

CS-200

Computer Architecture

Part 1a. Instruction Set Architecture

ISA Reminder, Assembly Language, and Compilers

Paolo Ienne
<paolo.ienne@epfl.ch>

High-Level Languages

```
int data    = 0x00123456;
int result  = 0;
int mask    = 1;
int count   = 0;
int temp    = 0;
int limit   = 32;
do {
    temp    = data & mask;
    result  = result + temp;
    data    = data >> 1;
    count   = count + 1;
} while (count != limit);
```

Variables have (hopefully)
expressive names

Variables have types

Computation is expressed as
math-like formulas

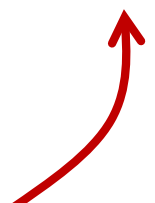
Control flow (e.g., loops) is expressed
through intuitive constructs

High-Level Languages



```
int data    = 0x00123456;
int result  = 0;
int mask    = 1;
int count   = 0;
int temp    = 0;
int limit   = 32;
do {
    temp    = data & mask;
    result  = result + temp;
    data    = data >> 1;
    count   = count + 1;
} while (count != limit);
```

name	value
data	0x00123456
result	0
mask	1
count	...
temp	
limit	
...	
my_float	3.141529
a_string	Hello world!



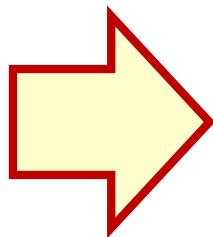
Unlimited variables



Any type supported
by the language

Assembly Language

```
int data    = 0x00123456;
int result  = 0;
int mask    = 1;
int count   = 0;
int temp    = 0;
int limit   = 32;
do {
    temp    = data & mask;
    result  = result + temp;
    data    = data >> 1;
    count   = count + 1;
} while (count != limit);
```



```
0          li    x1, 0x00123456
1          li    x2, 0
2          li    x3, 1
3          li    x4, 0
4          li    x5, 0
5          li    x6, 32
6  loop:  and    x5, x1, x3
7          add    x2, x2, x5
8          srli   x1, x1, 1
9          addi   x4, x4, 1
10         bne    x4, x6, loop
```

Assembly Language

Much more rigid format:
A sequence of numbered instructions

An *opcode* defines
the effect of the instruction

Predefined “variable” names

Only one form of control flow:
branch/jump to a particular instruction

```
0      li    x1, 0x00123456
1      li    x2, 0
2      li    x3, 1
3      li    x4, 0
4      li    x5, 0
5      li    x6, 32
6  loop: and   x5, x1, x3
7      add   x2, x2, x5
8      srli  x1, x1, 1
9      addi  x4, x4, 1
10     bne   x4, x6, loop
```

Assembly Language

	value
x0	0
x1	0x00123456
x2	0
x3	1
x4	...
...	
...	
x30	...
x31	...

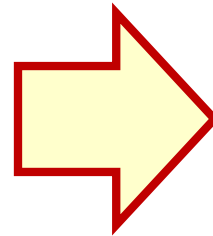
Only 32
“variables”

Only numbers
(32- or 64-bit)

```
0      li    x1, 0x00123456
1      li    x2, 0
2      li    x3, 1
3      li    x4, 0
4      li    x5, 0
5      li    x6, 32
6  loop: and    x5, x1, x3
7      add    x2, x2, x5
8      srli   x1, x1, 1
9      addi   x4, x4, 1
10     bne    x4, x6, loop
```

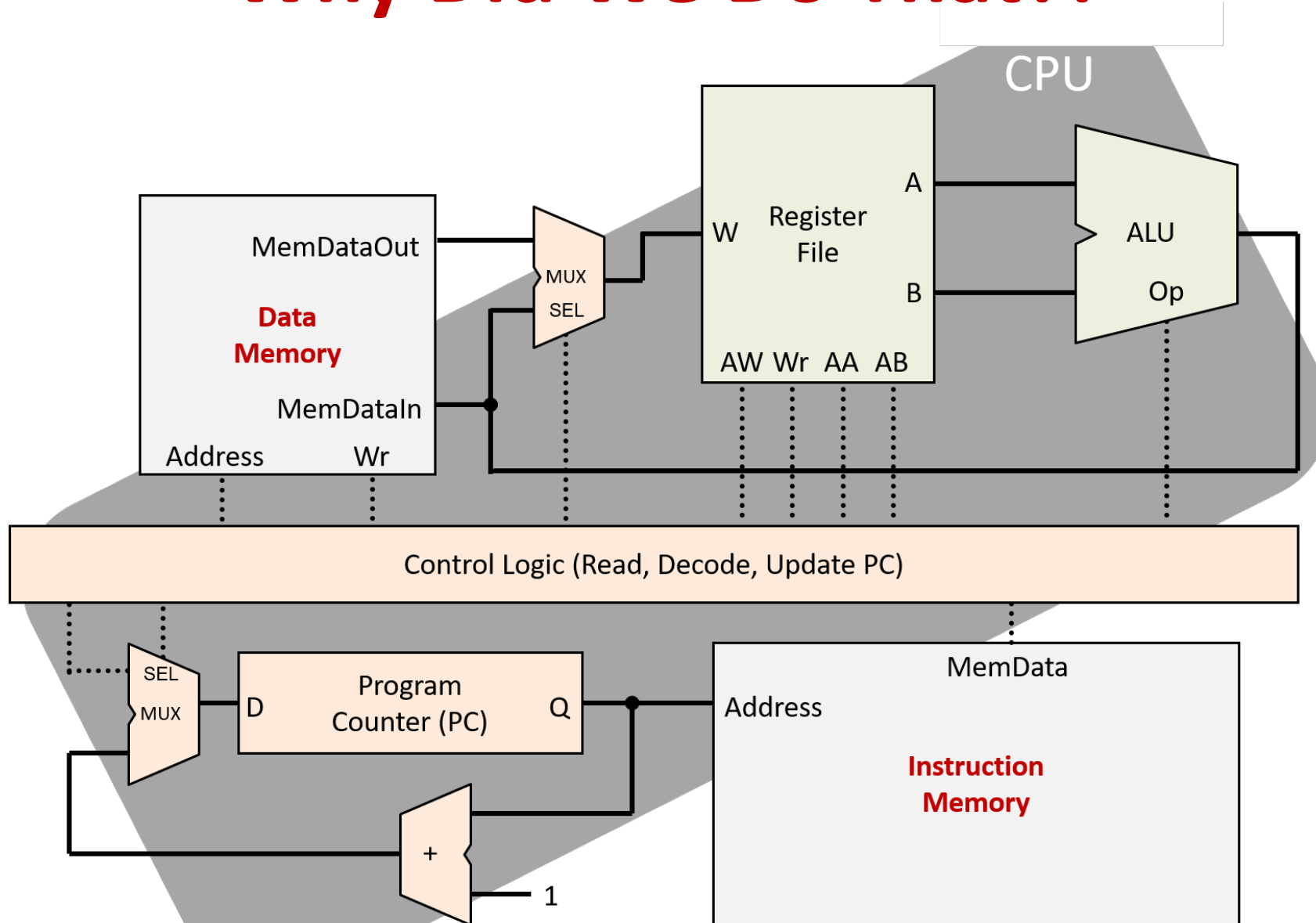
Why?

```
int data    = 0x00123456;
int result  = 0;
int mask    = 1;
int count   = 0;
int temp    = 0;
int limit   = 32;
do {
    temp    = data & mask;
    result  = result + temp;
    data    = data >> 1;
    count   = count + 1;
} while (count != limit);
```



