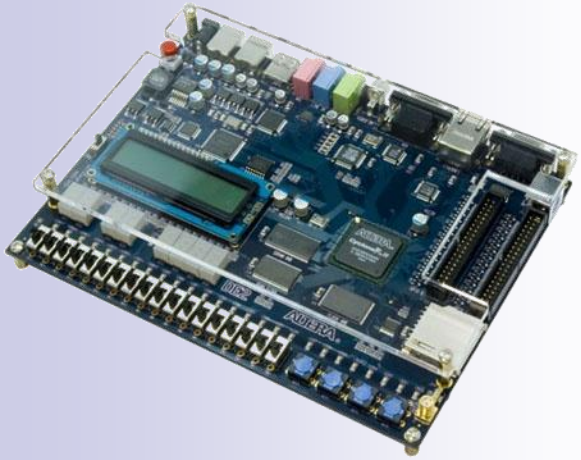
Logic Systems and Processors
cz:Logické systémy a procesory



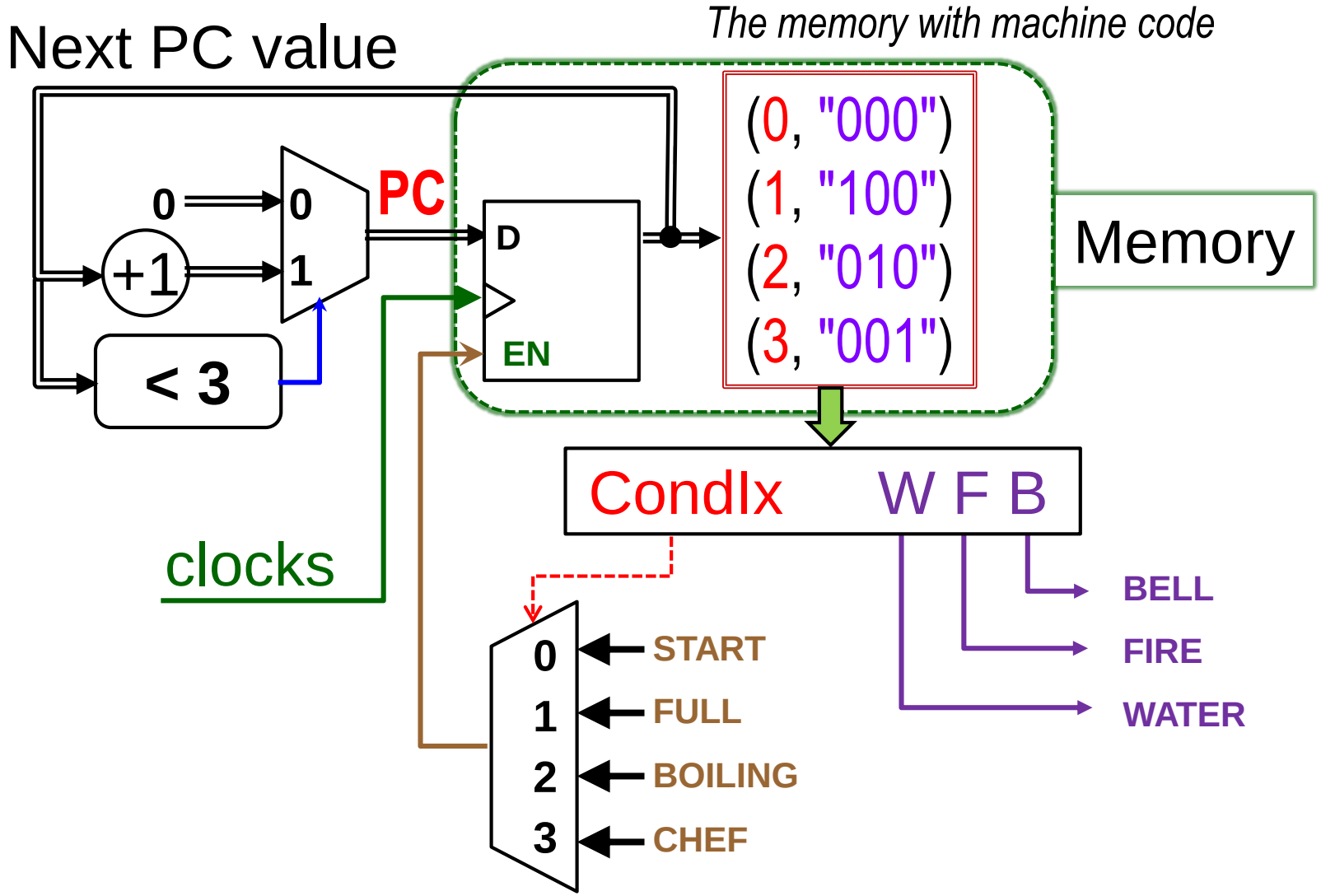
Lecturer: Richard Šusta

richard@susta.cz, susta@fel.cvut.cz,
+420 2 2435 **7359**

Version: 1.1 - corrections of typos

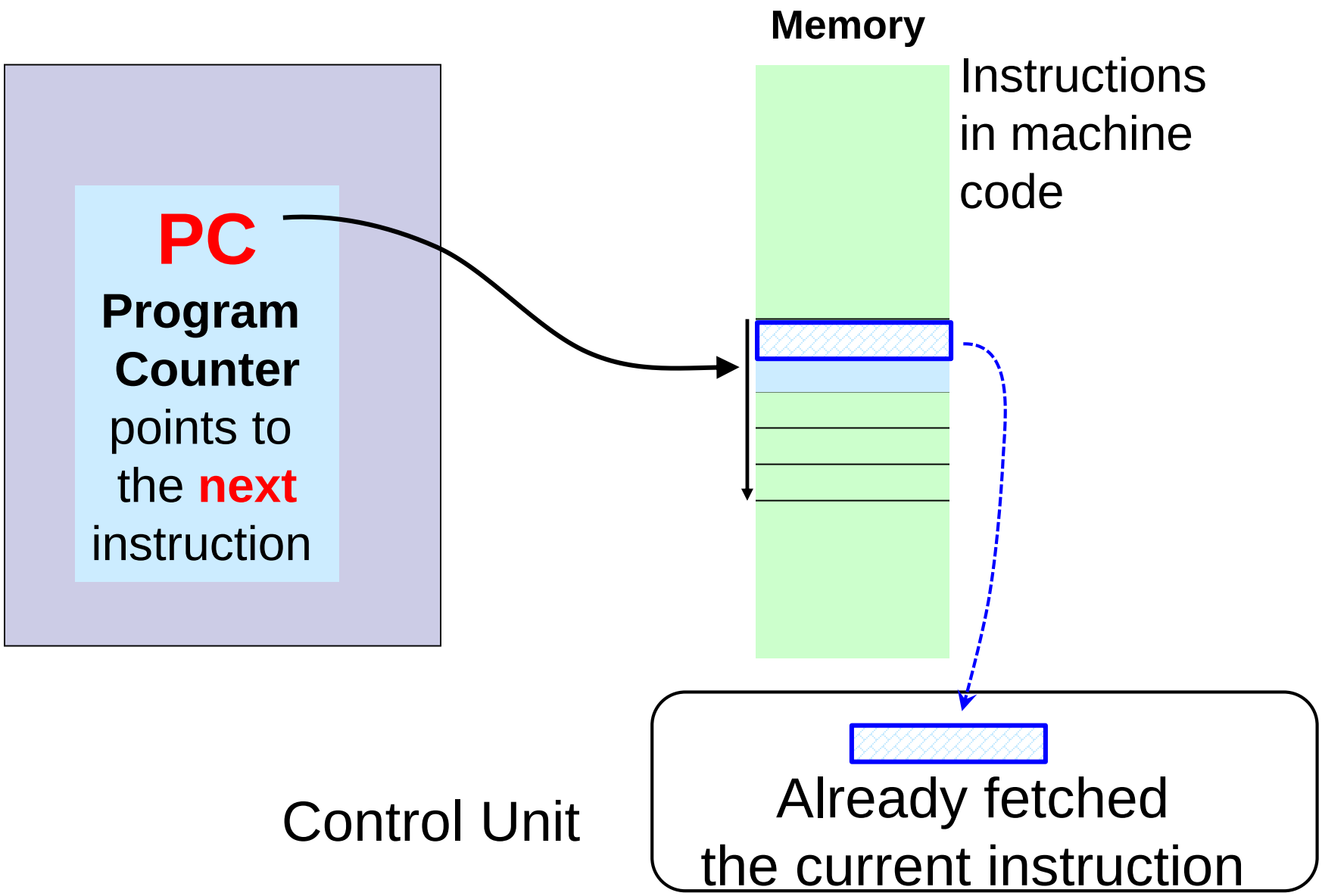
Microprocessors

Reminder: Control Unit with Micro-program

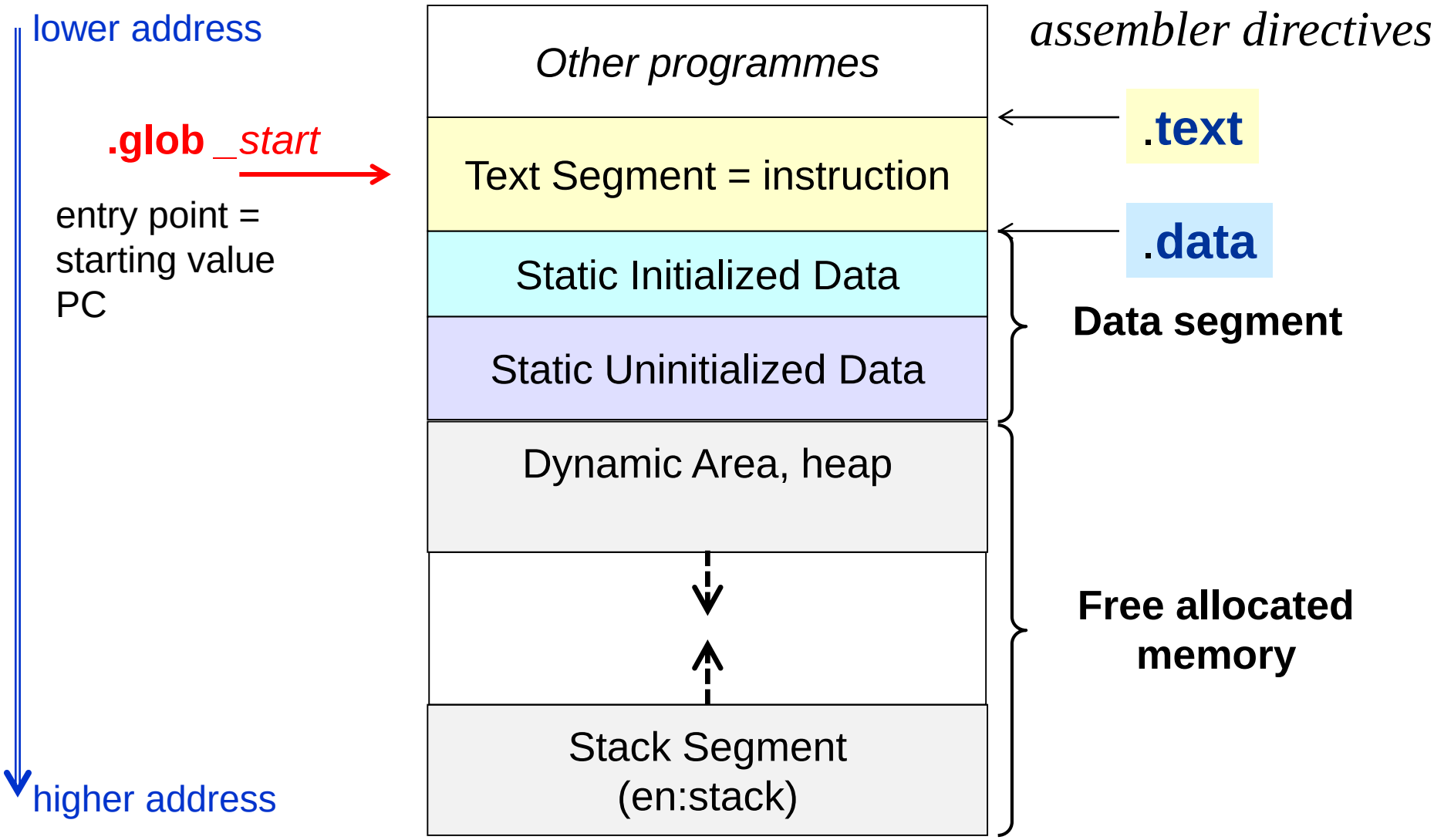


Memories need address registers, PC points to the next address.

