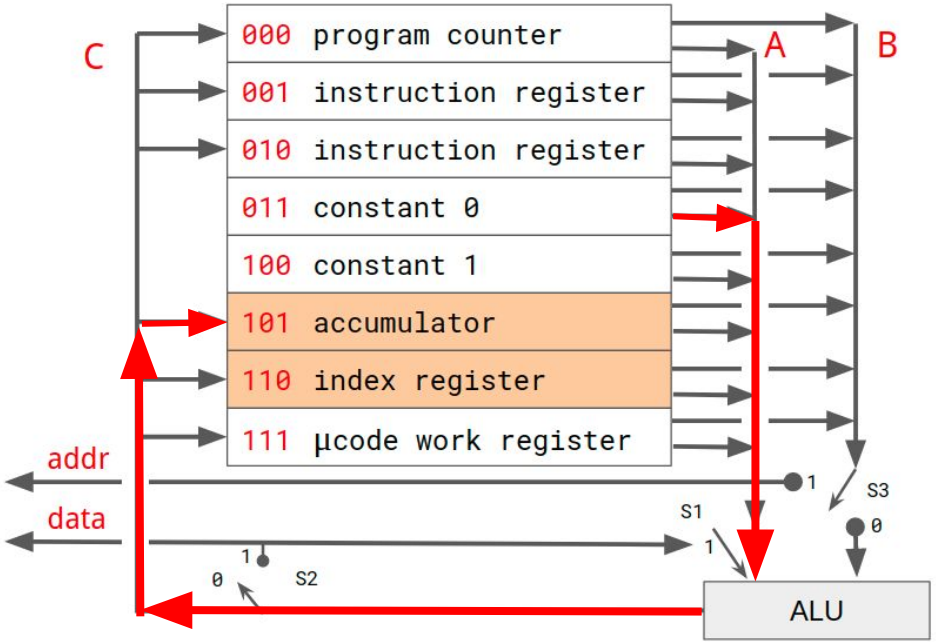
register 5 | register 3

acc ← 0



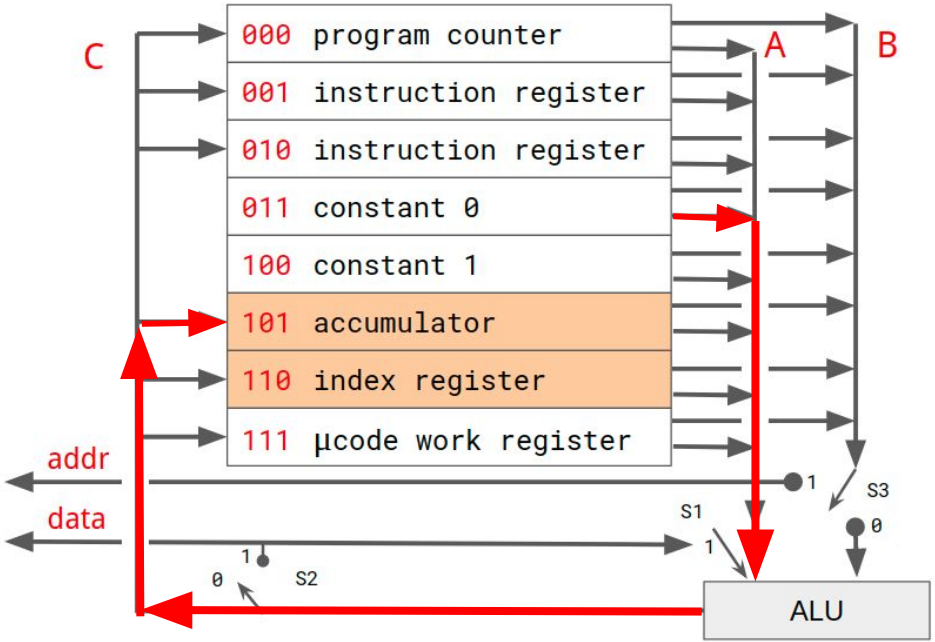
S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus
0	10	0				011		