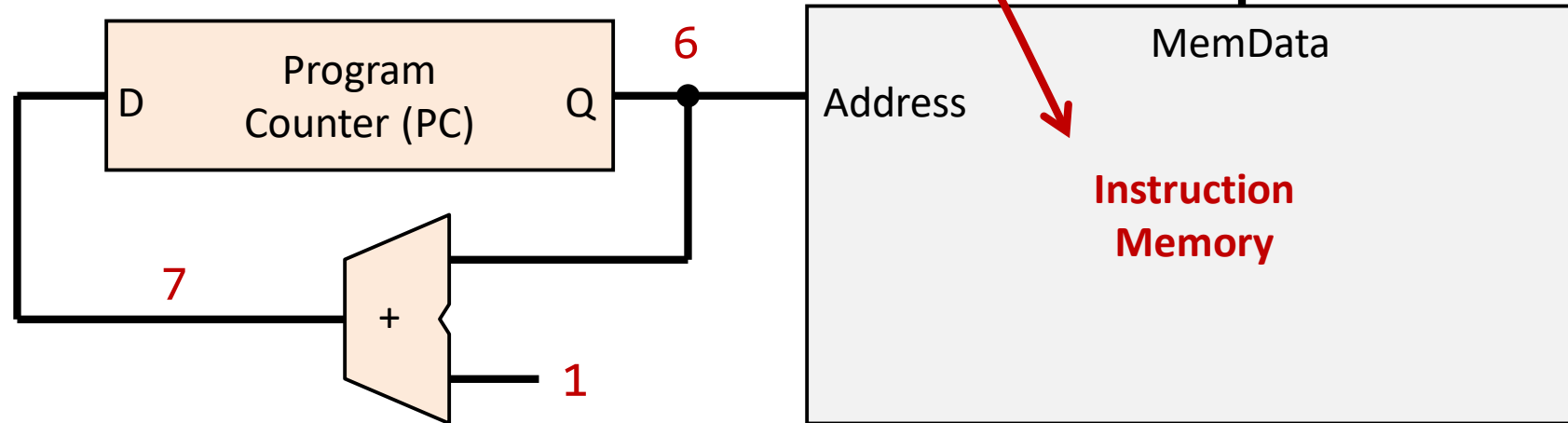
```
0          li    x1, 0x00123456
1          li    x2, 0
2          li    x3, 1
3          li    x4, 0
4          li    x5, 0
5          li    x6, 32
6  loop:  and    x5, x1, x3
7          add    x2, x2, x5
8          srli   x1, x1, 1
9          addi   x4, x4, 1
10         bne    x4, x6, loop
```

Why Did We Do That?!



A Processor: Fetching Instructions

```
...  
5      li    x6, 32  
6  loop: and   x5, x1, x3  
7      add   x2, x2, x5  
...
```