Program Counter



Program in memory



The processor follows the machine code

```
int x3sum=0, x1count=1, x2count=1;
int x4loop = 10;
for (int x4loop = 10; x4loop > 0; x4loop--)
{ x3sum = x1count + x2count;
   x1count = x2count; x2count = x3sum;
}
fib13 = x3sum;
```

assembler

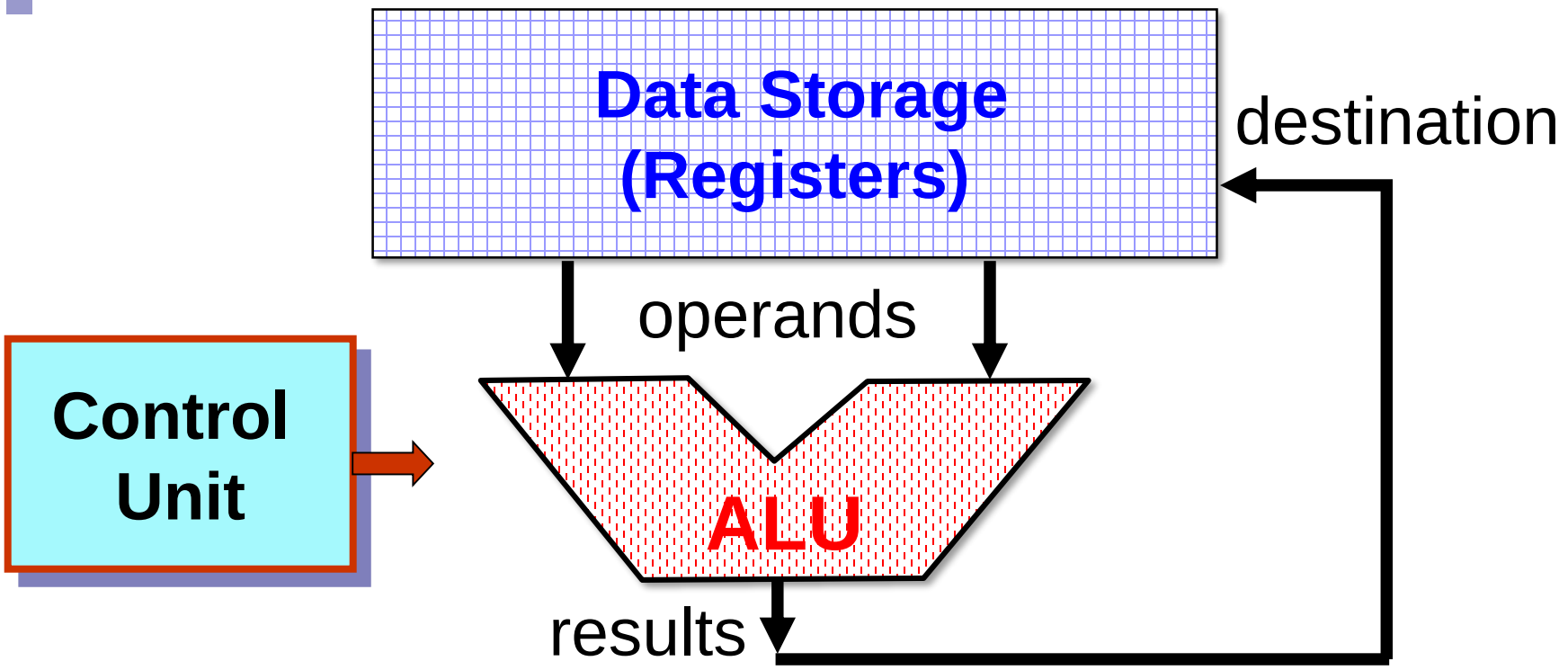
symbolic description of operations

machine code

memory address

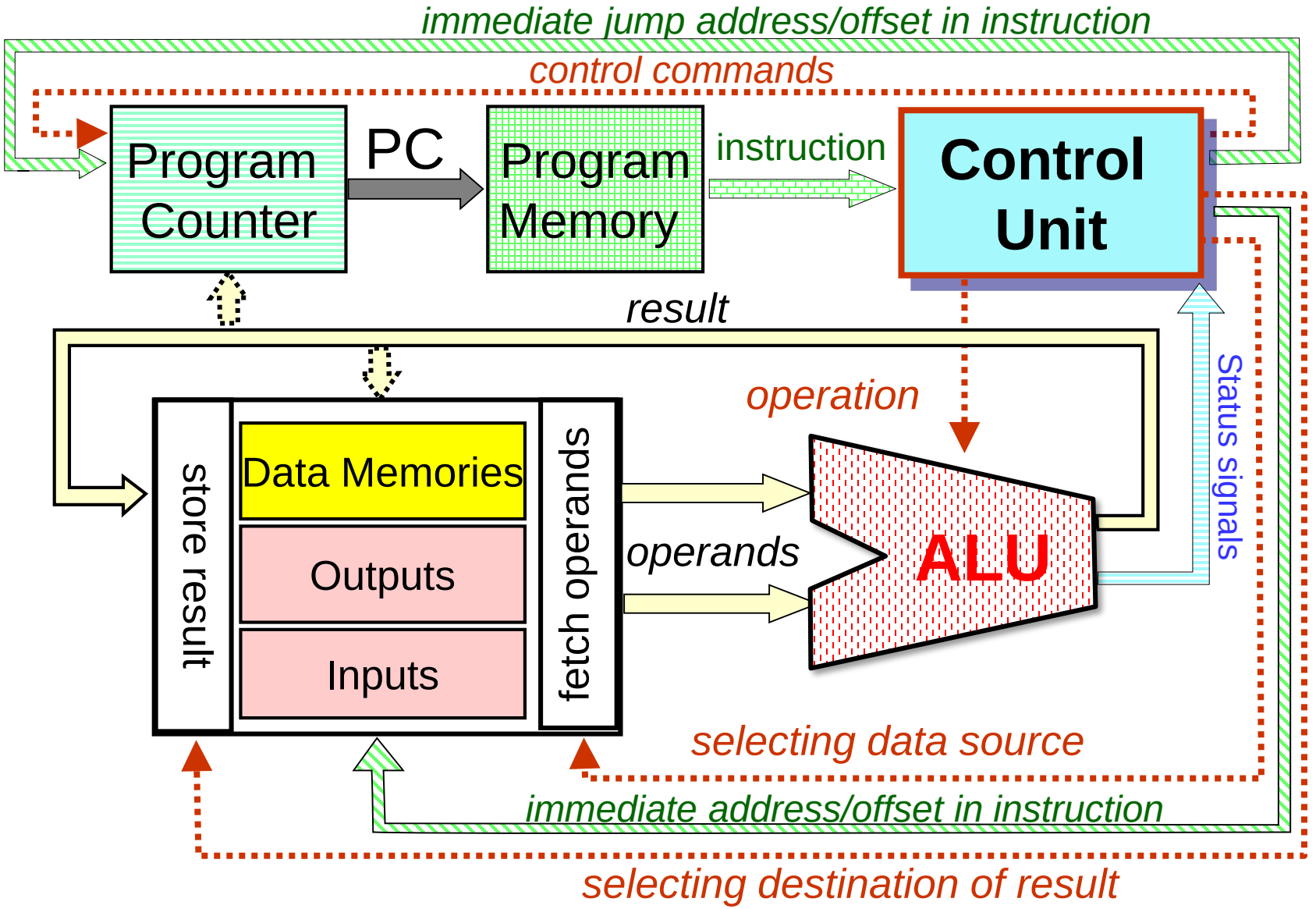
Address	Code	Instruction
0x00000200	00106093	ori x1, x0, 1
0x00000204	00106113	ori x2, x0, 1
0x00000208	00a06213	ori x4, x0, 10
0x0000020c	002081b3	add x3, x1, x2
0x00000210	000160b3	or x1, x2, x0
0x00000214	0001e133	or x2, x3, x0
0x00000218	fff20213	addi x4, x4, -1
0x0000021c	fe4048e3	blt x0, x4, 0x20c
0x00000220	00000217	auipc x4, 0x0
0x00000224	1e020213	addi x4, x4, 480
0x00000228	00322023	sw x3, 0(x4)

The Simplest Processor

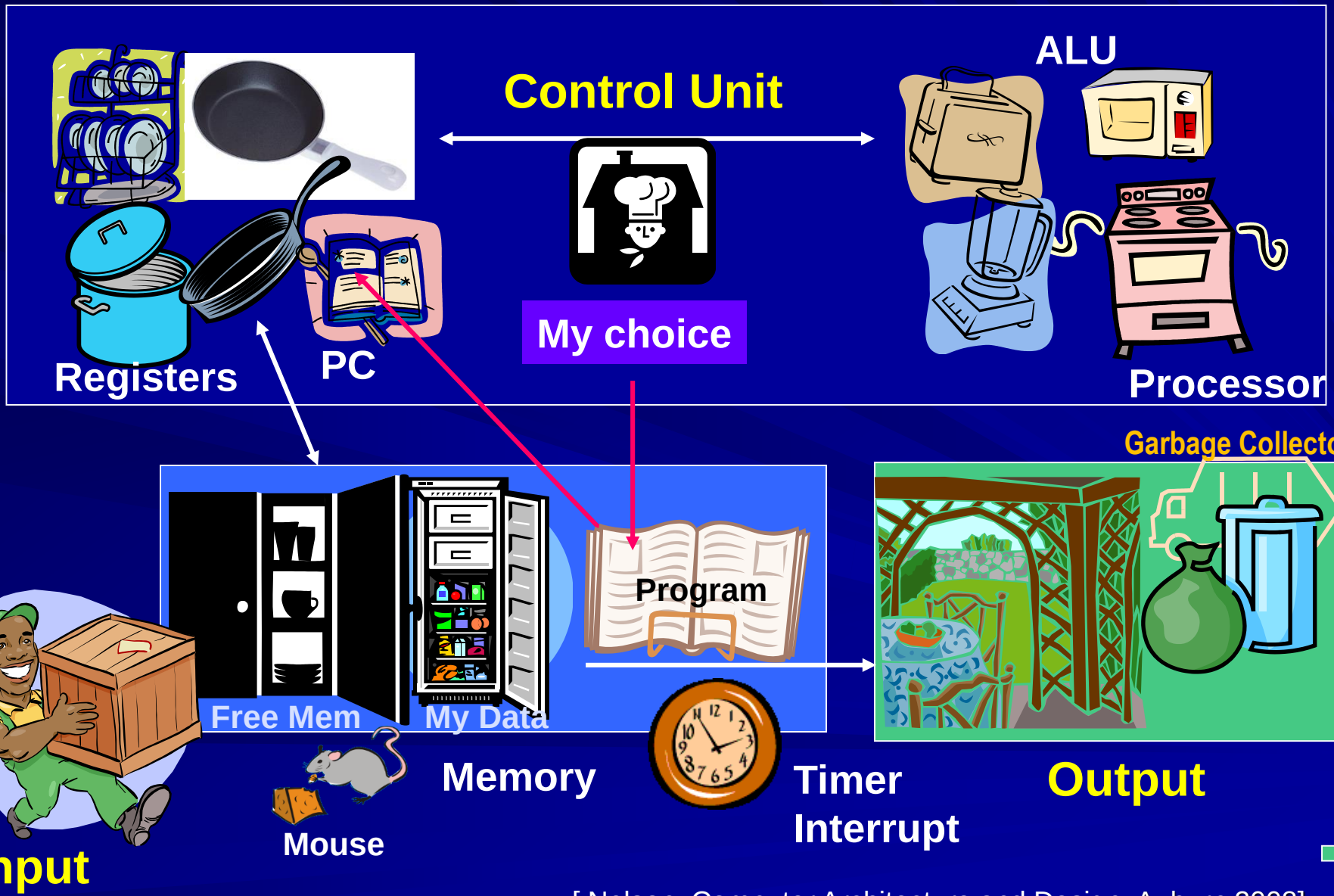


ALU: Arithmetic and Logic Unit

Model of Simple Digital Computer



Processor is something à la Kitchen



[Nelson: Computer Architecture and Design, Auburn 2008]

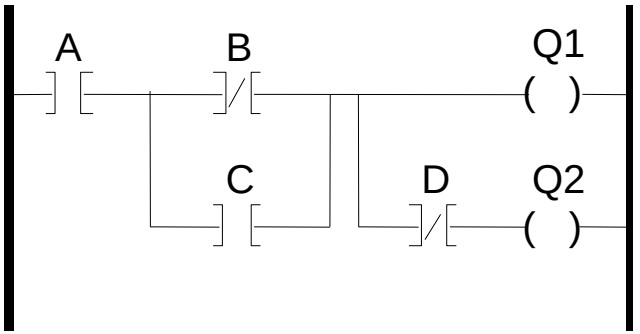
1-bit Processors had 1-bit ALU

1-bit processor for industrial applications at the beginning of processor's evolution.

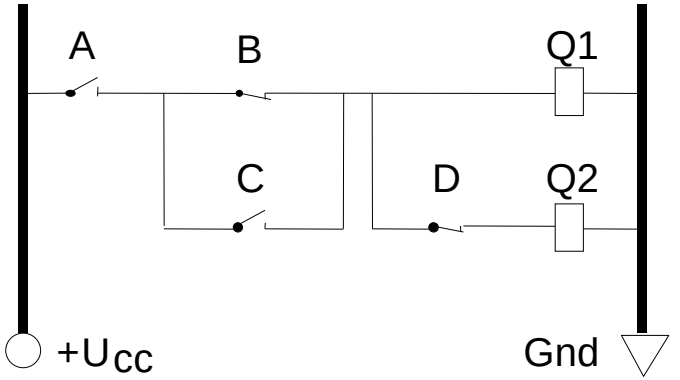
[http://www.motorola.com/]