Example 1: clear acc



S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus
0	10	0				011		101

Example 1: clear acc



S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus
0	10	0	0	0	0	011	000	101

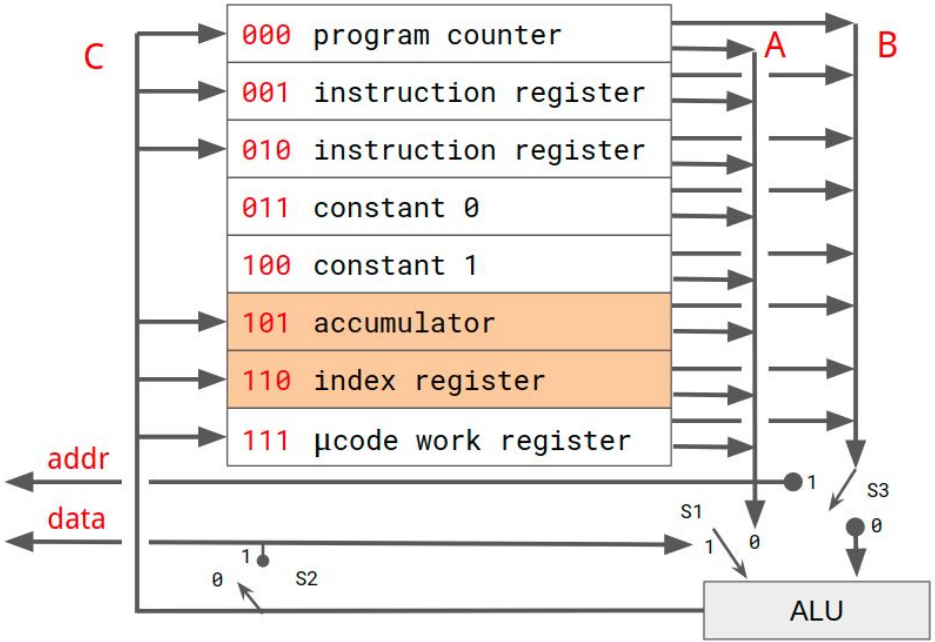
These bits in the micro-instruction don't matter.

Example 2: add acc,[var]

OPCODE **OPERAND**

```
μwr ← [var]
acc ← acc + μwr
```

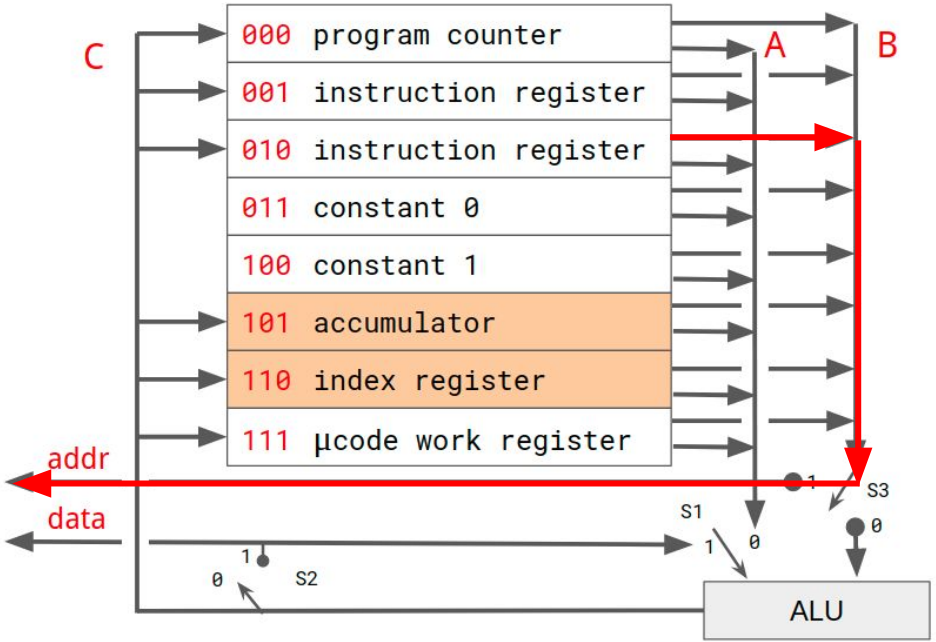
S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus