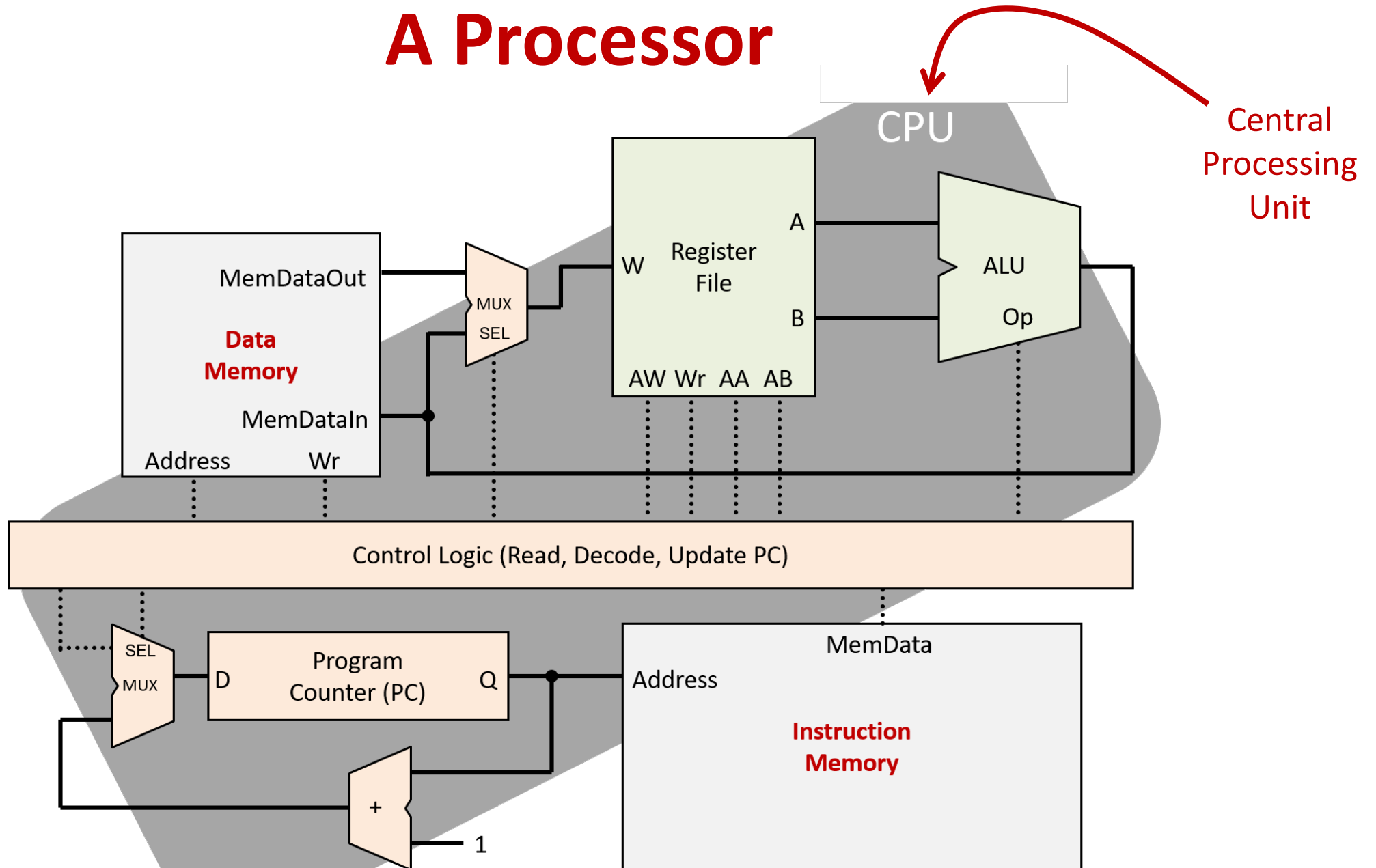


10

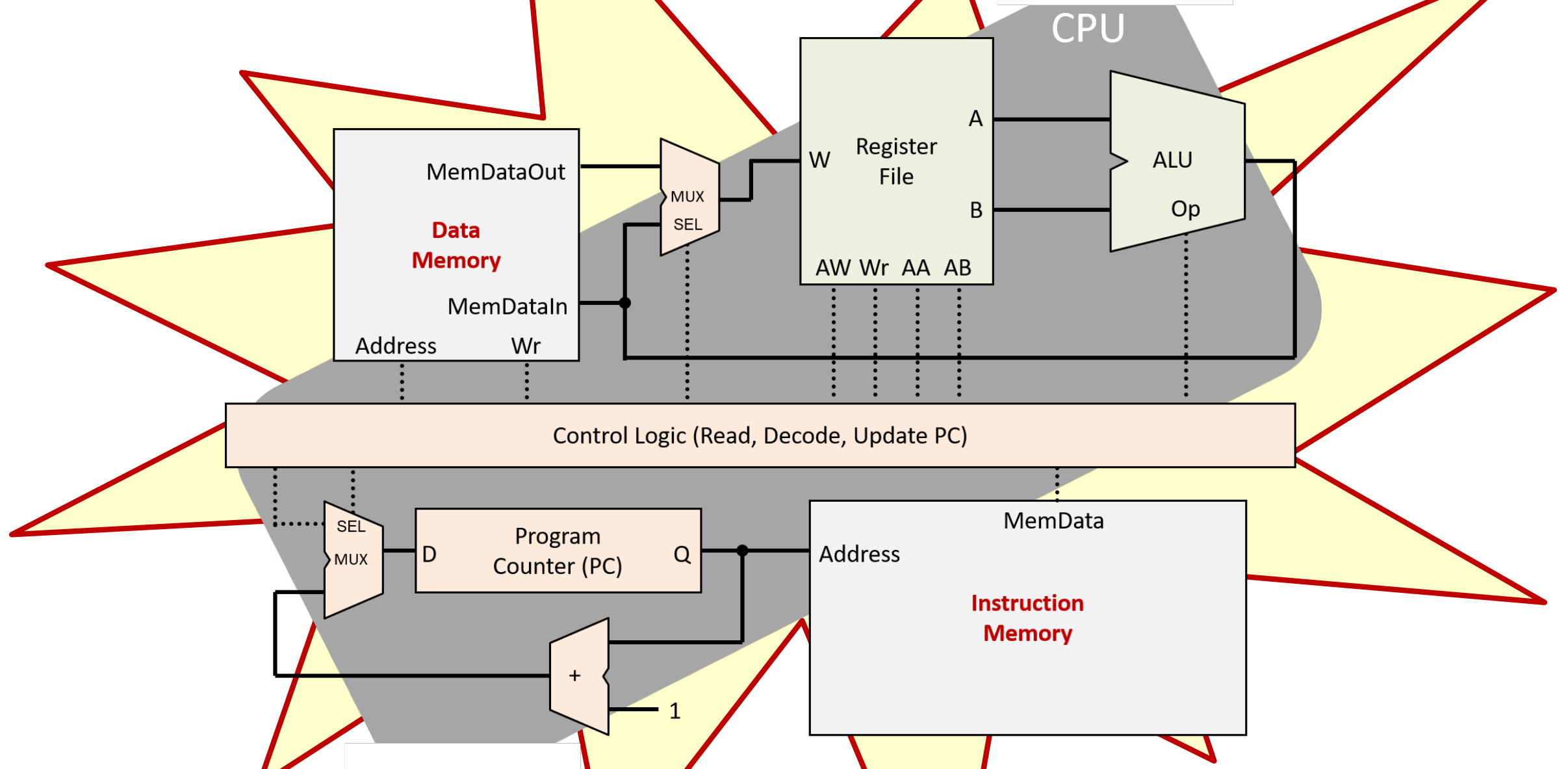
0

“and x5, x1, x3”

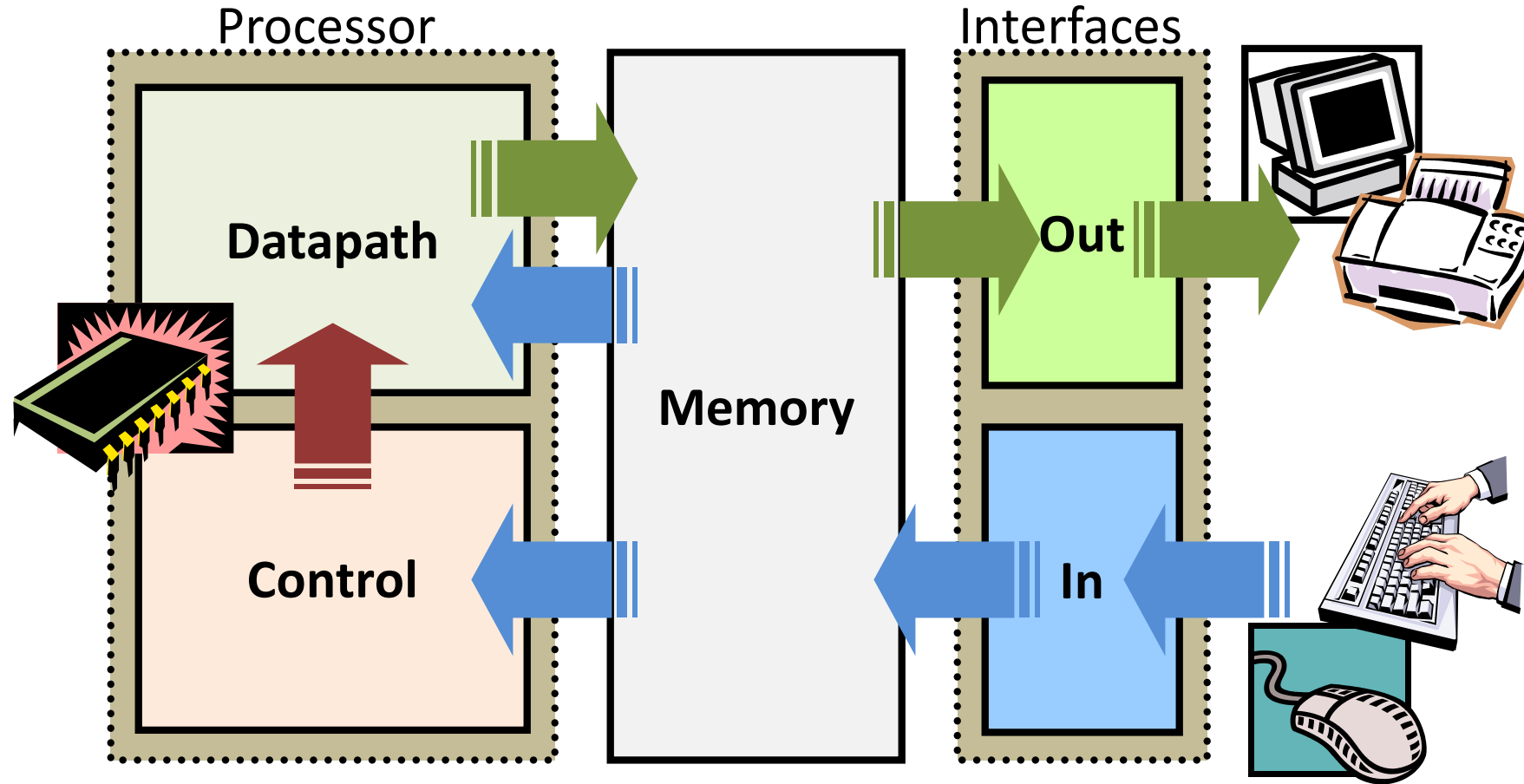
A Processor



Labs B & C: Design Your Own RISC-V Processor!