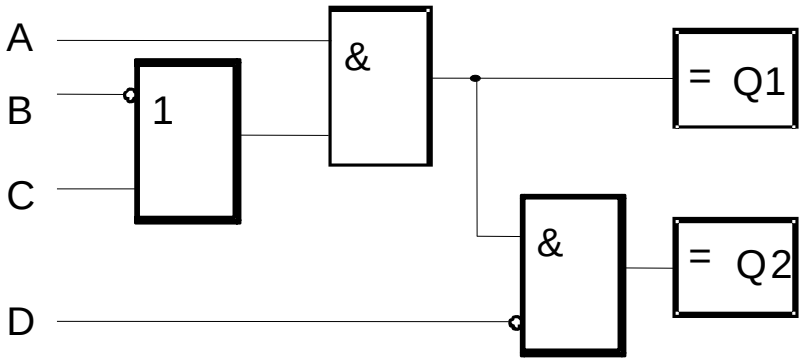
Industrial Programming Languages

Ladder

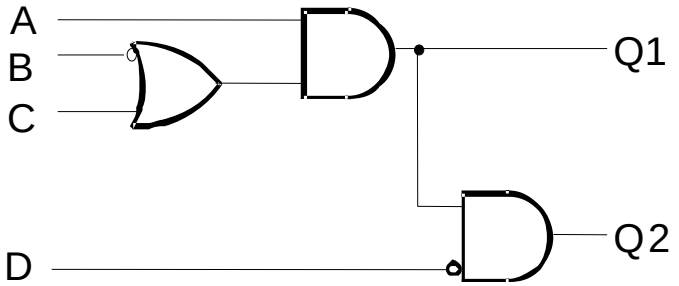


Relay





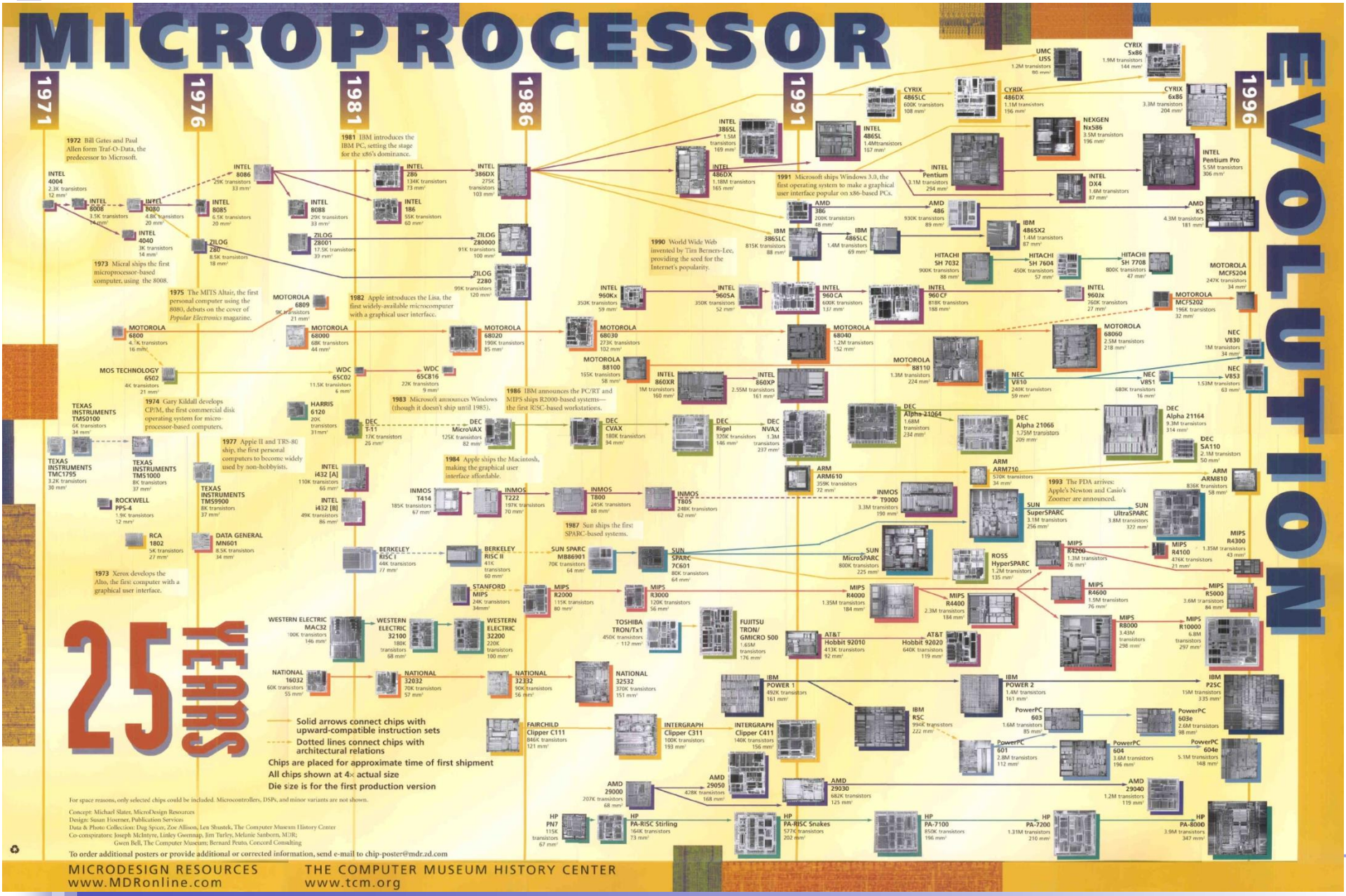
Logic Diagram



Logic circuit

25 Early Years of Multi-bit Processors

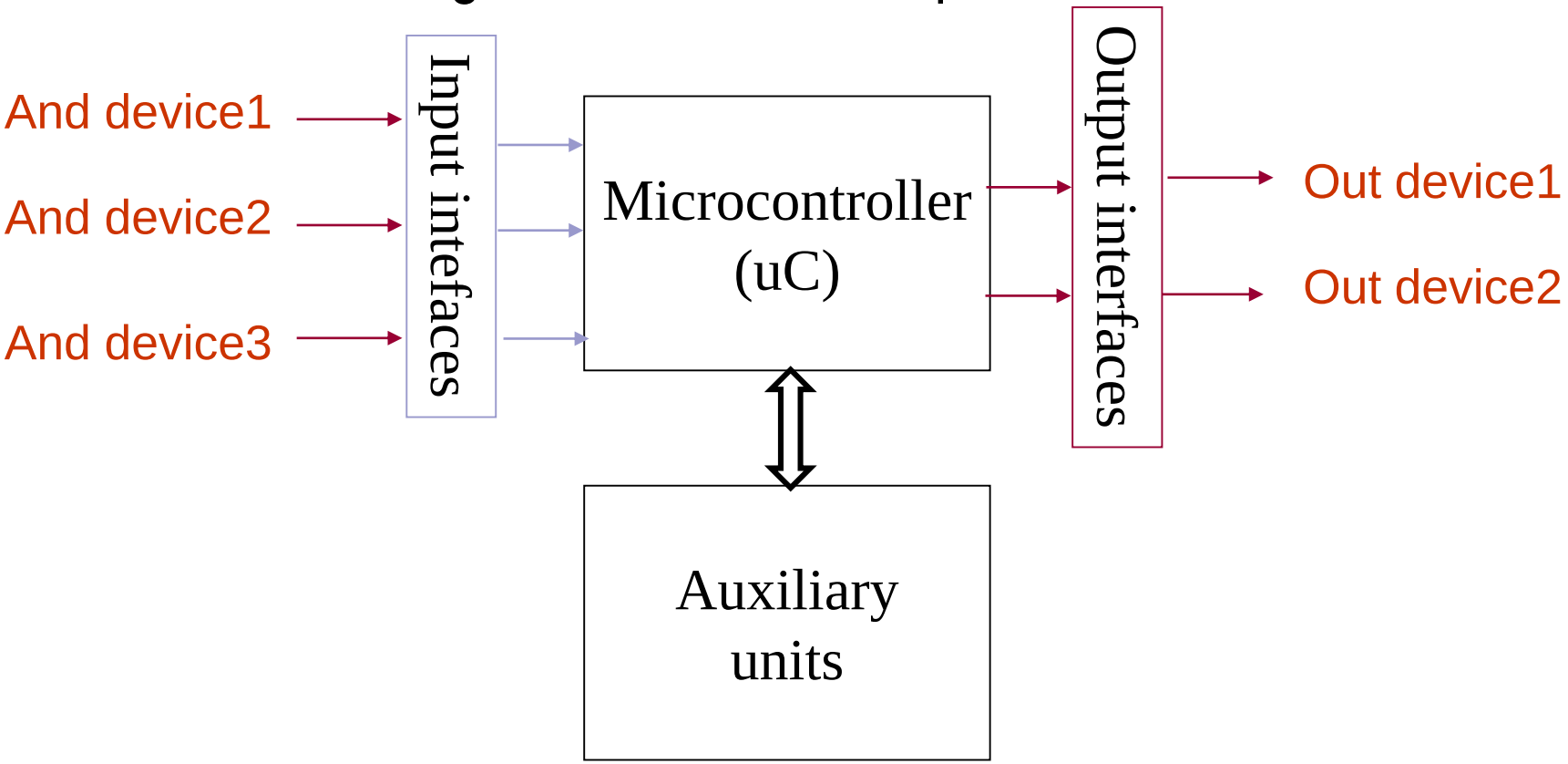
http://research.microsoft.com/en-us/um/people/gbell/CyberMuseum_contents/Microprocessor_Evolution_Poster.jpg



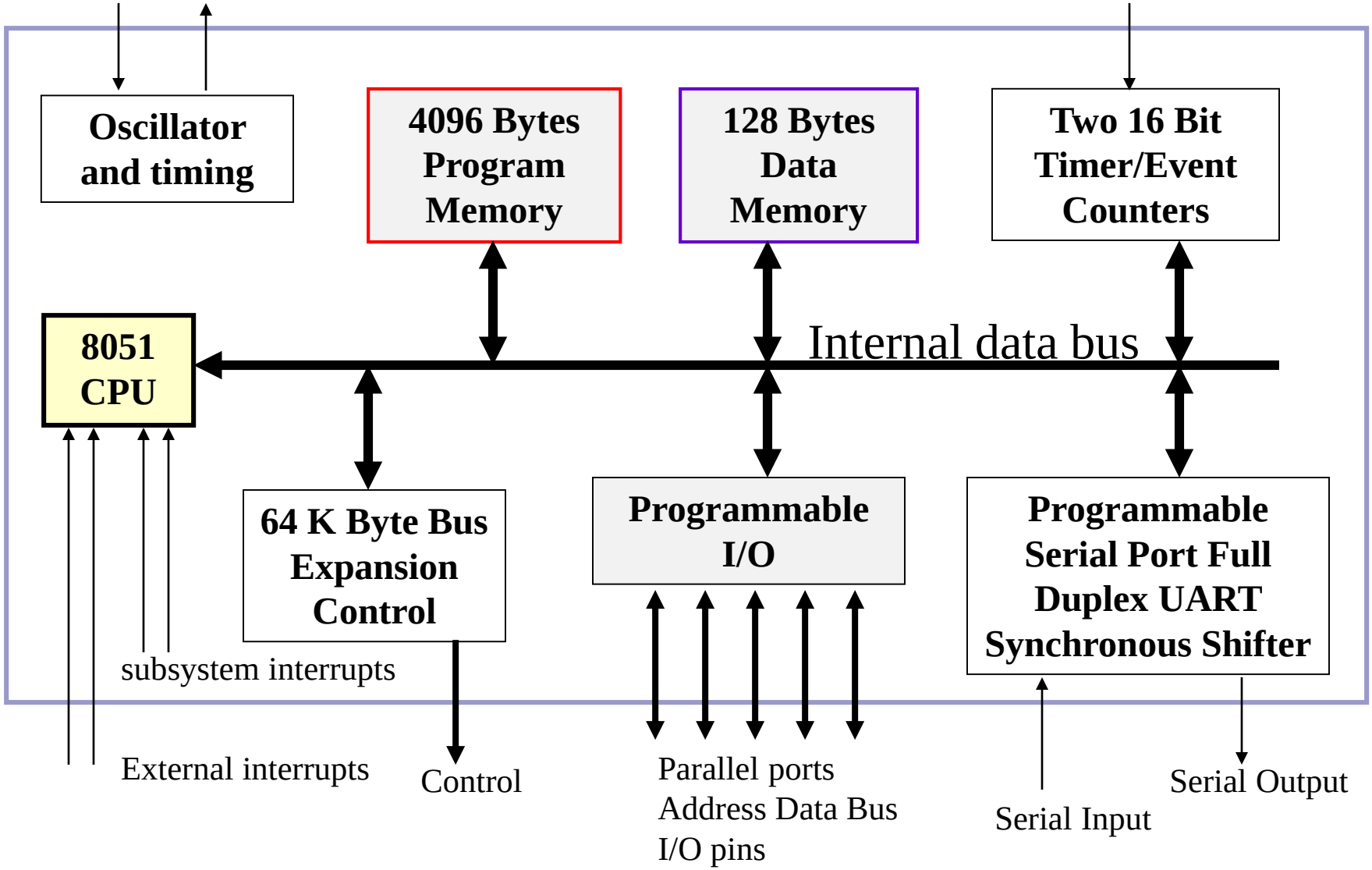
Hardcore/Softcore Processors

- **Hard-core processors**
  - They are manufactured as integrated circuits and can operate stand-alone or be built into another circuit or configurable logic,
    e.g., I7, AMD, ARM, RISC V processors.
- **Soft-core processors**
  - Their circuitry is specified in HDL language and implemented in configurable logic, i.e., by FPGAs. HDLs can describe each processor.
  - Soft-core implementations are slower but more flexible.

- A **microcontroller** is a single-chip unit of any architecture (RISC, CISC, hard-core, soft-core) that has enhanced input/output capabilities and a number of several embedded functional units, e.g., ARM9 or ARM7 processors.