Example 2: add acc,[var]

```
μwr ← [var]
acc ← acc + μwr
```

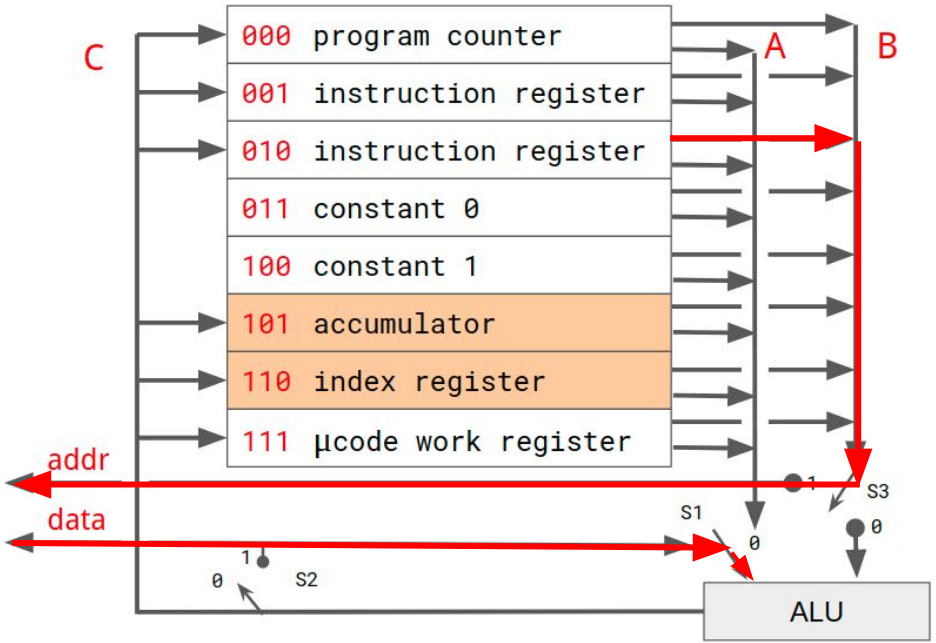
S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus
			1	1			010	



Example 2: add acc,[var]