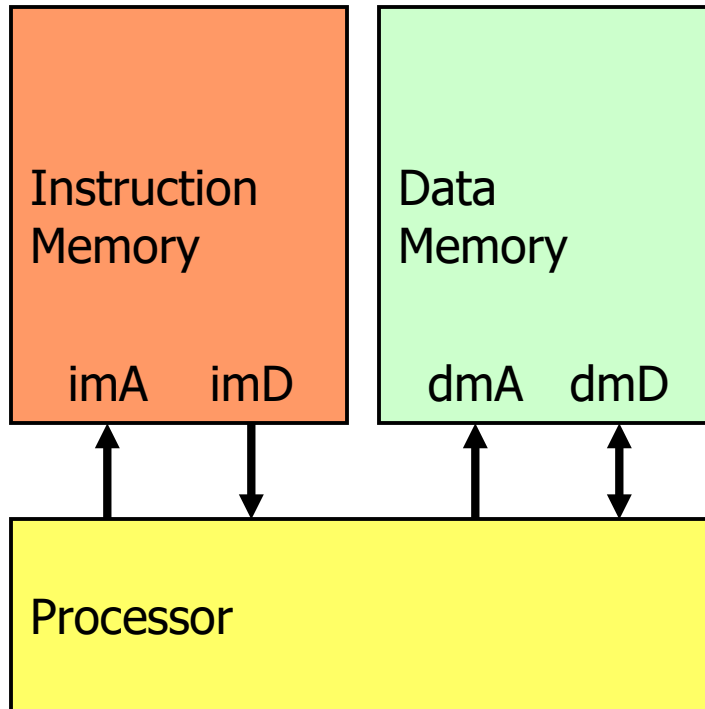


The Five Classic Components of a Computer

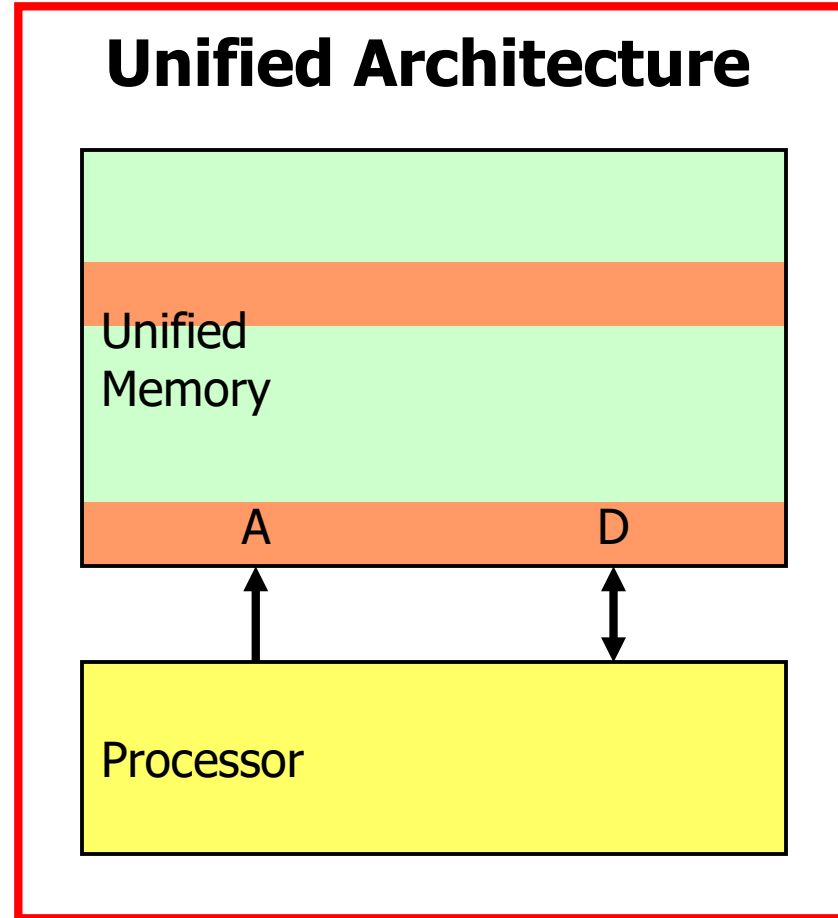


Joint or Disjoint Program and Data Memories

Harvard Architecture

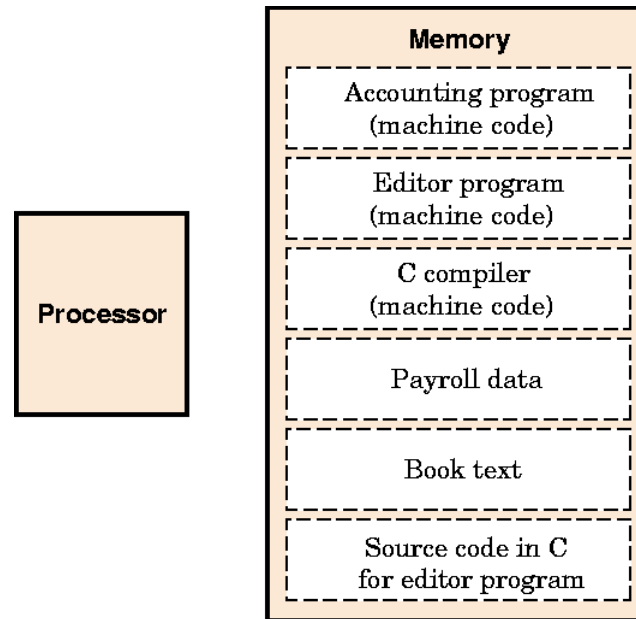


Unified Architecture

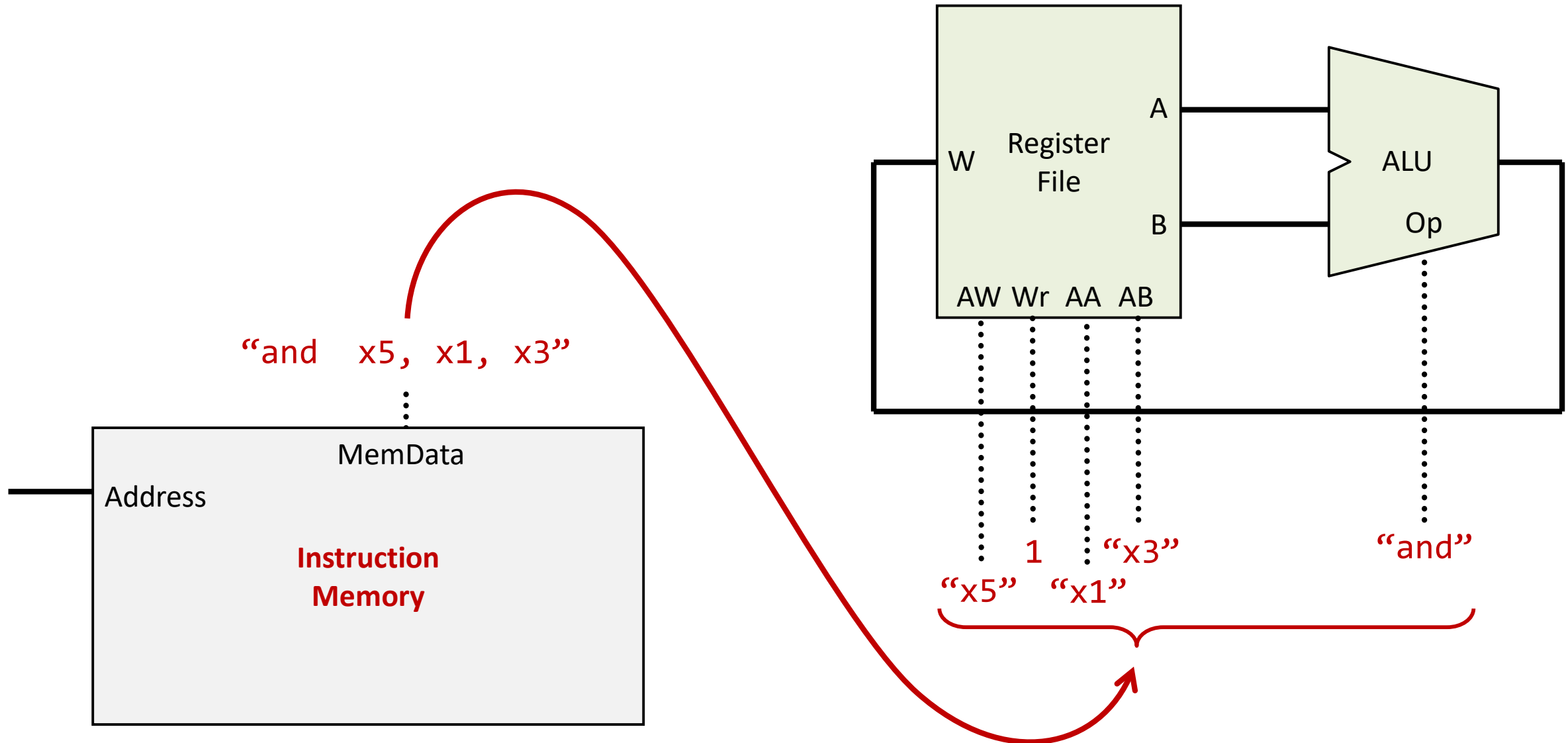


Stored Program Concept: The Key to Computer Science

- Instructions represented as numbers
- Programs are stored in memory and read/written just like data



Representing Instructions?