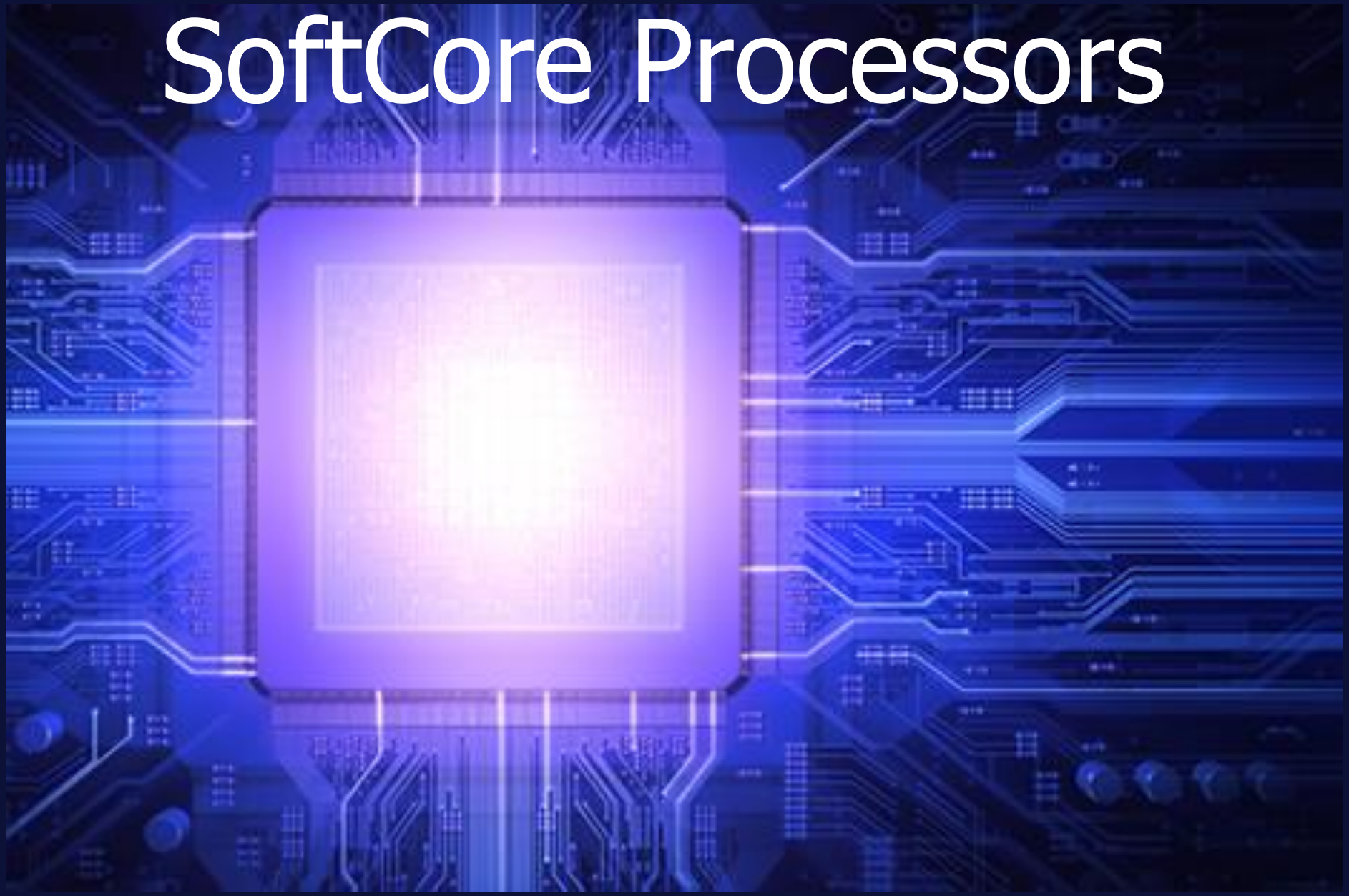


Example: 8051 Microcontroller



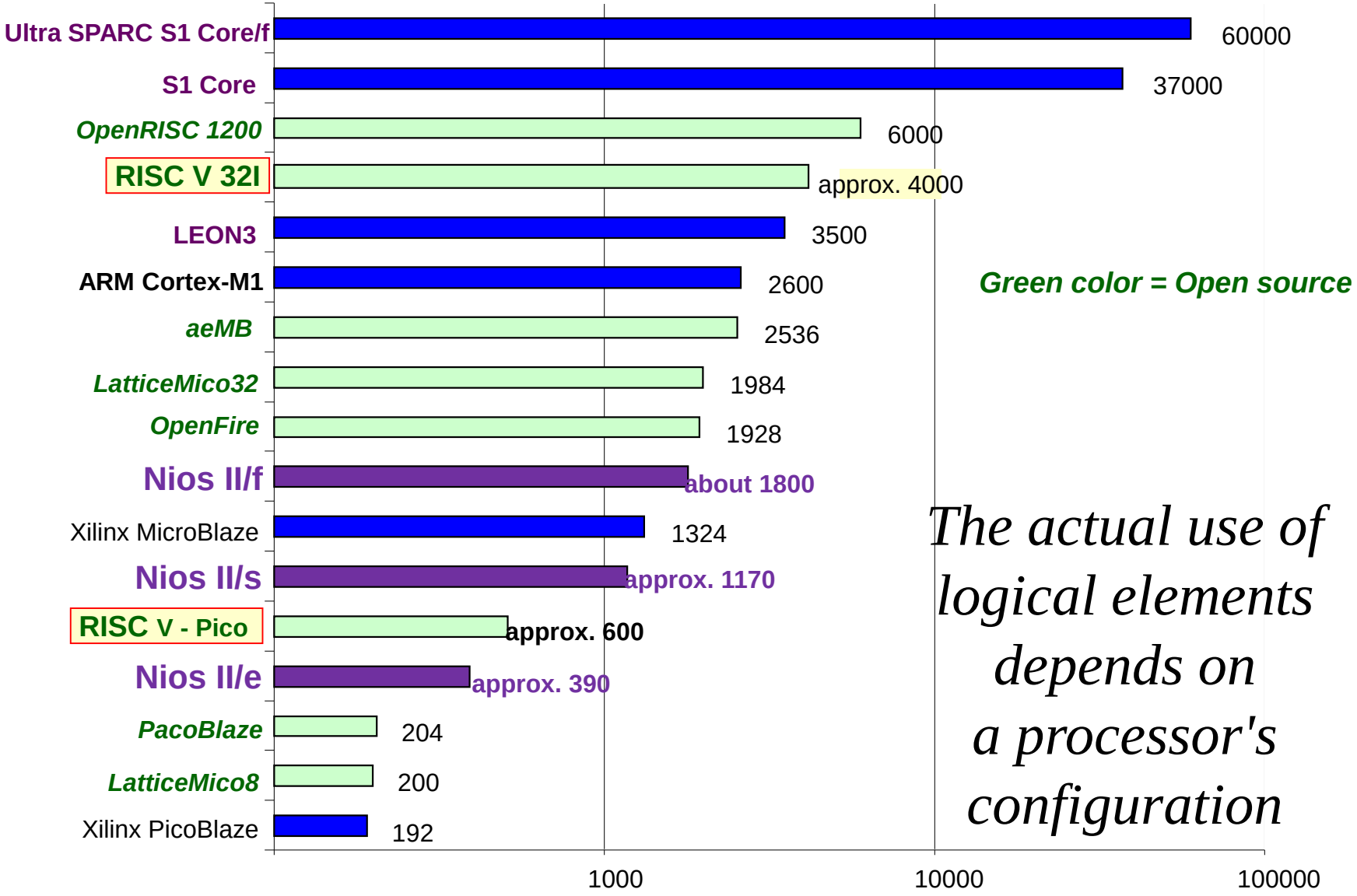
Source: Raj Kamal, Embedded Systems: Architecture, Programming and Design

SoftCore Processors



Picture source: UniQore and TrustGUARD Secure SoftCore Processors for FPGAs

SoftCore processors



Green color = Open source

The actual use of logical elements depends on a processor's configuration

Nios II - available in Quartus

- **Three ISA (Instruction Set Architecture) compatible versions**

- **FAST: Optimised for performance**

- **STANDARD: The middle ground between speed and size**

- **ECONOMY: Minimum size**

- **Software**
  - Binary code remains compatible

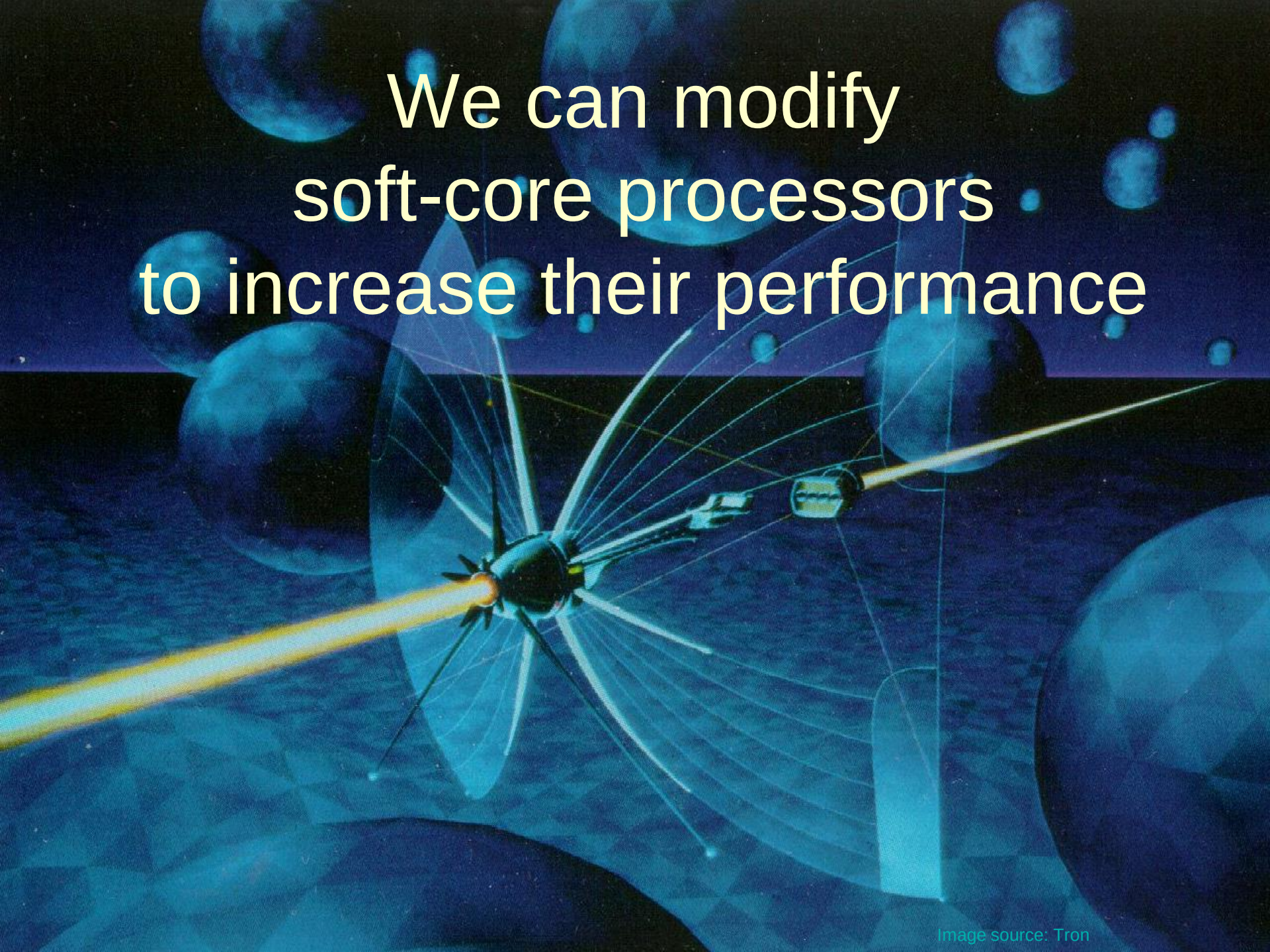
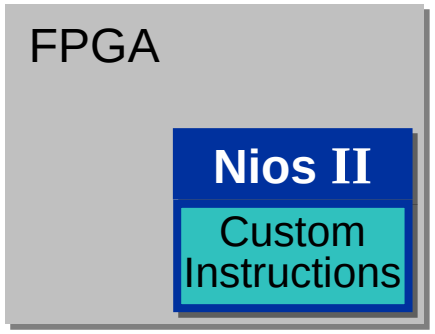


Image source: Tron

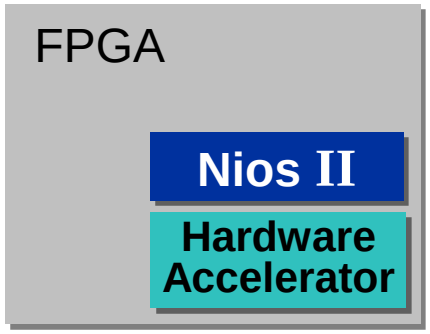
Three possible ways

User instructions



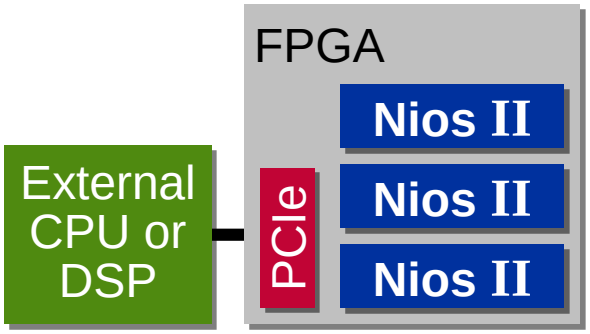
- Let's add instructions suitable for solving our problem

Hardware accelerators



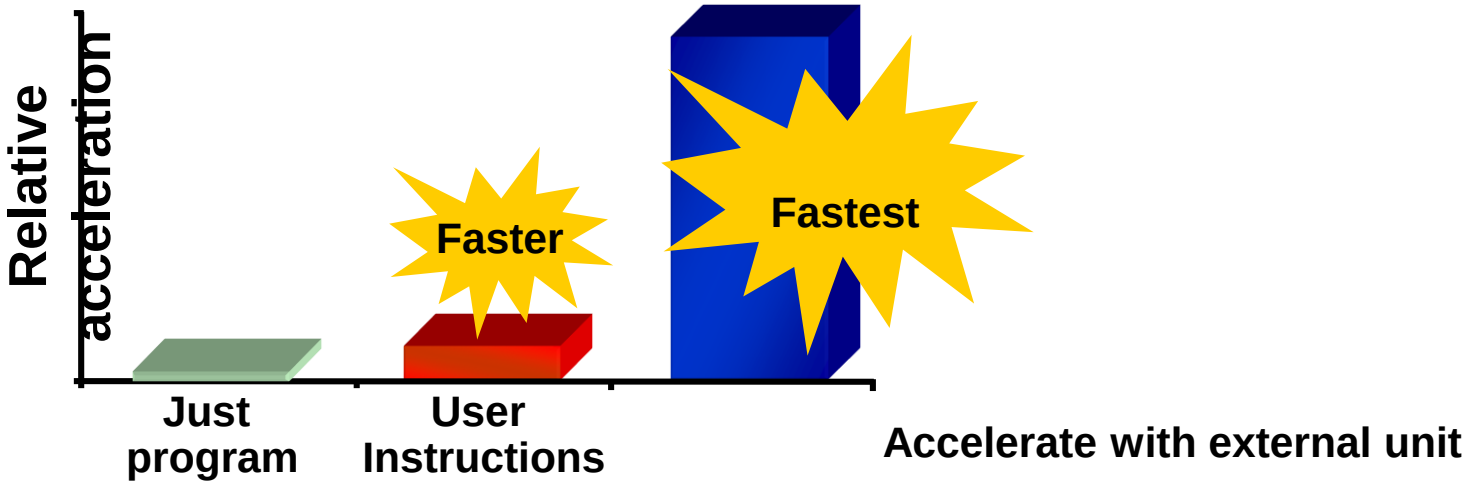
- We add an external circuit to which we transfer part of our solution e.g. our **ControlPanel**

Multi-processor system



- We'll add a few processors

Speeding up the program in FPGA



RISC-V: The Free and Open RISC Instruction Set Architecture

We describe not only his activities,
but also possible wiring in FPGAs.

Image: https://riscv.org/

RISC design strategy

RISC = Reduced Instruction Set Computer

- ***Its design goals:***
  - *speed, size and power reduction, reliability, low cost (← design, manufacturing, test, packaging),*
  - *space on the chip (← in embedded systems).*
- **Its philosophy** - keep everything simple!
  - Fixed length instructions (usually one word);
  - memory instructions only read or write and do nothing else;
  - limited memory addressing modes, and
  - a limited number of instructions - minimum RISC V has only 47 instructions.