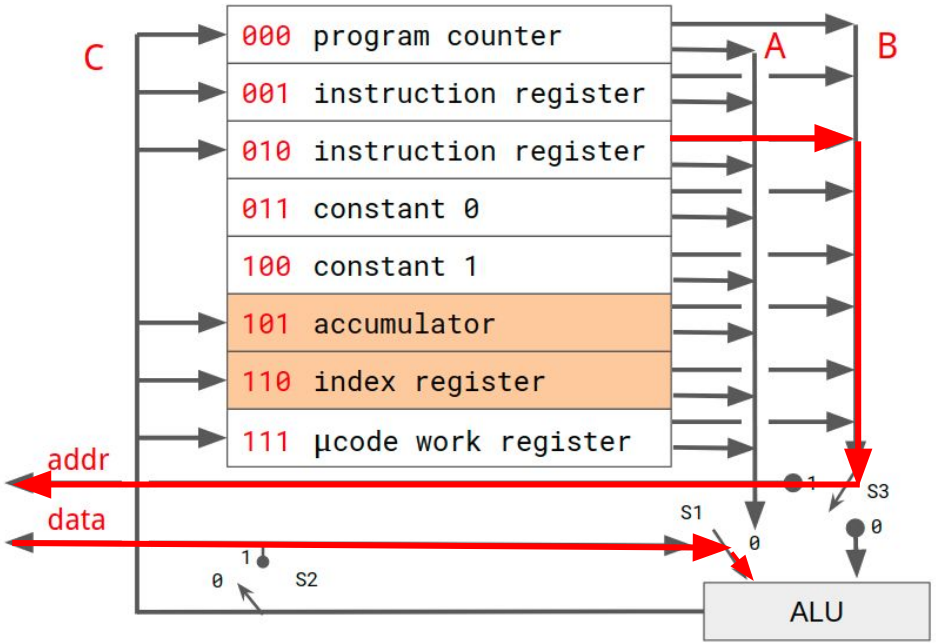
```
μwr ← [var]
acc ← acc + μwr
```

S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus
1			1	1	0		010	



Example 2: add acc,[var]

```
μwr ← [var]
acc ← acc + μwr
```

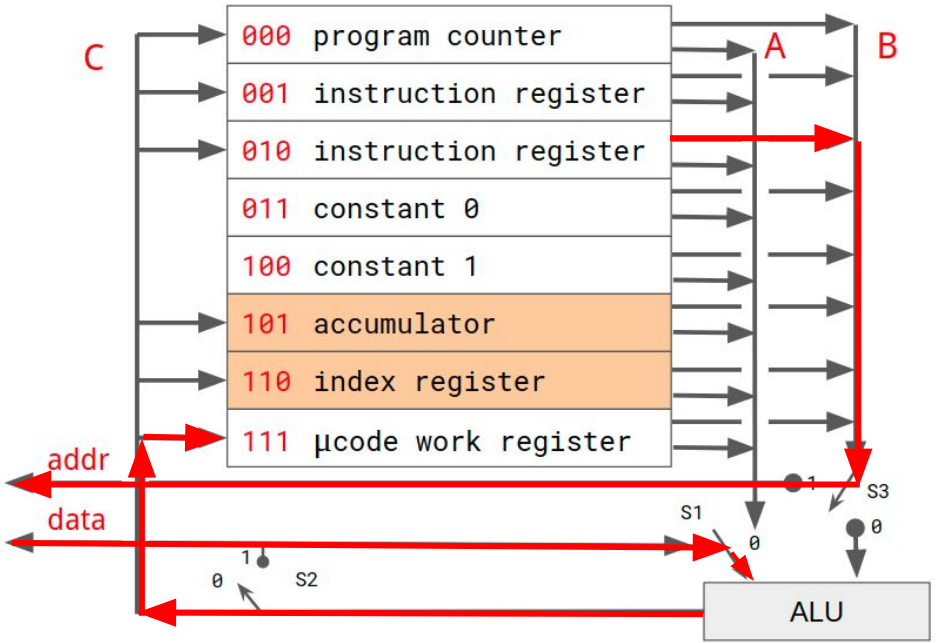


S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus
1	10		1	1	0		010	

pass A

Example 2: add acc,[var]