The Encoding Problem

“and x5, x1, x3”

???



0101101... ??? ...0101110

Decoder

01001	1	00001	00011	???...???
“x5”	1	“x1”	“x3”	“and”

A Possible Encoding...

add x1, x1, x1
add x1, x1, x2
add x1, x1, x3
add x1, x1, x4
add x1, x1, x5

...

and x1, x1, x1
and x1, x1, x2
and x1, x1, x3
and x1, x1, x4
and x1, x1, x5

...

0 = 0000 0000 0000 0000 0000 0000 0000 0000
1 = 0000 0000 0000 0000 0000 0000 0000 0001
2 = 0000 0000 0000 0000 0000 0000 0000 0010
3 = 0000 0000 0000 0000 0000 0000 0000 0011
4 = 0000 0000 0000 0000 0000 0000 0000 0100

...

32768 = 0000 0000 0000 0000 1000 0000 0000 0000
32769 = 0000 0000 0000 0000 1000 0000 0000 0001
32770 = 0000 0000 0000 0000 1000 0000 0000 0010
32771 = 0000 0000 0000 0000 1000 0000 0000 0011
32772 = 0000 0000 0000 0000 1000 0000 0000 0100

...

of opcodes × # destinations × # source 1 × # source 1 ≤ 2³² combinations

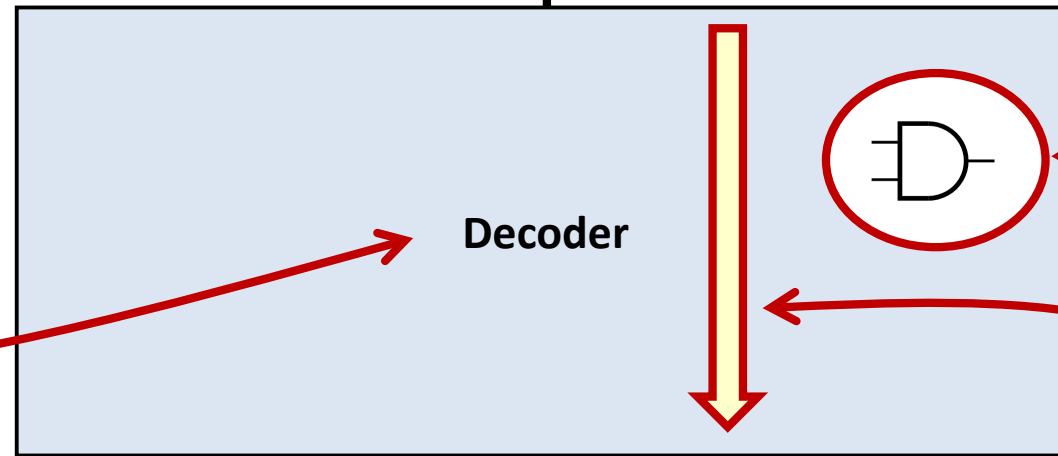
What Makes a Good Encoding?

“and x5, x1, x3”

0000 0000 0010 0000 1000 0011 0010 0011

A Boolean combinational function whose complexity depends on the encoding

We want to minimize hardware resources...



...and we want to spend minimal time decoding!

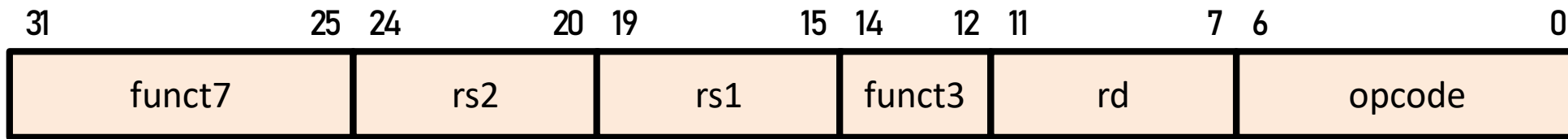
01001 1 00001 00011 ???...???

“x5” 1 “x1” “x3” “and”

A Systematic Encoding

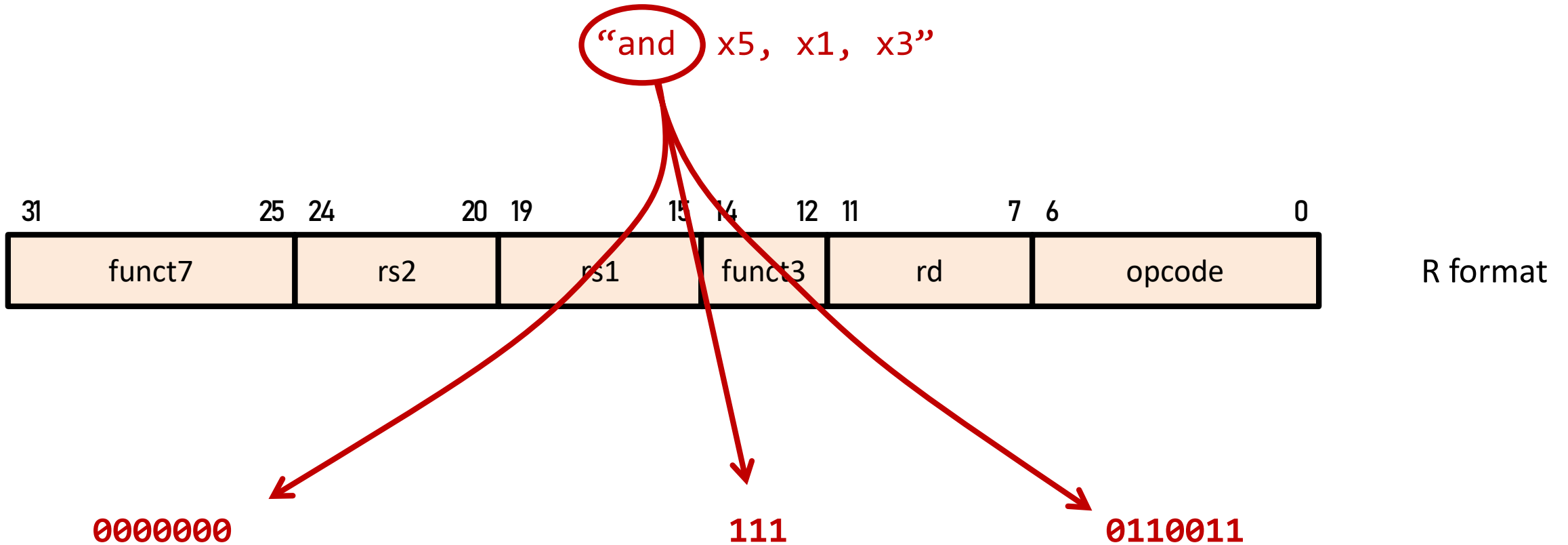
“and x5, x1, x3”

opcode rd, rs1, rs2



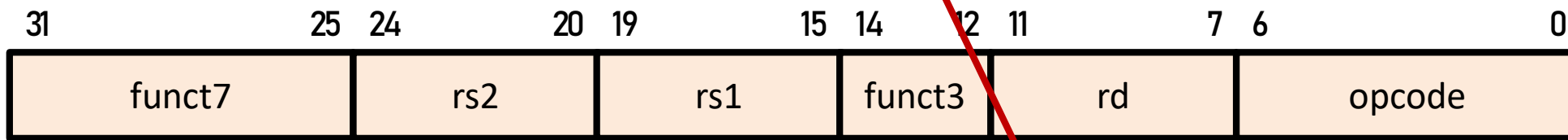
R format

A Systematic Encoding



A Systematic Encoding

“and (x5), x1, x3”