*Examples: MIPS, **NIOS**, Sun SPARC, HP PA-RISC, IBM PowerPC, Intel (Compaq), Alpha, **RISC-V**...*

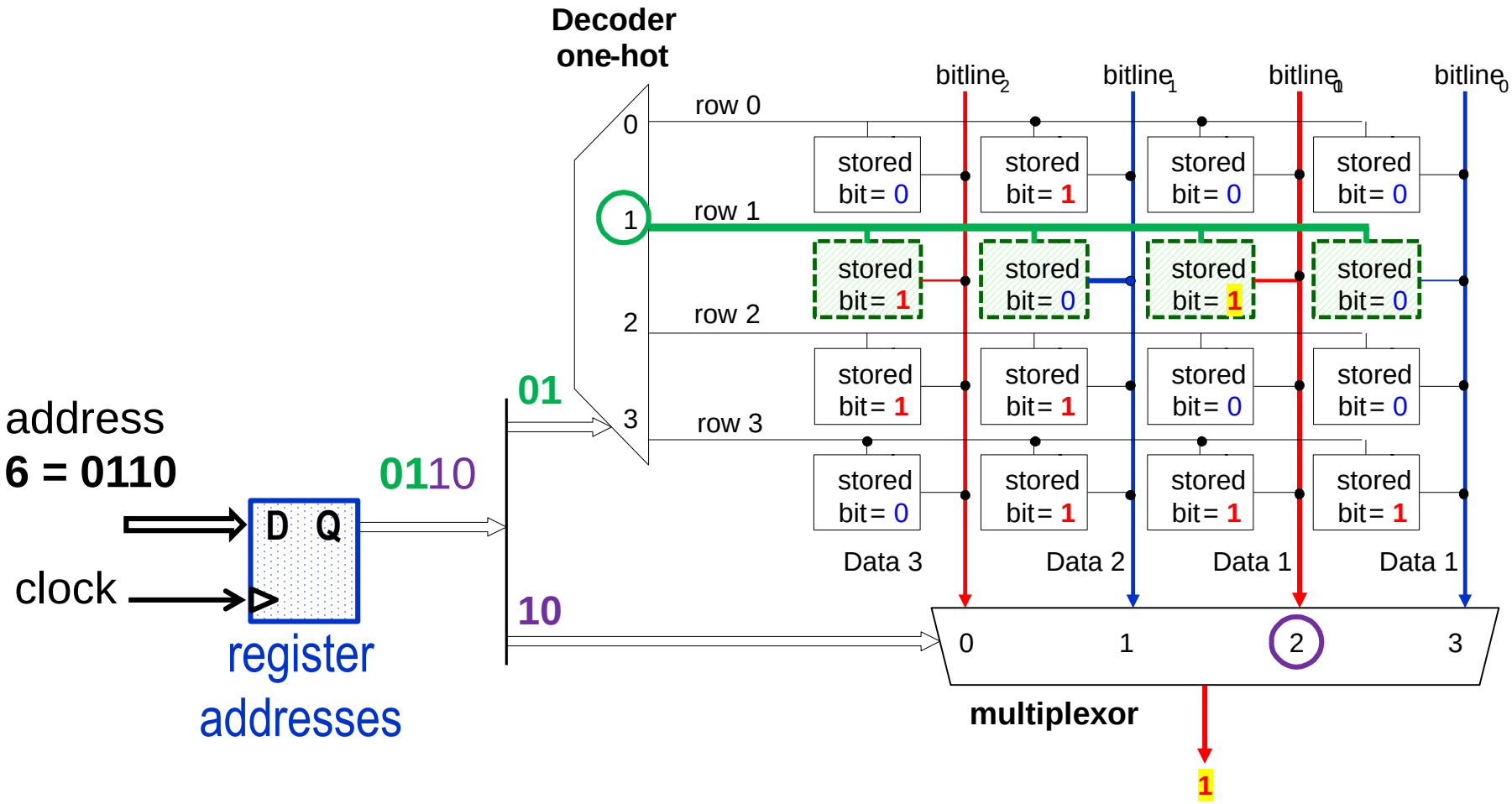
CISC Design Strategy

CISC = Complex Instructions Set Computers

Examples of CISC instructions

Processor	Directions	Activities
Pentium	**MOVS**	copying strings
PowerPC	**cntlzd**	counts consecutive zeros
IBM 360-370	**EN**	compares and swaps registers if the specified condition is met
Digital VAX	**POLYD**	calculate the value of the polynomial according to the table of coefficients

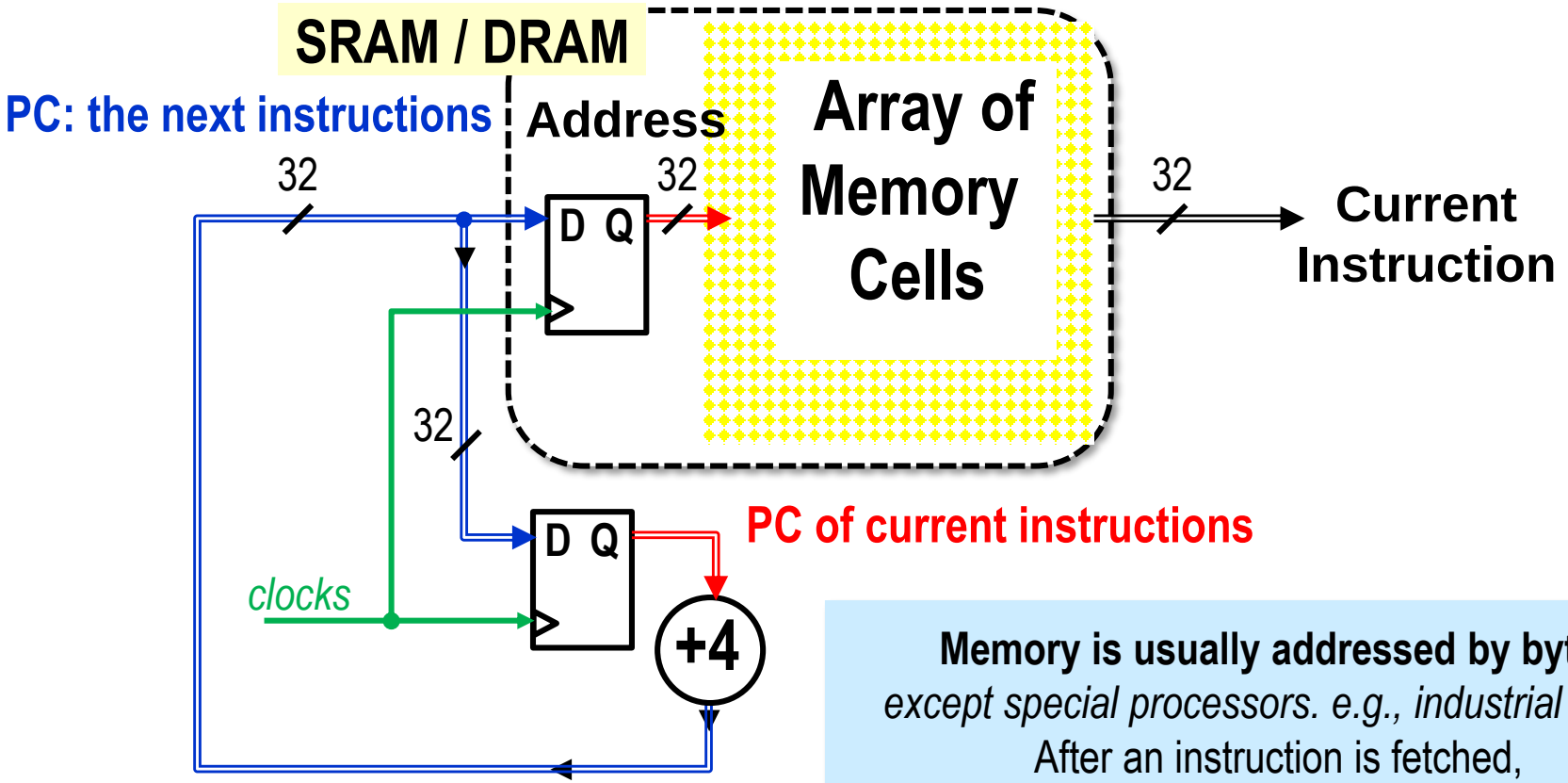
Reminder: memories with a selection matrix need an address register



The selection matrix is used in memories SRAM, DRAM, DDR, FLASH, and others.

32 bit RICS V - Basic PC Cycle (Program Counter)

Fetch instruction from memory = stage: **FETCH**



Memory is usually addressed by bytes
except special processors. e.g., industrial PLCs.
After an instruction is fetched,
PC is moved to the next instruction
(default action), i.e., PC+4 on a 32-bit processors

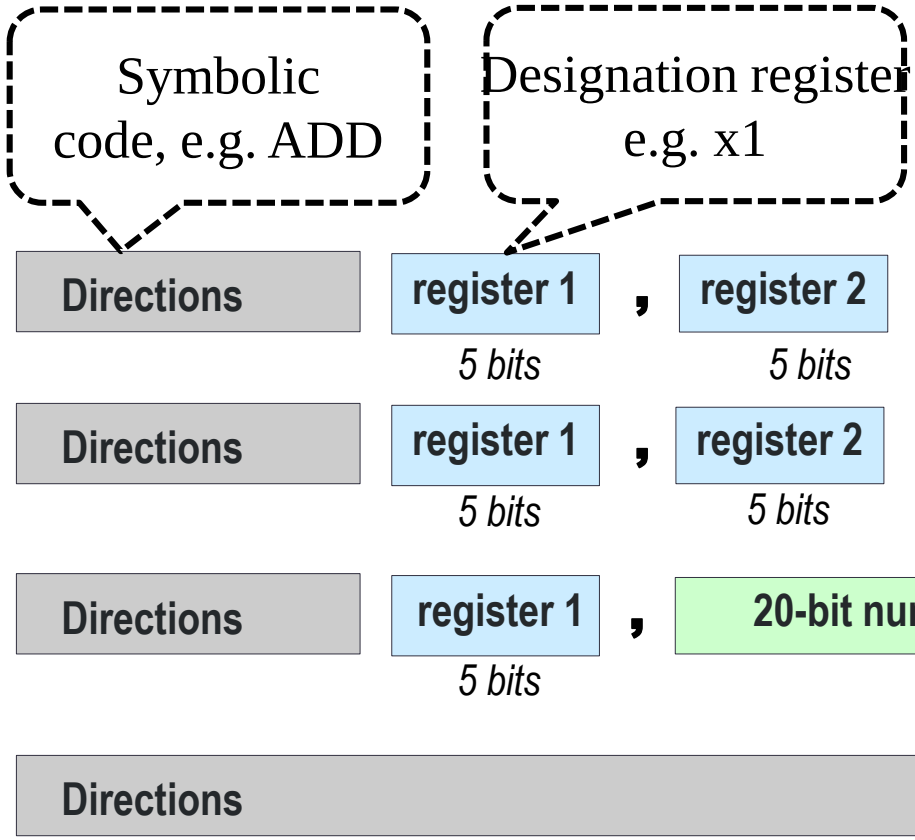
32 bit RISC-V : Register names

Reg	Alternative name
x0	**zero** = constant 0, cannot be overwritten
x1	ra
x2	sp
x3	gp
x4	tp
x5	t0
x6	t1
x7	t2
x8	s0/fp
x9	s1
x10	a0
x11	a1
x12	a2
x14	a3
x14	a4
x15	a5

Reg	Alternative name
x16	a6
x17	a7
x18	s2
x19	s3
x20	s4
x21	s5
x22	s6
x23	s7
x24	s8
x25	s9
x26	s10
x27	s11
x28	t3
x29	t4
x30	t5
x31	t6

RISC V 32I - instructions

The RISC V 32I base processor has 32-bit instructions and its minimum version contains 47 of them.
In symbolic assembler, they are written in the form:



Examples of instructions

ADD x1, x2, x3 // add registers
x1 ← x2 + x3

ORI x1, x2, 15 // bit or
x1 ← x2 or 0xF

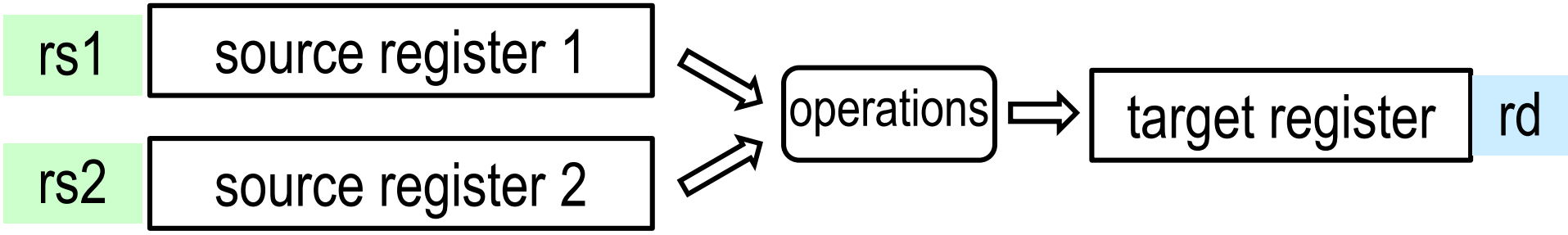
LUI x1, 0x12345 load into the upper bits of the 32-bit register
x1=|0x12345 | 0x000 |

EBREAK - breakpoint, debugging instructions

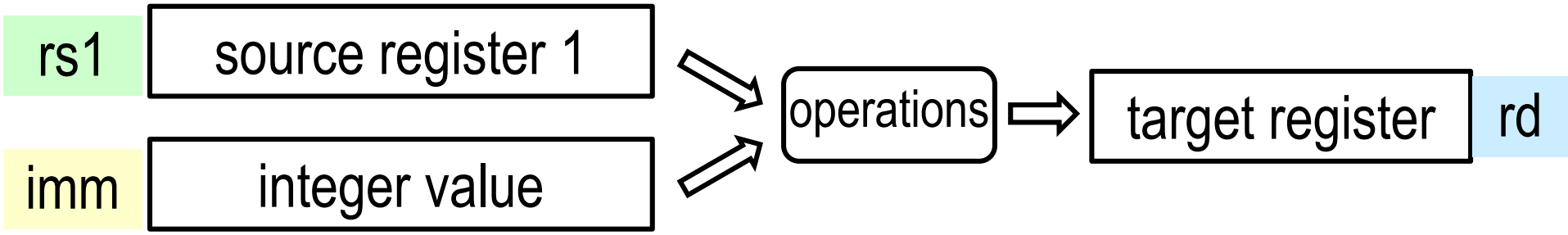
RISC V mostly works with registers

rs - register source

rd - register destination



example **add x3, x1, x2** *operation: x3←x1+x2*



example **addi x4, x4, -1** *operation: x4←x4 -1 ≡ x4--*

Macroinstructions

Instructions that are not in the basic set are often substituted with macro instructions. These are converted by the assembler compiler to other or a sequence of them, e.g.