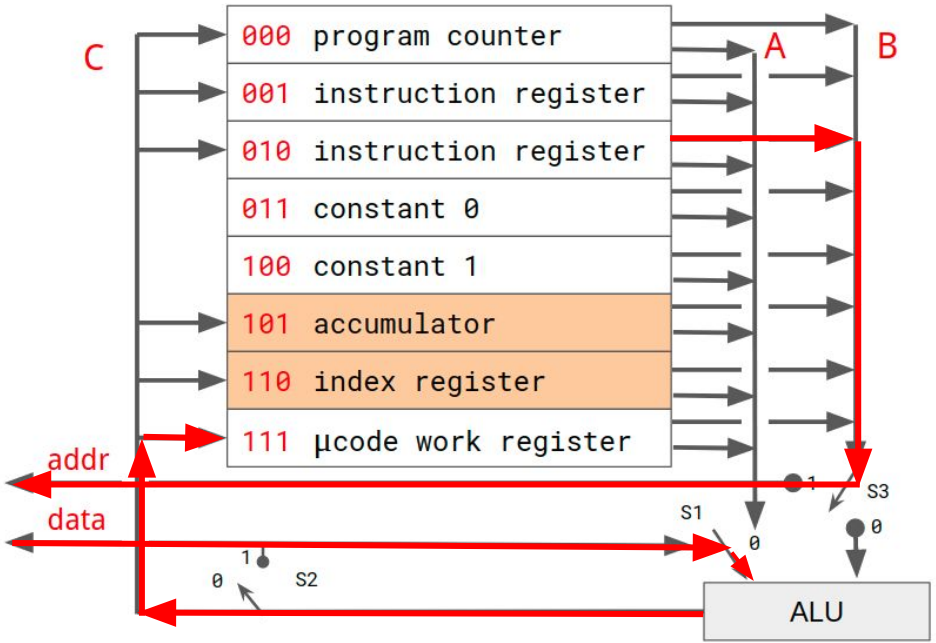
```
μwr ← [var]
acc ← acc + μwr
```



S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus
1	10	0	1	1	0		010	111

Example 2: add acc,[var]

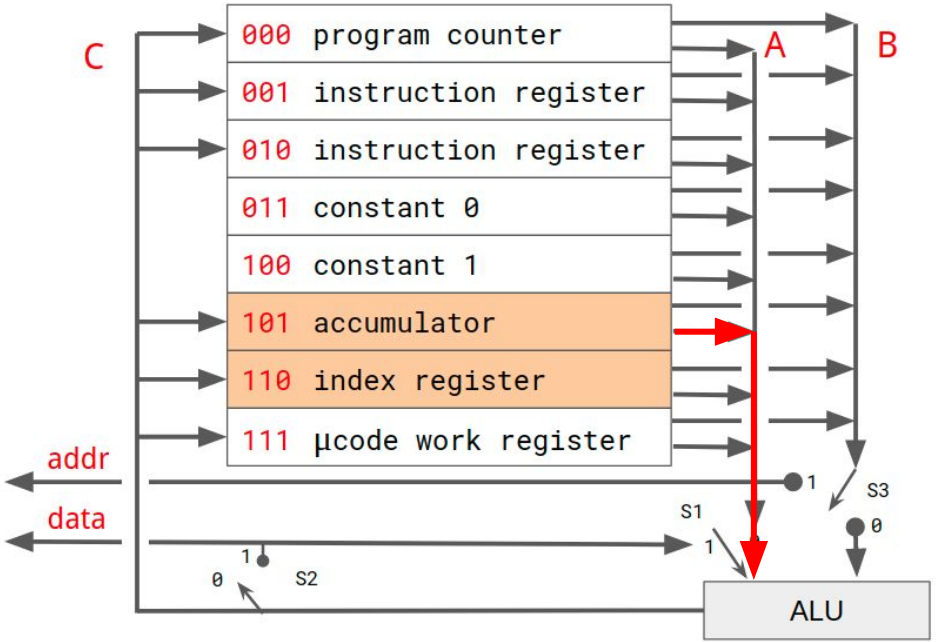
```
μwr ← [var]
acc ← acc + μwr
```



S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus
1	10	0	1	1	0	000	010	111

Example 2: add acc,[var]