R format

0000000

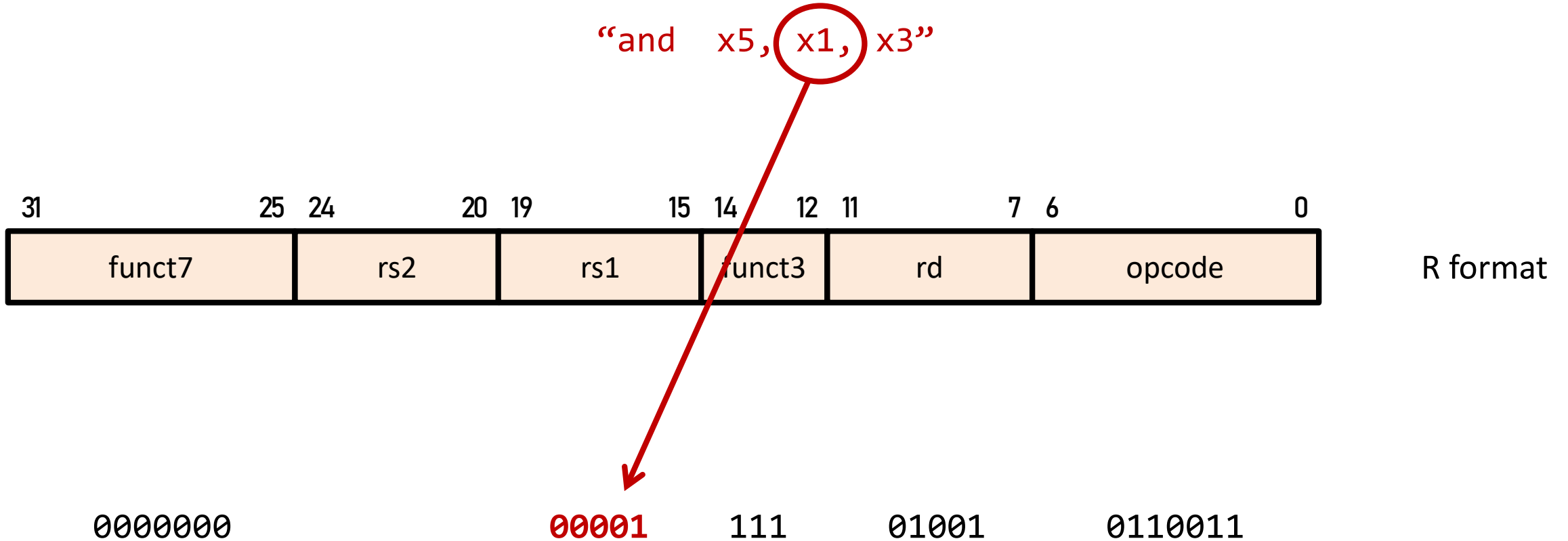
111

01001

0110011

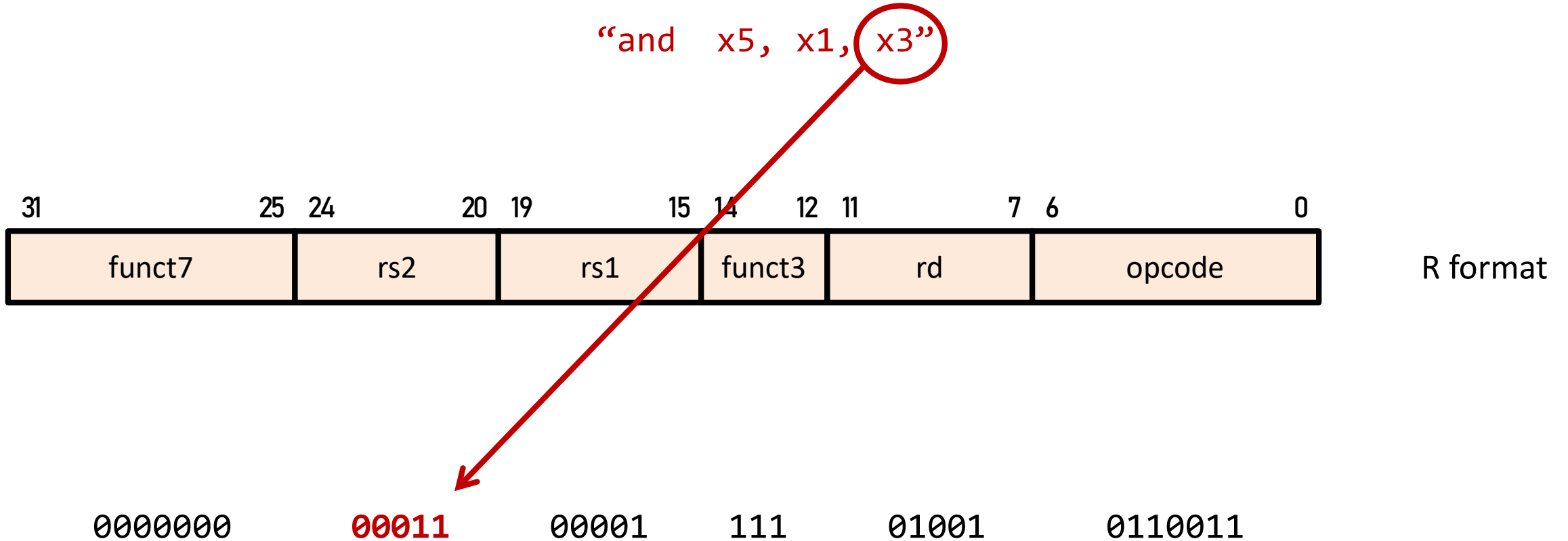
A Systematic Encoding

“and x5, **x1**, x3”



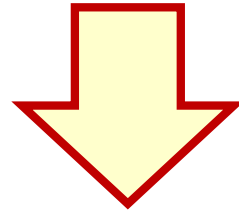
A Systematic Encoding

“and x5, x1, x3”