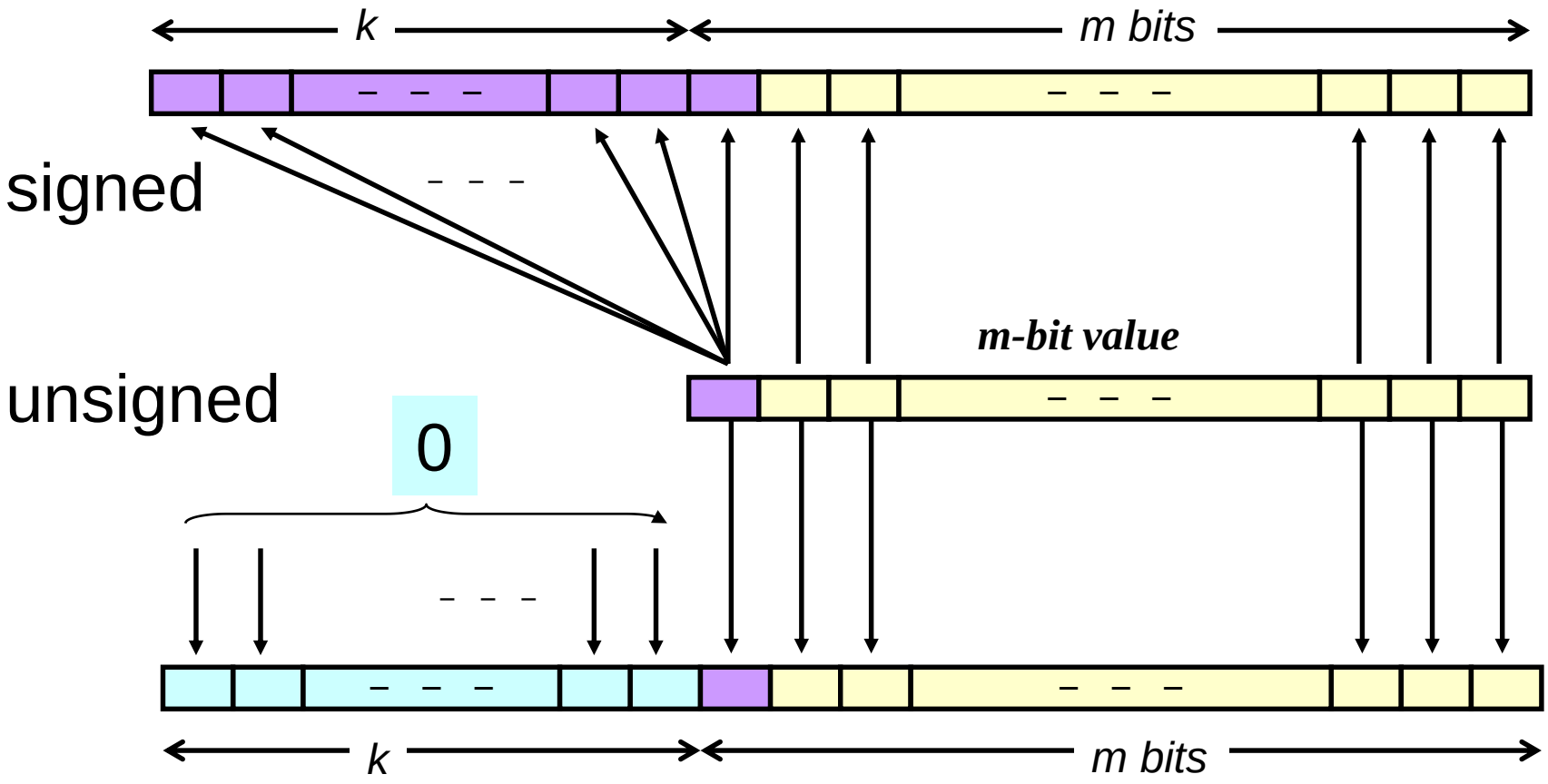
LI x1, 0x12345678 // macroinstruction x1 ← 0x12345678
(*Load Immediate*)

is transferred to the pair

LUI x1,0x12345 // x1 ← *0x12345000* Load Upper Immediate
ORI x1, 0x678 // x1 ← x1 or 0x678 OR Immediate

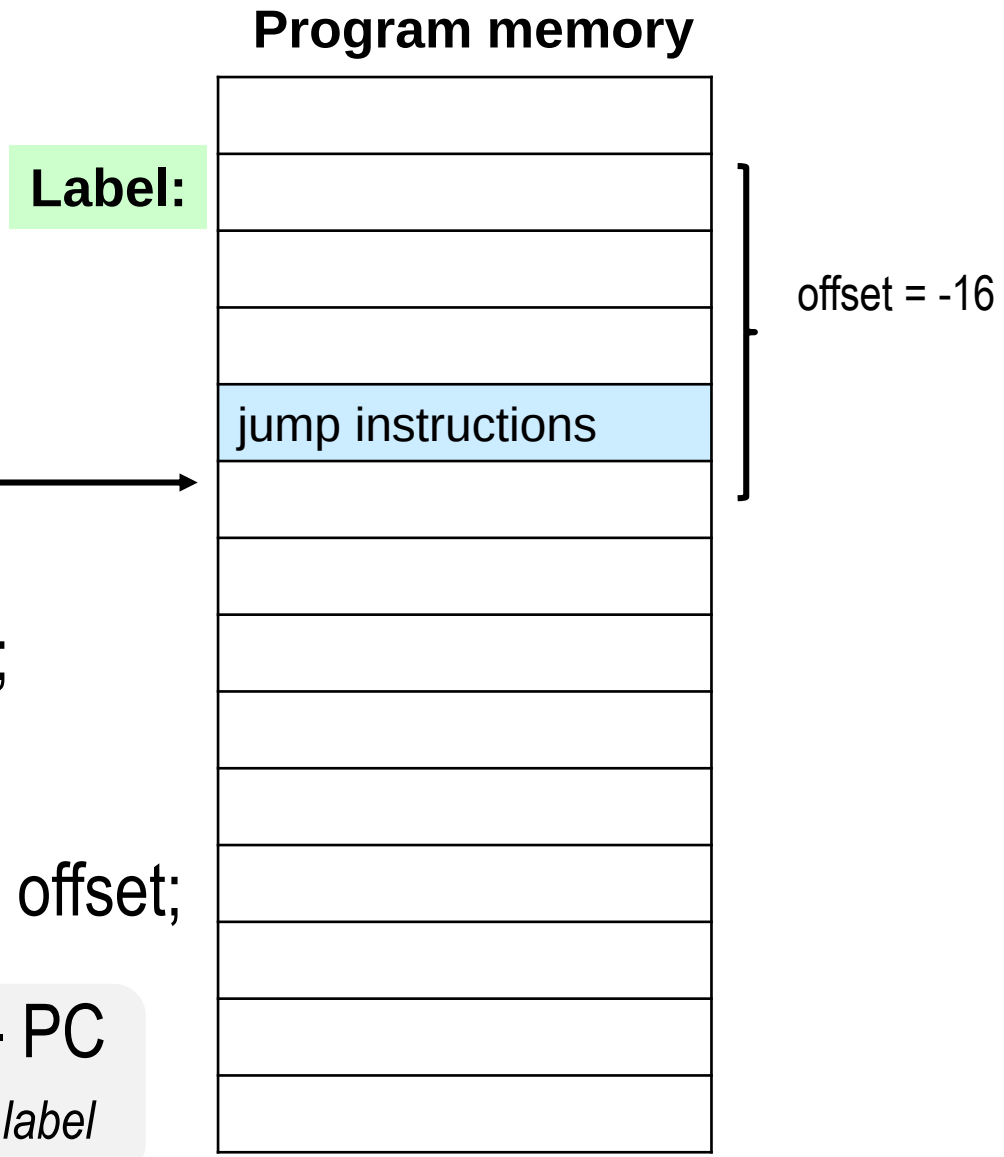
Extending a number using Unsigned/Signed Extension (signed extension)

The extension type is always derived from the code of the instruction being executed. CPU registers do not know types in the C sense, they only store values.



Conditional jumps in the program

The destination address of the jump is most often stated relative to the instantaneous value of the PC



cmp == != < >=

if(xi cmp xj) goto Label;

translated as

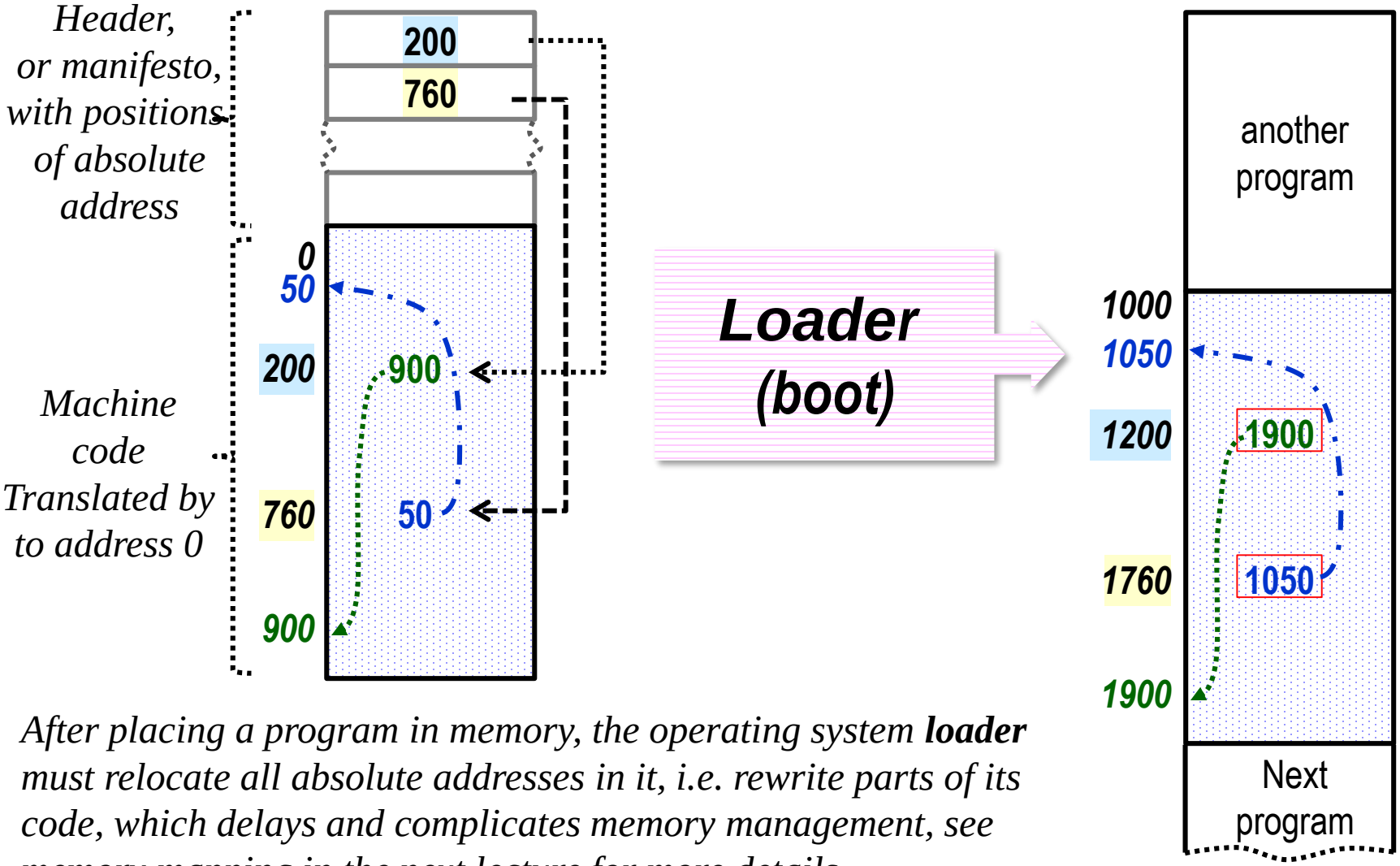
if(xi cmp xj) PC = PC + offset;

offset = &label - PC

where &label =*address label*

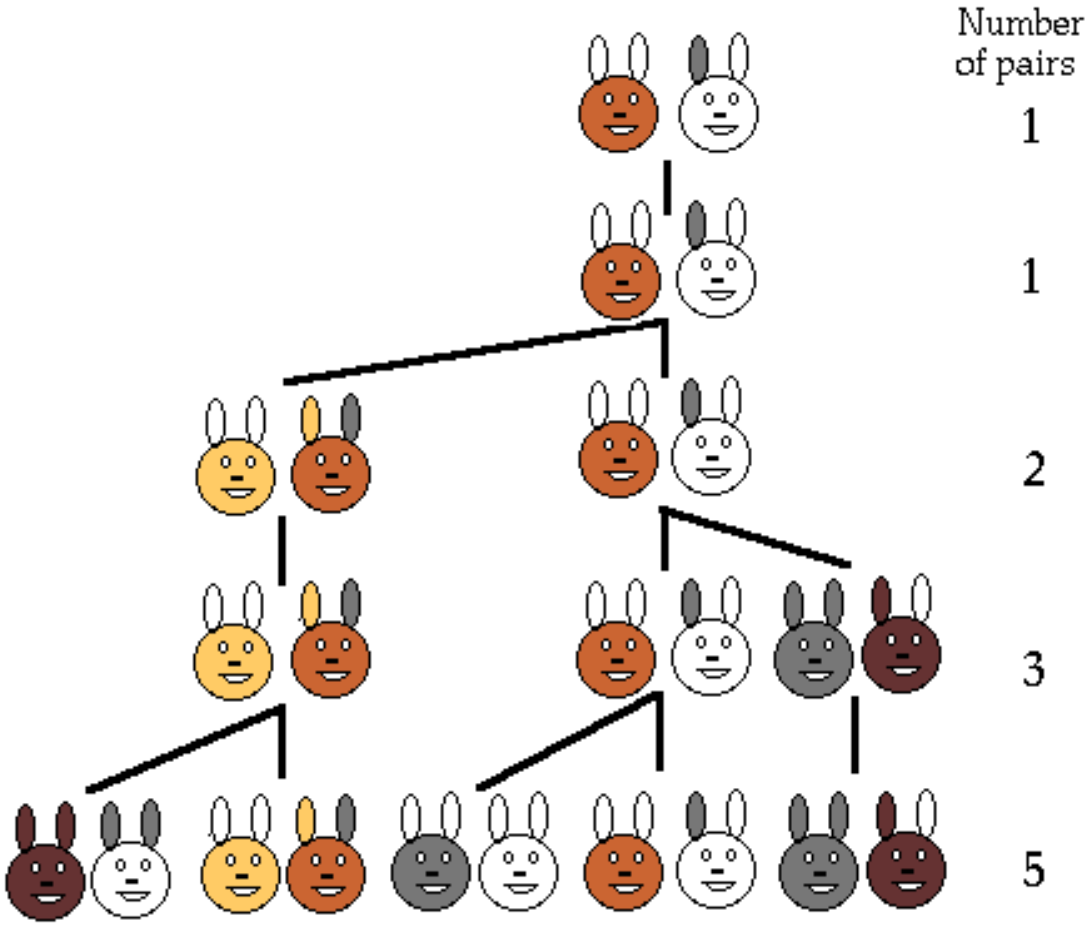
Position-dependent program file with absolute addresses