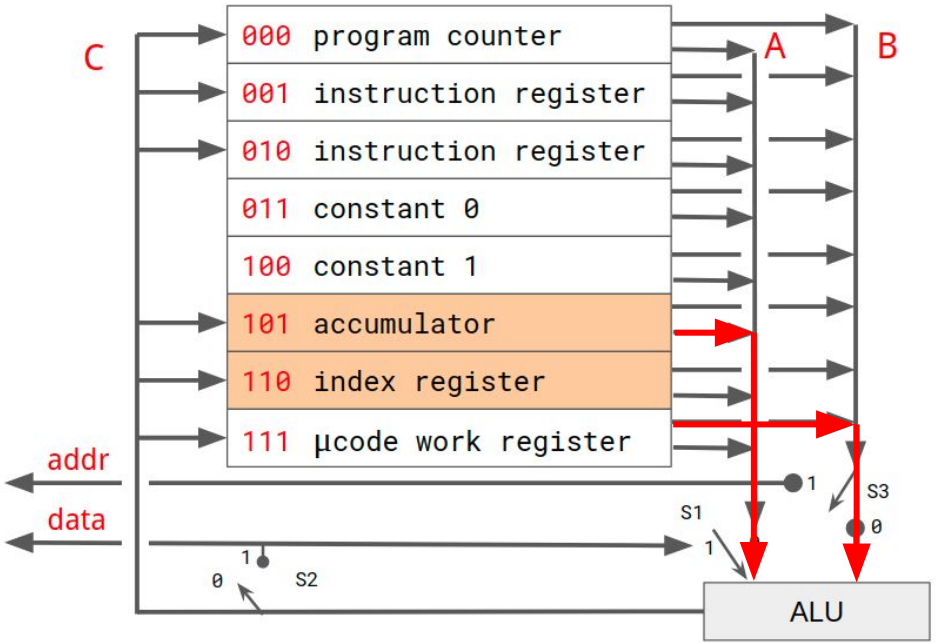
```
μwr ← [var]
acc ← acc + μwr
```



S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus
1	10	0	1	1	0	000	010	101
0						101		

Example 2: add acc,[var]

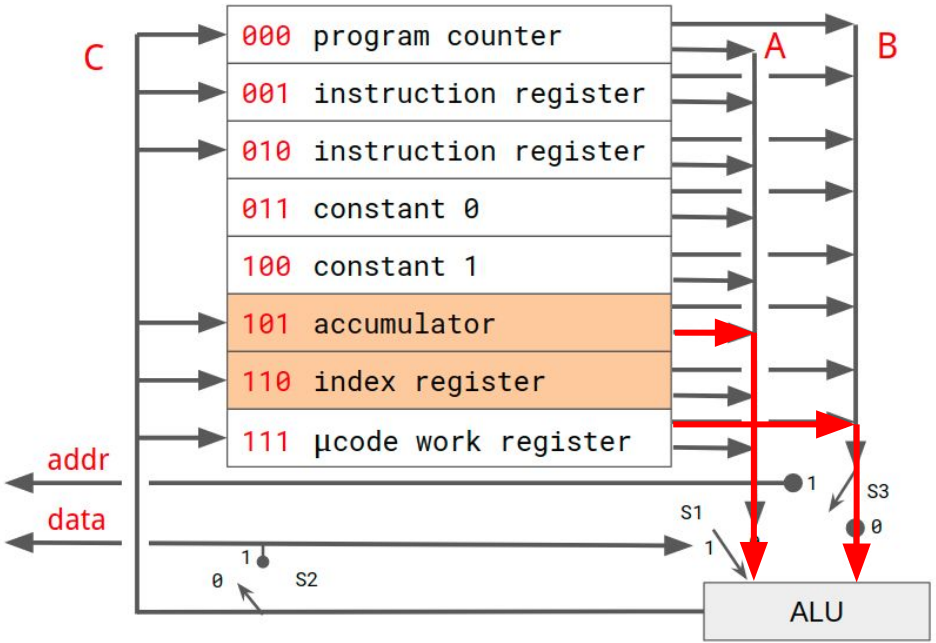
```
μwr ← [var]
acc ← acc + μwr
```



S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus
1	10	0	1	1	0	000	010	101
0			0			101	111	