A Systematic Encoding

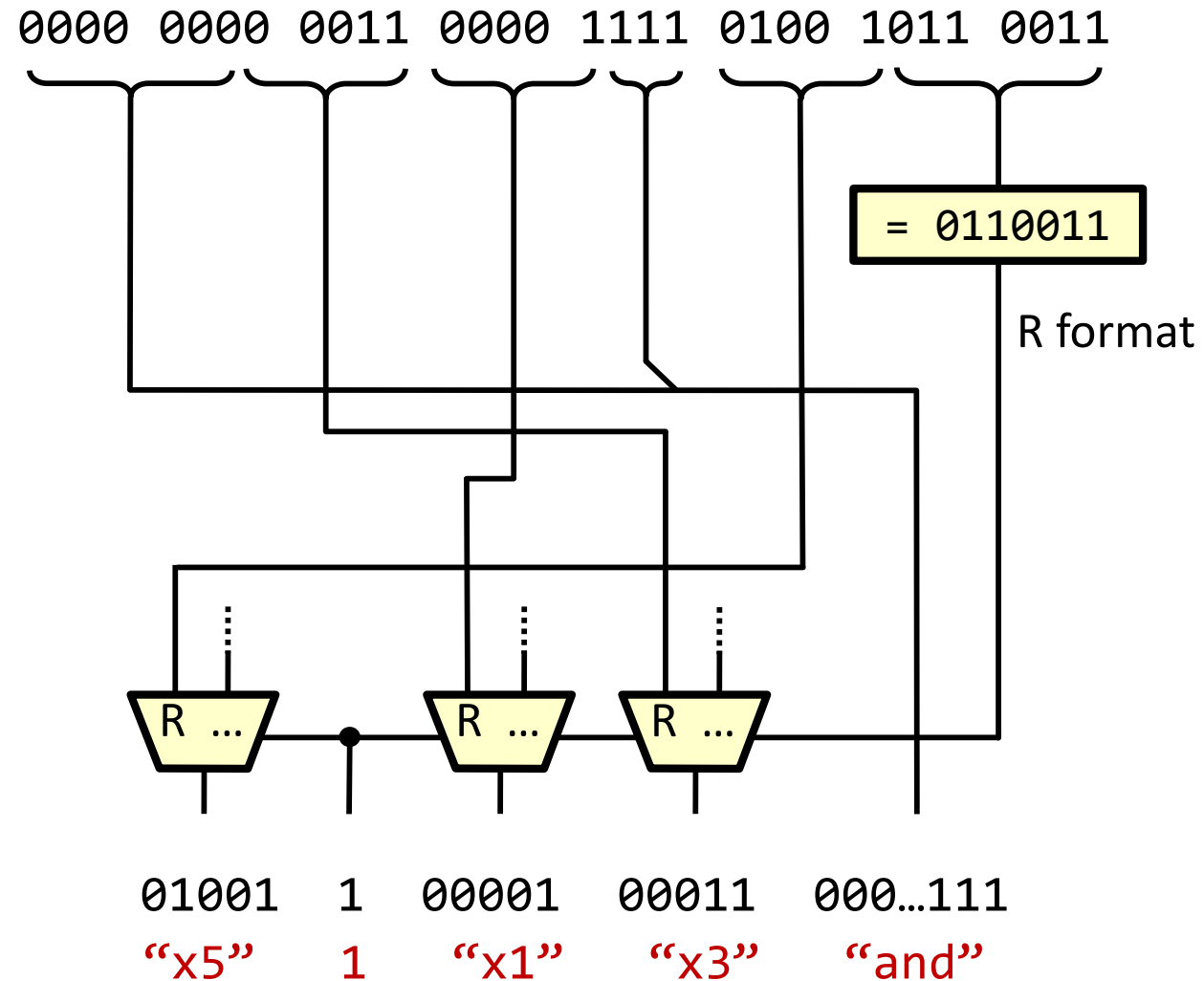
`“and x5, x1, x3”`



0000 0000 0011 0000 1111 0100 1011 0011

An Efficient Encoding!

“and x5, x1, x3”



RISC-V Encoding

Instruction	Pseudocode	Type	funct7	funct3	opcode
Shift					
sll rd,rs1,rs2	$rd \leftarrow rs1 \ll rs2$	R	0x00	0x1	0x33
slli rd,rs1,imm	$rd \leftarrow rs1 \ll imm$	I	0x00	0x1	0x13
srl rd,rs1,rs2	$rd \leftarrow rs1 \gg_u rs2$	R	0x00	0x5	0x33
srli rd,rs1,imm	$rd \leftarrow rs1 \gg_u imm$	I	0x00	0x5	0x13
sra rd,rs1,rs2	$rd \leftarrow rs1 \gg_s rs2$	R	0x20	0x5	0x33
srai rd,rs1,imm	$rd \leftarrow rs1 \gg_s imm$	I	0x20	0x5	0x13

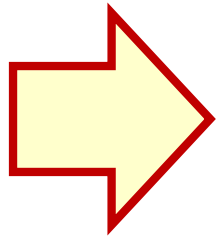
Complete ISA on
Moodle!

Arithmetic					
add rd,rs1,rs2	$rd \leftarrow rs1 + rs2$	R	0x00	0x0	0x33
addi rd,rs1,imm	$rd \leftarrow rs1 + sext(imm)$	I		0x0	0x13
sub rd,rs1,rs2	$rd \leftarrow rs1 - rs2$	R	0x20	0x0	0x33
lui rd,imm	$rd \leftarrow imm$				
auipc rd,imm	$rd \leftarrow pc + imm$				

Logical			Instruction types													
			31	25	24	20	19	15	14	12	11	7	6	0		
xor	rd,rs1,rs2	$rd \leftarrow rs1 \oplus rs2$	R	funct7			rs2		rs1	funct3		rd		opcode	Register-Register	
xori	rd,rs1,imm	$rd \leftarrow rs1 \oplus imm$		I	imm[11:0]					rs1	funct3		rd	opcode	Register-Immediate	
or	rd,rs1,rs2	$rd \leftarrow rs1 \vee rs2$			I	funct7			imm[4:0]		rs1	funct3		rd	opcode	Register-Immediate Shift
ori	rd,rs1,imm	$rd \leftarrow rs1 \vee imm$		S		imm[11:5]			rs2		rs1	funct3		imm[4:0]		opcode
and	rd,rs1,rs2	$rd \leftarrow rs1 \wedge rs2$			B	imm[12—10:5]			rs2		rs1	funct3		imm[4:1—11]		opcode
andi	rd,rs1,imm	$rd \leftarrow rs1 \wedge imm$		U		imm[31:12]							rd		opcode	Upper Immediate
					J	imm[20—10:1—11—19:12]							rd		opcode	Jump

Assemblers

```
0      li    x1, 0x00123456
1      li    x2, 0
2      li    x3, 1
3      li    x4, 0
4      li    x5, 0
5      li    x6, 32
6  loop: and   x5, x1, x3
7      add   x2, x2, x5
8      srli  x1, x1, 1
9      addi  x4, x4, 1
10     bne   x4, x6, loop
```

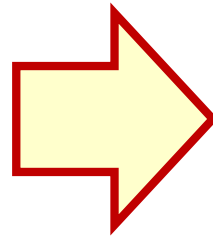


```
0101 0101 0101 0000 0100 0111 1010 1110
0001 0100 1001 1101 0011 0000 1100 1001
1101 1100 1101 0110 0000 1101 0001 0111
0010 0011 1101 0110 0010 0000 0001 1001
1100 1010 1011 1010 0111 0100 0000 0110
1111 0010 1001 0011 1001 1110 1001 1101
0011 0000 0010 0111 1111 0000 0100 0011
0111 1001 0101 1101 1000 1000 0111 1011
1100 1010 1011 0000 0100 0100 0110 0101
0111 1001 0010 0110 0000 0011 0001 0010
0101 1100 1000 0101 0000
```