Program in memory



*After placing a program in memory, the operating system **loader** must relocate all absolute addresses in it, i.e. rewrite parts of its code, which delays and complicates memory management, see memory mapping in the next lecture for more details.*

Fibonacci model AD 1202 of immortal rabbits in assembler



Source : https://r-knott.surrey.ac.uk/Fibonacci/fibnat.html

Author of the Fibonacci Sequence

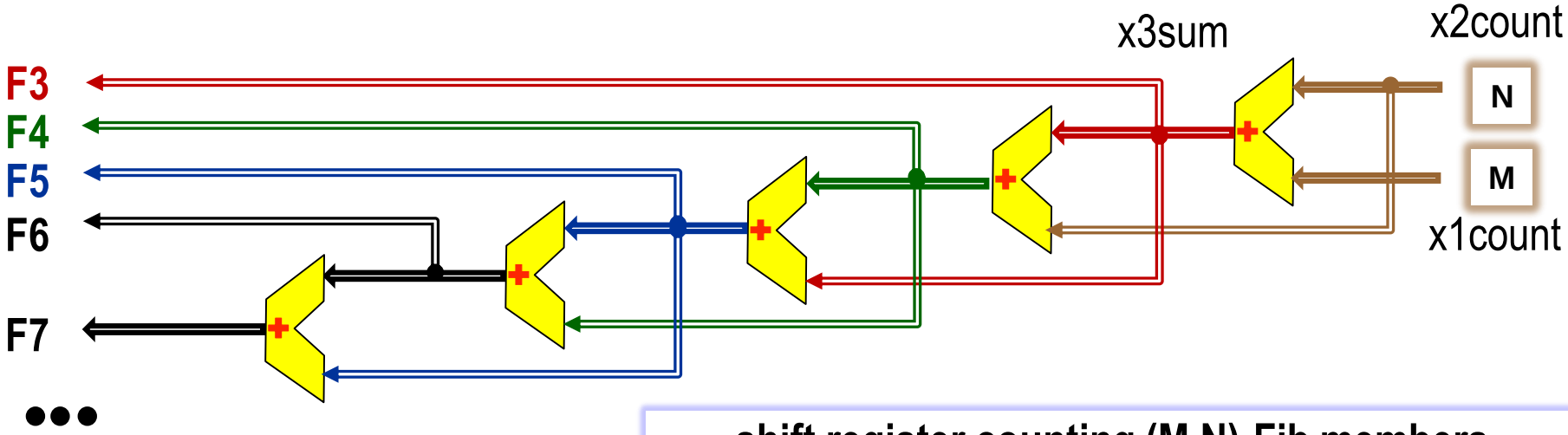
The sequence was named after Fibonacci, but it was based on the *Chandaḥśāstra* sequence described by an Indian mathematician named **Acharya Pingala**, who lived sometime between the 3rd and 2nd centuries BC and is also known as the first to describe the **binary** number system, see https://en.wikipedia.org/wiki/Pingala for more details.

There are also generalized Fibonacci sequences of the type Fib(M,N) that start with arbitrary numbers M and N. Commonly used has M=1 and N=1, see https://en.wikipedia.org/wiki/Generalizations_of_Fibonacci_numbers

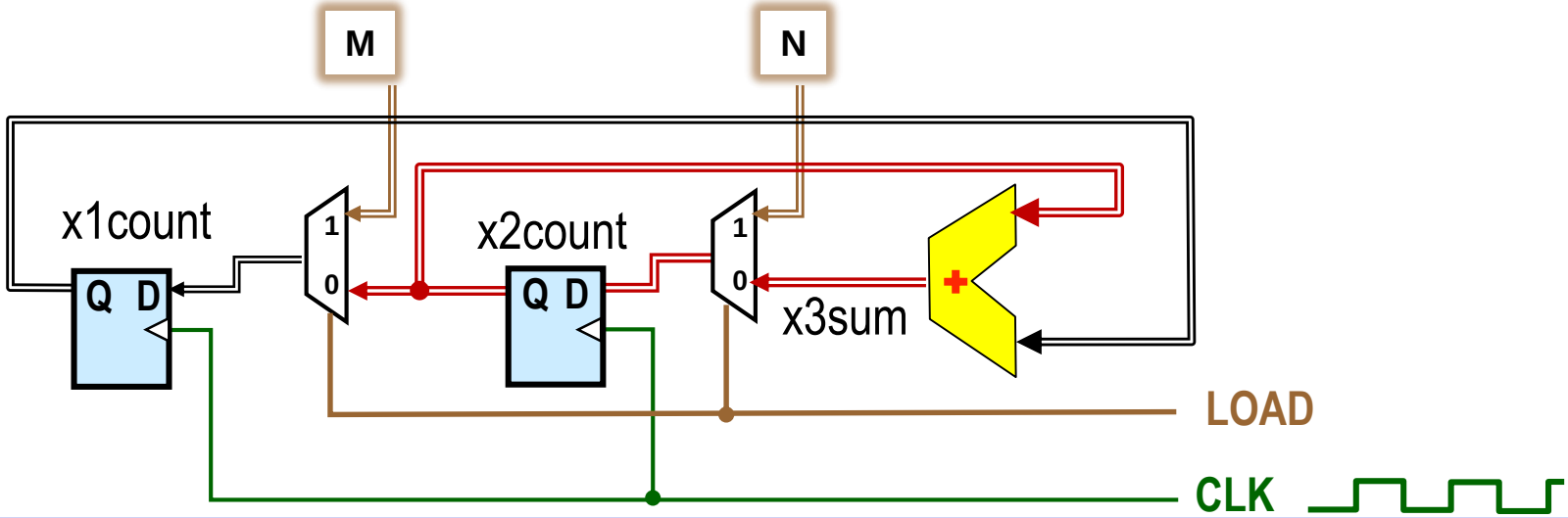
Image Indica.today

Algorithm (1,1)-Fibonacci seq. in C

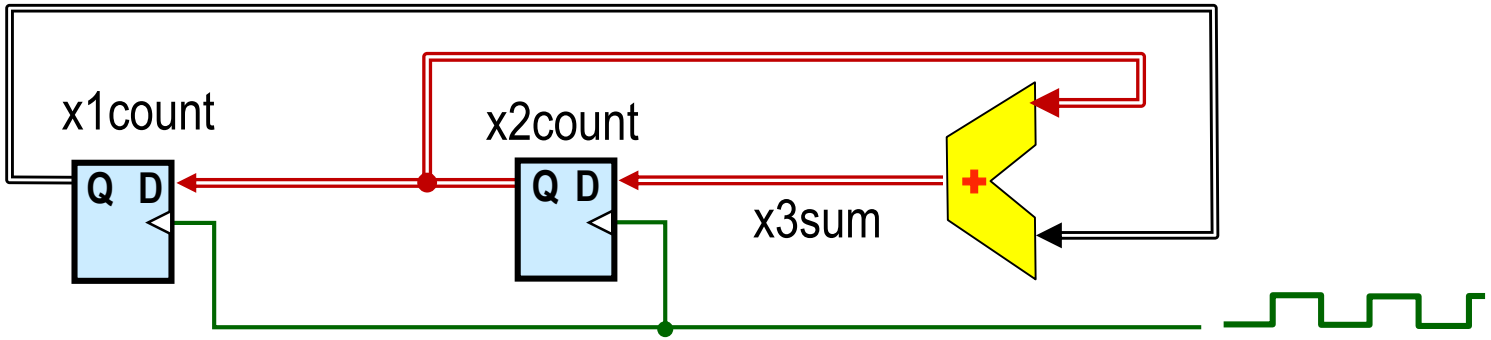
combinational circuit counting (M,N)-Fib terms



shift register counting (M,N)-Fib members



Algorithm (1,1)-Fibonacci seq. in C




```
int x3sum, x1count = 1, x2count = 1;
for (int x4loop = 10; x4loop > 0; x4loop--)
{
    x3sum = x1count + x2count;
    x1count = x2count; x2count = x3sum;
}
```

- *Structured programming is not used in assembler!*
- *The algorithm must be decomposed into simple operations, that respect the instructions of the processor.*

First we decompose the "for" loop

```
int x3sum, x1count = 1, x2count = 1;
for (int x4loop = 10; x4loop > 0; x4loop--)
{
   x3sum = x1count + x2count;
   x1count = x2count; x2count = x3sum;
}
```

for is a shortened notation of **while** loop

```
int x4loop = 10;
while (x4loop > 0);
{ x3sum = x1count + x2count;
   x1count = x2count; x2count = x3sum;
   x4loop--;
}
```

Conversion to assembler operations

```
int x3sum, x1count = 1, x2count = 1;
int x4loop = 10;
while (x4loop > 0)
{ x3sum = x1count + x2count;
   x1count = x2count; x2count = x3sum;
   x4loop--;}
```

The Risc-V assembler does not know while *- we convert it to* goto *by two jumps in the instruction sequence and a reverse comparison condition:*

```
        int x3sum, x1count = 1, x2count = 1;
        int x4loop = 10;
Loop:   if( x4loop <= 0 ) goto Next;
        x3sum = x1count + x2count;
        x1count = x2count; x2count = x3sum;
        x4loop--;
        if(0==0) goto Loop;
Next:
```

The do-while cycle

If the loop is sure to run at least once, the C compiler will automatically convert it to a do-while with only one jump.

```
int x3sum, x1count = 1, x2count = 1;
int x4loop = 10;
to // only 1 jump
{ x3sum = x1count + x2count; x1count = x2count; x2count = x3sum;
  x4loop--;
} while (0 < x4loop);
```

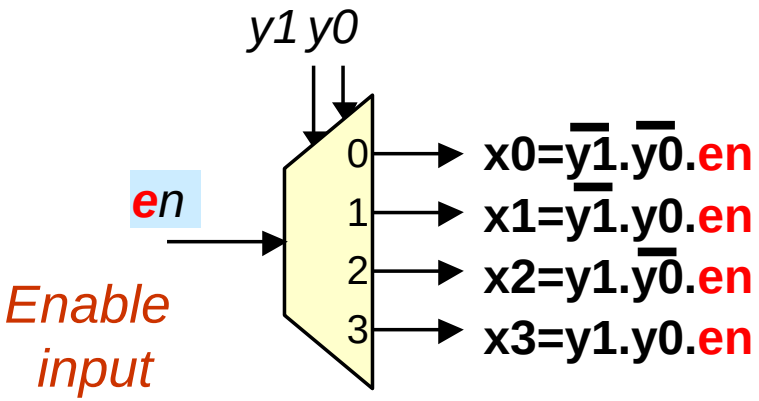
```
        int x3sum, x1count = 1, x2count = 1;
        int x4loop = 10;
Loop:   x3sum = x1count + x2count;
        x1count = x2count; x2count = x3sum;
        x4loop--;
        if(0<x4loop) goto Loop;
```

RISC V assembler

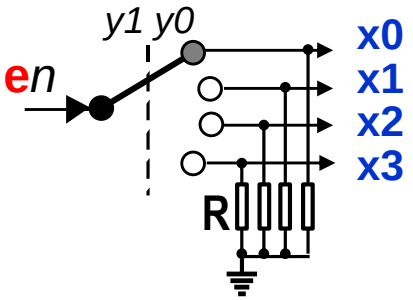
Registry	
zero	always 0
x1	step t-1
x2	step t
x3	step t+1
x4	counter

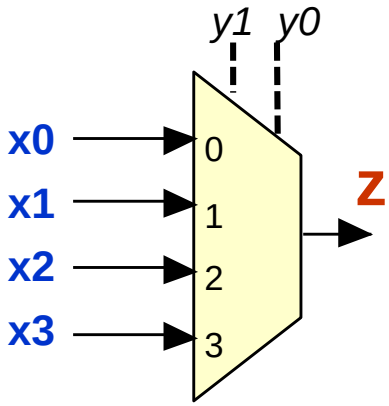
```
_start: ori x1, zero, 1 // x1←0 or 1
        ori x2, zero, 1 // x2←0 or 1
        ori x4, zero, 10 // x4←0 or 10
loop:
        add x3, x1, x2 // x3←x1 + x2
        or x1, x2, zero // x1←x2 or 0 ≡ x1 ← x2
        or x2, x3, zero // x2←x3 or 0 ≡ x2 ← x3
        addi x4, x4, -1 // x4←x3 + -1 ≡ x4- -
        blt zero, x4, loop // if(0<x4) goto loop;
```

Reminder: Demultiplexor and Multiplexor



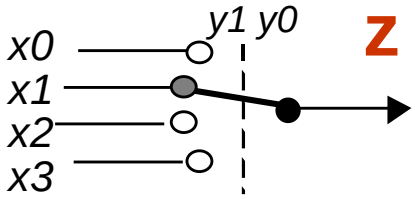
	y1	y0	x0	x1	x2	x3
0	0	0	en	0	0	0
1	0	1	0	en	0	0
2	1	0	0	0	en	0
3	1	1	0	0	0	en