Example 2: add acc,[var]

```
μwr ← [var]
acc ← acc + μwr
```

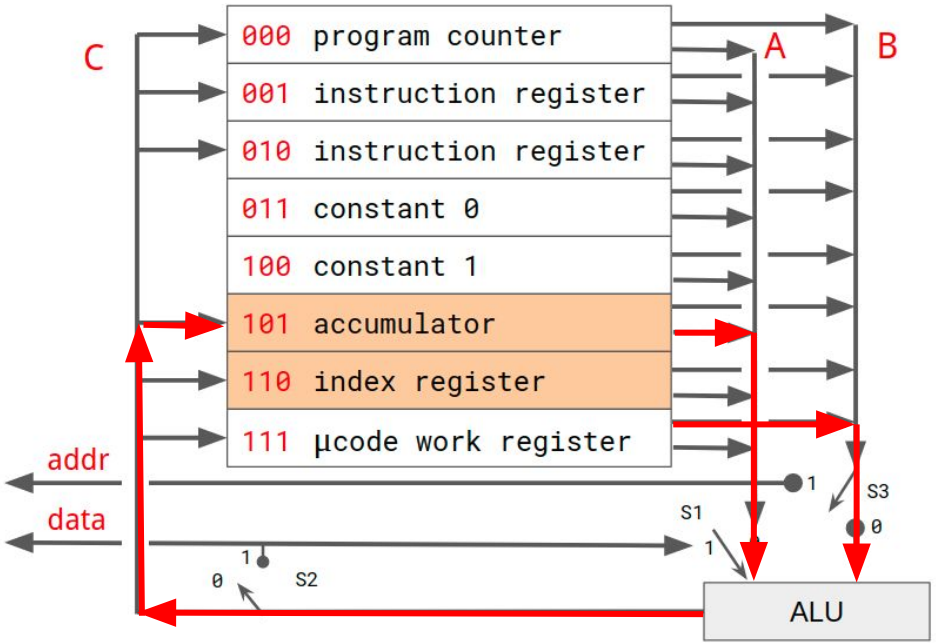


S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus
1	10	0	1	1	0	000	010	101
0	00		0			101	111	

add

Example 2: add acc,[var]

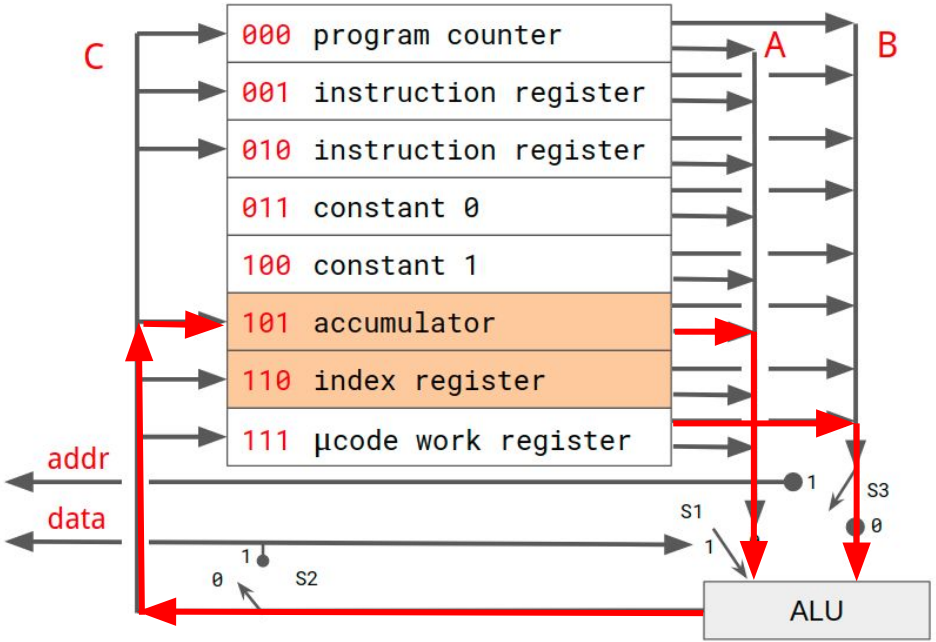
```
µwr ← [var]
acc ← acc + µwr
```



S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus
1	10	0	1	1	0	000	010	101
0	00	0	0			101	111	101