A fairly trivial job

Compilers

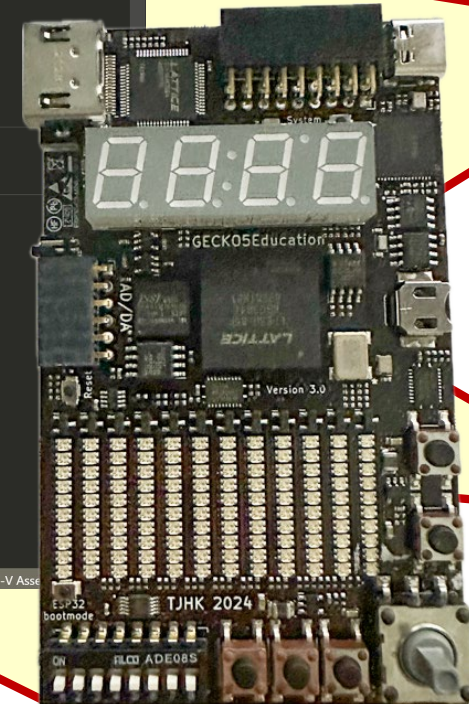
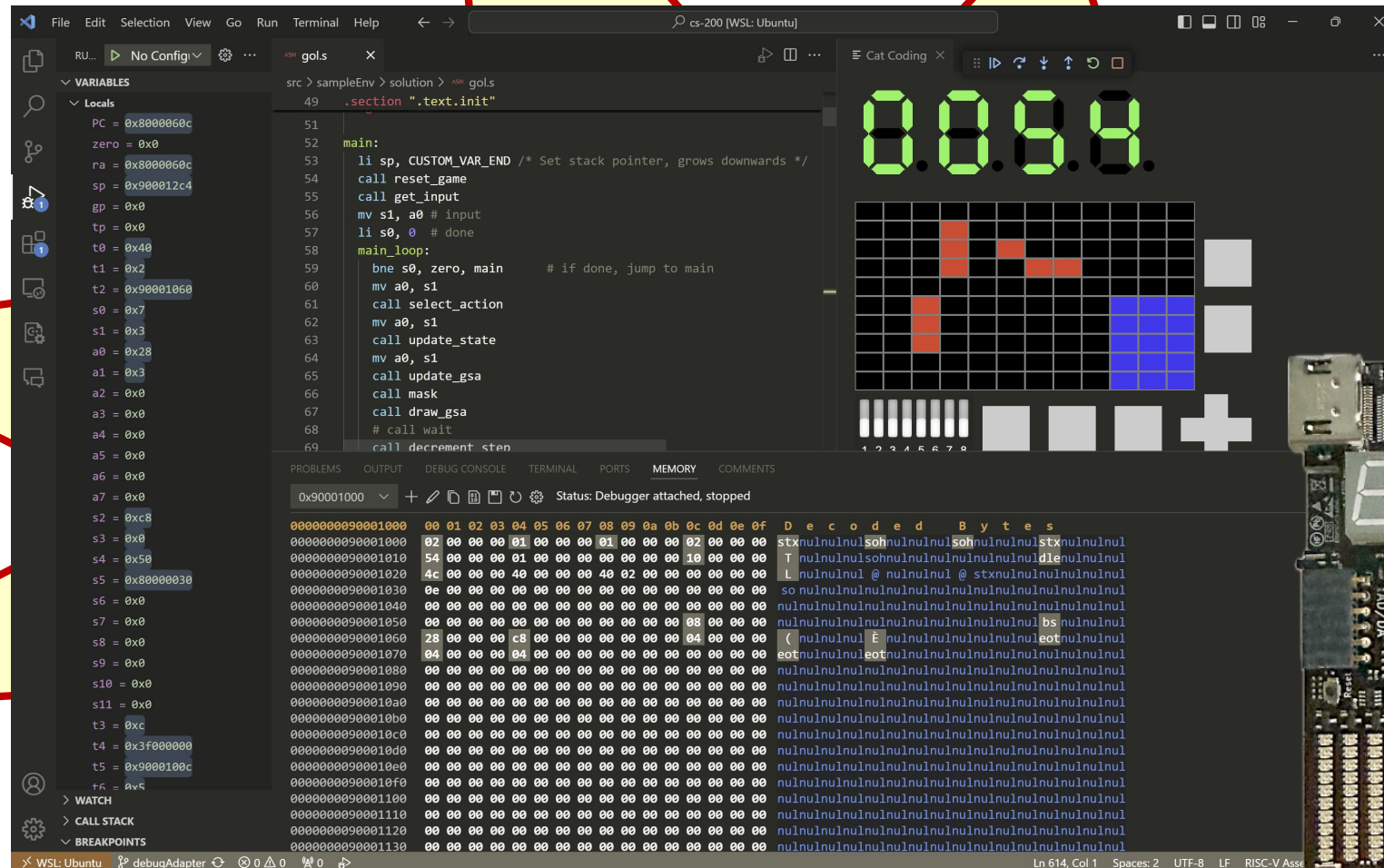
```
int data    = 0x00123456;
int result  = 0;
int mask    = 1;
int count   = 0;
int temp    = 0;
int limit   = 32;
do {
    temp    = data & mask;
    result  = result + temp;
    data    = data >> 1;
    count   = count + 1;
} while (count != limit)
```



```
0      li    x1, 0x00123456
1      li    x2, 0
2      li    x3, 1
3      li    x4, 0
4      li    x5, 0
5      li    x6, 32
6  loop: and    x5, x1, x3
7      add    x2, x2, x5
8      srli   x1, x1, 1
9      addi   x4, x4, 1
       bne    x4, x6, loop
```

A pretty hard job!...

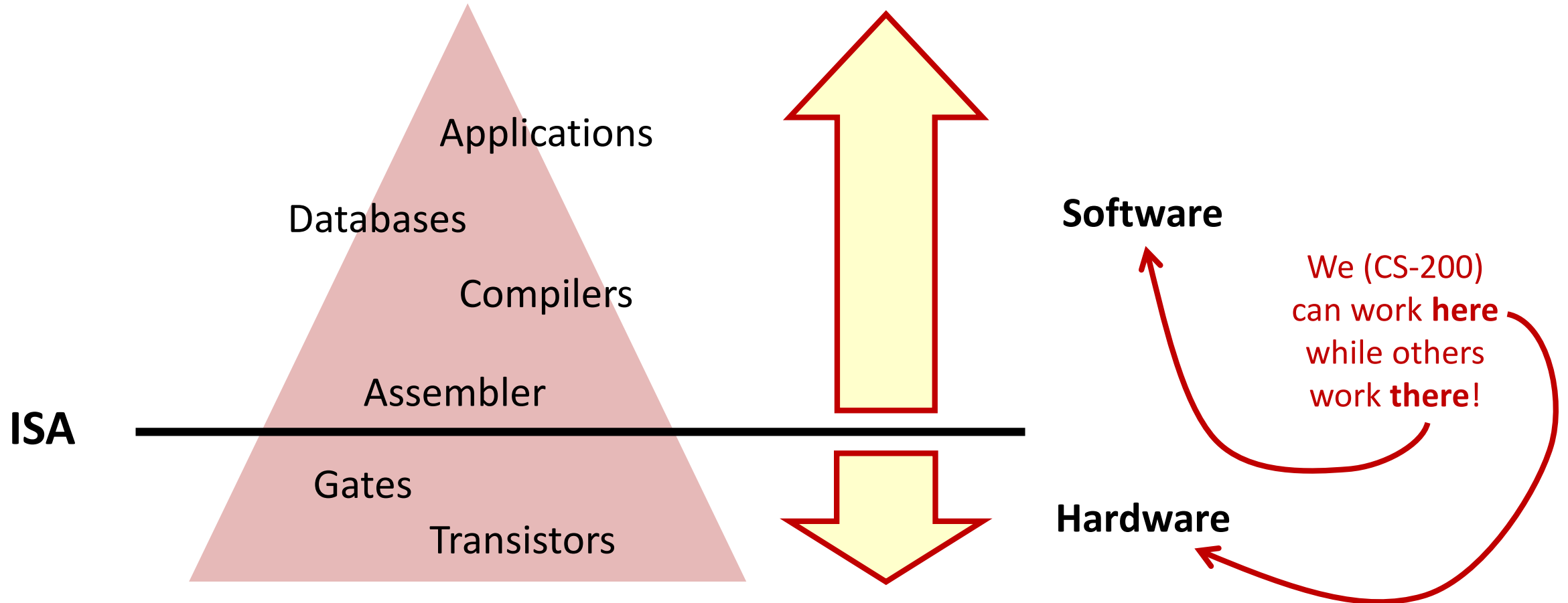
Lab A: Design a Real Game in Assembly!



Instruction Set Architecture (ISA)

- **Everything one needs to know to program the processor**
 - Instructions the processors can execute
 - Registers available, size, etc.
 - Binary encoding of the instructions
- No (direct) details on the processor hardware but **strong indirect impact**
- Typical example:
 - **x86** is a very common ISA introduced by Intel
 - **x64** is an extension of x86 introduced by AMD and now supported also by Intel
 - 8086, 80286, 80386, Pentium (Intel) and Athlon (AMD) are old processors conforming to this ISA
 - Xeon, Core i5, i7, i9 (Intel) and EPYC, Ryzen 5, 7, 9 (AMD) are more recent ones

ISA Is the Key Abstraction in Computer Systems



References

- Patterson & Hennessy, COD – RISC-V Edition
 - **Chapter 2** and, in particular, **Sections 2.1-2.5**