Example 2: add acc,[var]

```
μwr ← [var]
acc ← acc + μwr
```



S1	ALU	S2	S3	AV	R/W	A bus	B bus	C bus
1	10	0	1	1	0	000	010	101
0	00	0	0	0	0	101	111	101

Example 3: dec acc

- Simple
  - `acc = acc - 1 ?`

Example 3: dec acc

- Simple
  - `acc = acc - 1` ?
  - No ... not really.
- ALU does not subtract
- We can use two's complement addition
  - `acc = acc - 1 = acc + (-1)`

Example 3: dec acc

- Simple
  - `acc = acc - 1` ?
  - No … not really.
- ALU does not subtract
- We can use two's complement addition
  - `acc = acc - 1 = acc + (-1)`
  - Oops. No constant for -1
  - OK. We'll have to create it.

Example 3: dec acc

- Plan
  - Create -1
  - `acc = acc + (-1)`
- Where to create/store -1?
  - µwr
  - -1 = not 1 + 1

- µOPs

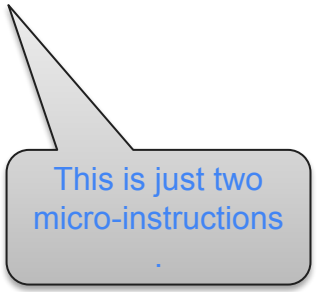
```
µwr = not 1
µwr = µwr + 1
acc = acc + µwr
```

Example 3: dec acc

- 3 instructions
- We can do better

```
          1  ;00000001
μwr = not 1  ;11111110
μwr = μwr + 1    ;11111111
acc = acc + μwr
```

```
          0  ;0000000
μwr = not 0  ;1111111
acc = acc + μwr
```



Example 4: sub [var],acc

- Code

```
var = var + (-acc)
```

- Steps
  - Fetch [var] — into μwr
  - Build -acc — into μwr
  - Calc [var] + (-acc)
  - Write [var]

There is only 1 work register

Example 4: sub [var],acc

- Code

```
var = var + (-acc)
```

- Steps
  - Fetch [var] — into µwr
  - Build -acc — into µwr
  - Calc [var] + (-acc)
  - Write [var]

- Requires
  - 5 operations
  - 2 work registers
- What will be the 2nd register?
  - After decoding OPCODE
  - OPCODE is no longer needed
  - Use IR 001

Example 4: sub [var],acc

- Code

```
var = var + (-acc)
```

- Steps
  - Fetch [var] — into μwr
  - Build -acc — into μwr
  - Calc [var] + (-acc)
  - Write [var]

```
read: μwr = [var]
IR001 = not acc
IR001 = IR001 + 1 ; -acc
μwr   = μwr + IR001
write: [var] = μwr
```

Example 4: sub [var],acc

- 5 instructions
- Can do it in 3
- Two facts
  - -x = not(x) + 1 ⇒ not(x) = -x - 1
  - Data read/write can be
    - Passed
    - Inverted
    - ~~Added~~
    - ~~Anded~~

```
read: μwr = [var]
IR001 = not acc
IR001 = IR001 + 1 ; -acc
μwr   = μwr + IR001
write: [var] = μwr
```

Example 4: sub [var],acc

```
read: μwr = not [var]
μwr = μwr + acc
write: [var] = not μwr
```

```
μwr = -var - 1
μwr = -var - 1 + acc
[var] = not(-var - 1 + acc)
      = -(-var - 1 + acc) - 1
      = var + 1 - acc - 1
      = var - acc
```

Summary

- Microcode gives hardware its personality (defines what machine instructions do)
- Different microprograms allow the same physical chip to execute widely varying machine instructions
- Can create optimized machine instructions for an application like word processing or an industry that runs specialized programs
- Clever micro programmers can make a big difference in chip performance