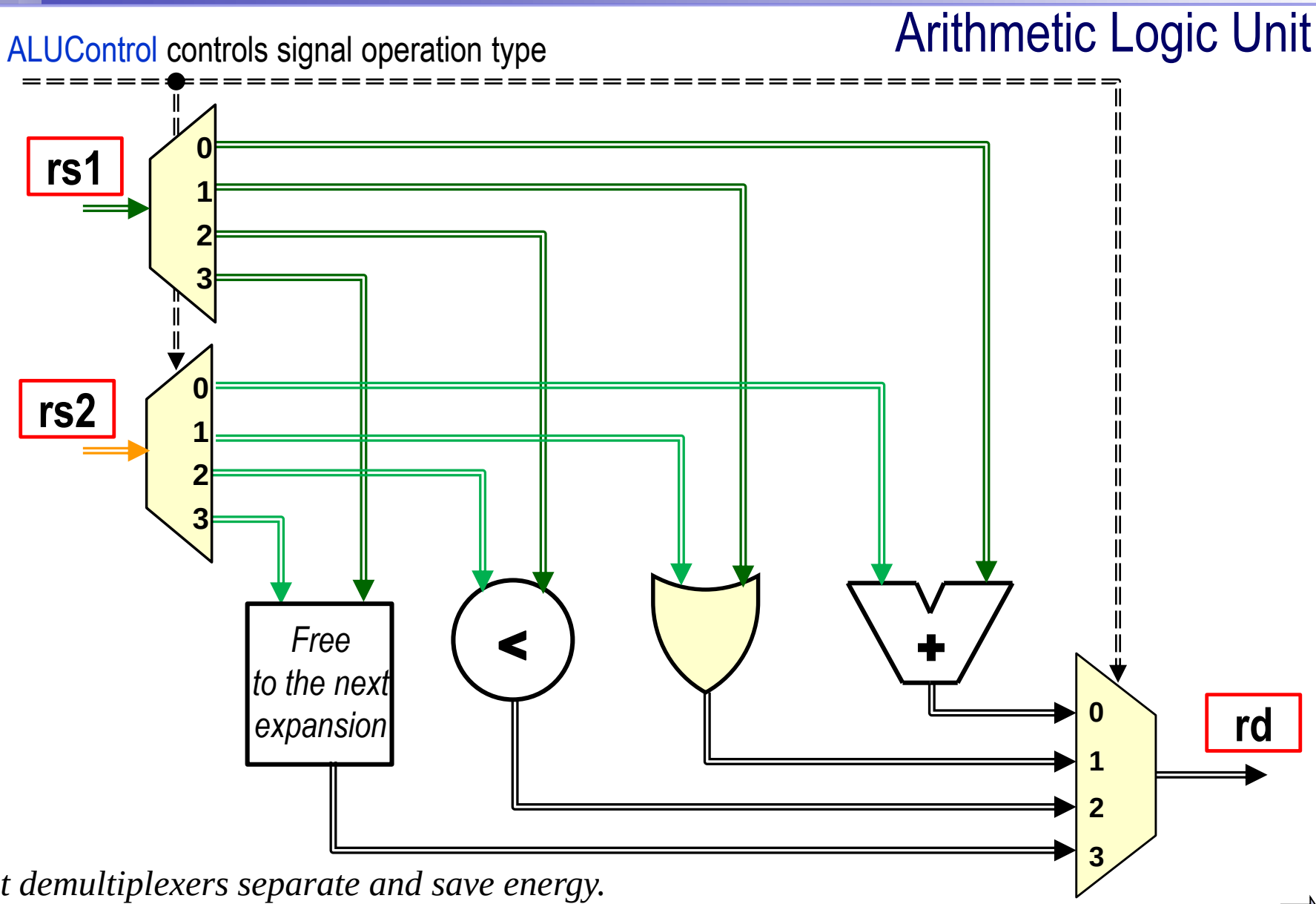


$$z = \overline{y1}.\overline{y0}.x0 + \overline{y1}.y0.x1 + y1.\overline{y0}.x2 + y1.y0.x3$$

	y1	y0	z
0	0	0	x0
1	0	1	x1
2	1	0	x2
3	1	1	x3

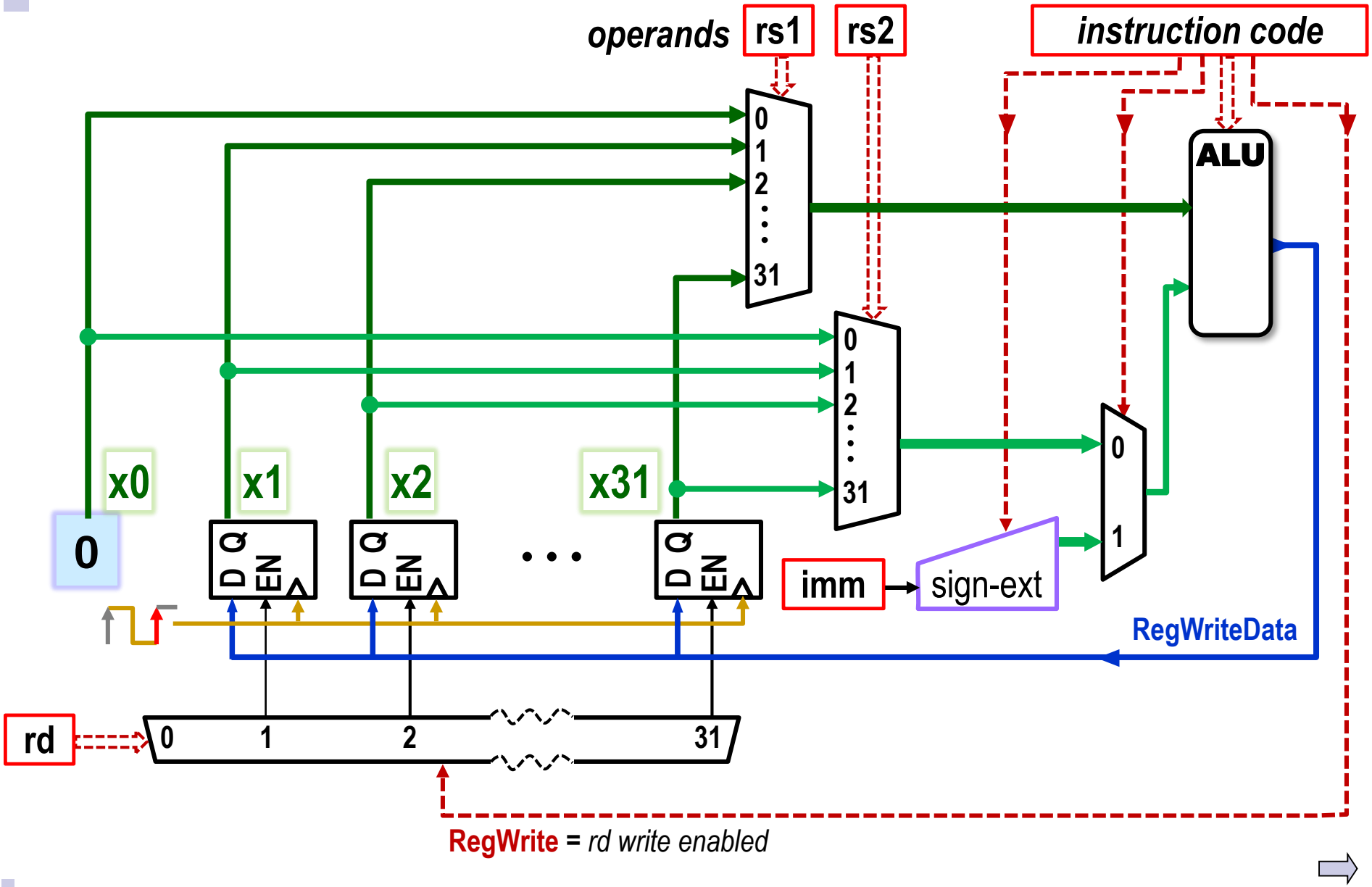


Our ALU - Arithmetic Logic Unit

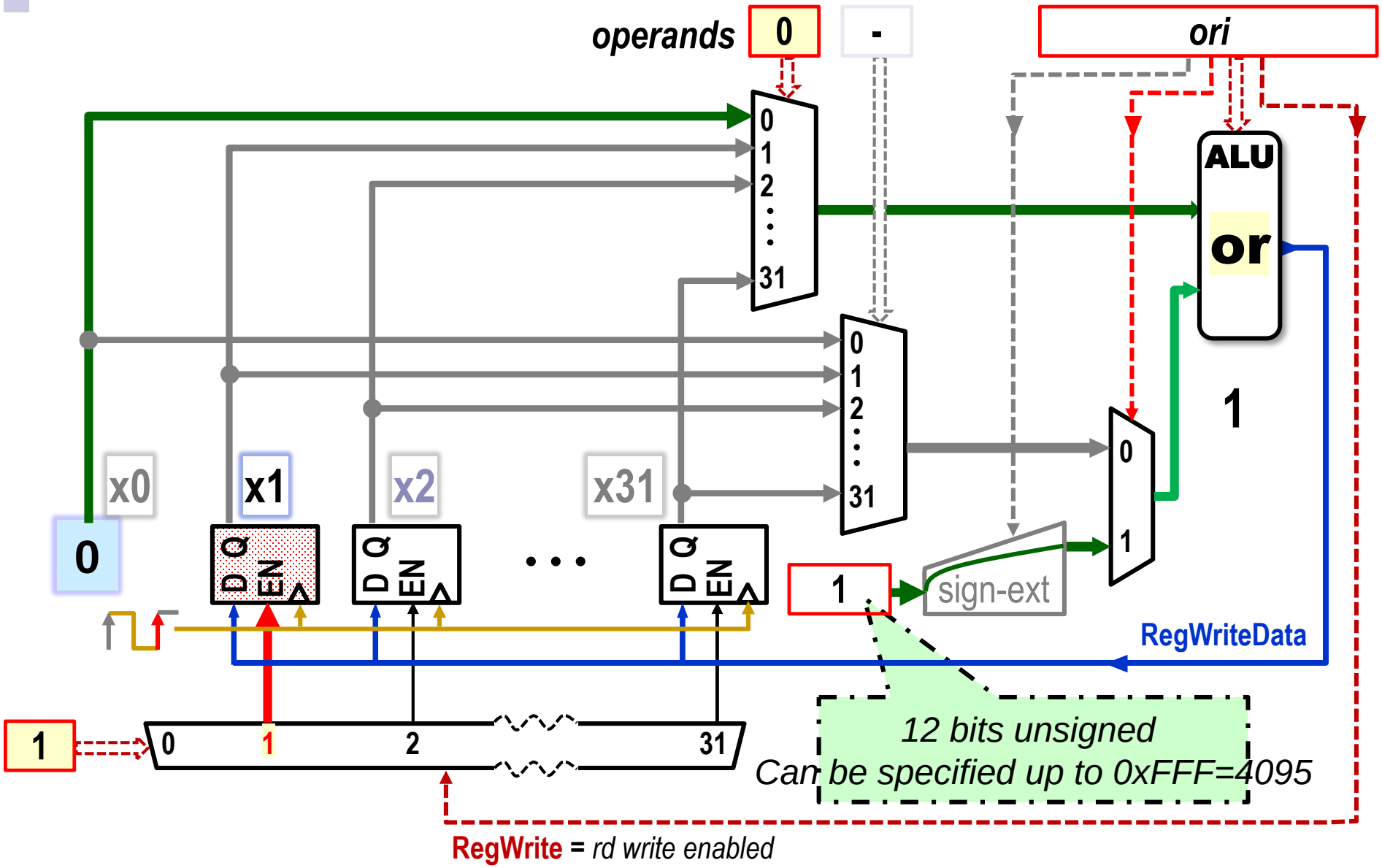


Input demultiplexers separate and save energy.
Only the active computing unit is working.

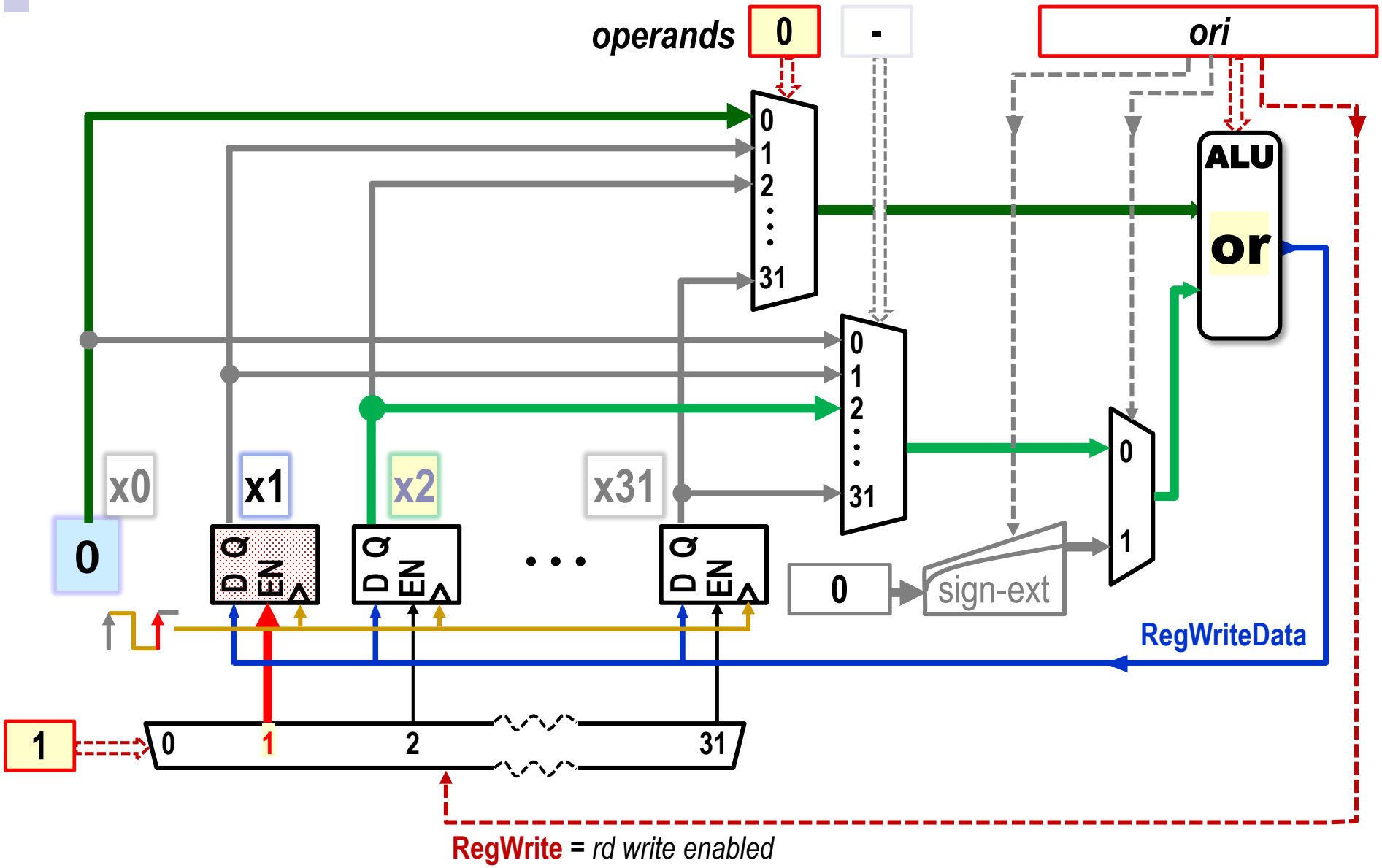
Data flow during register operations



Directions: ori x1, zero, 1 ≡ x1←1

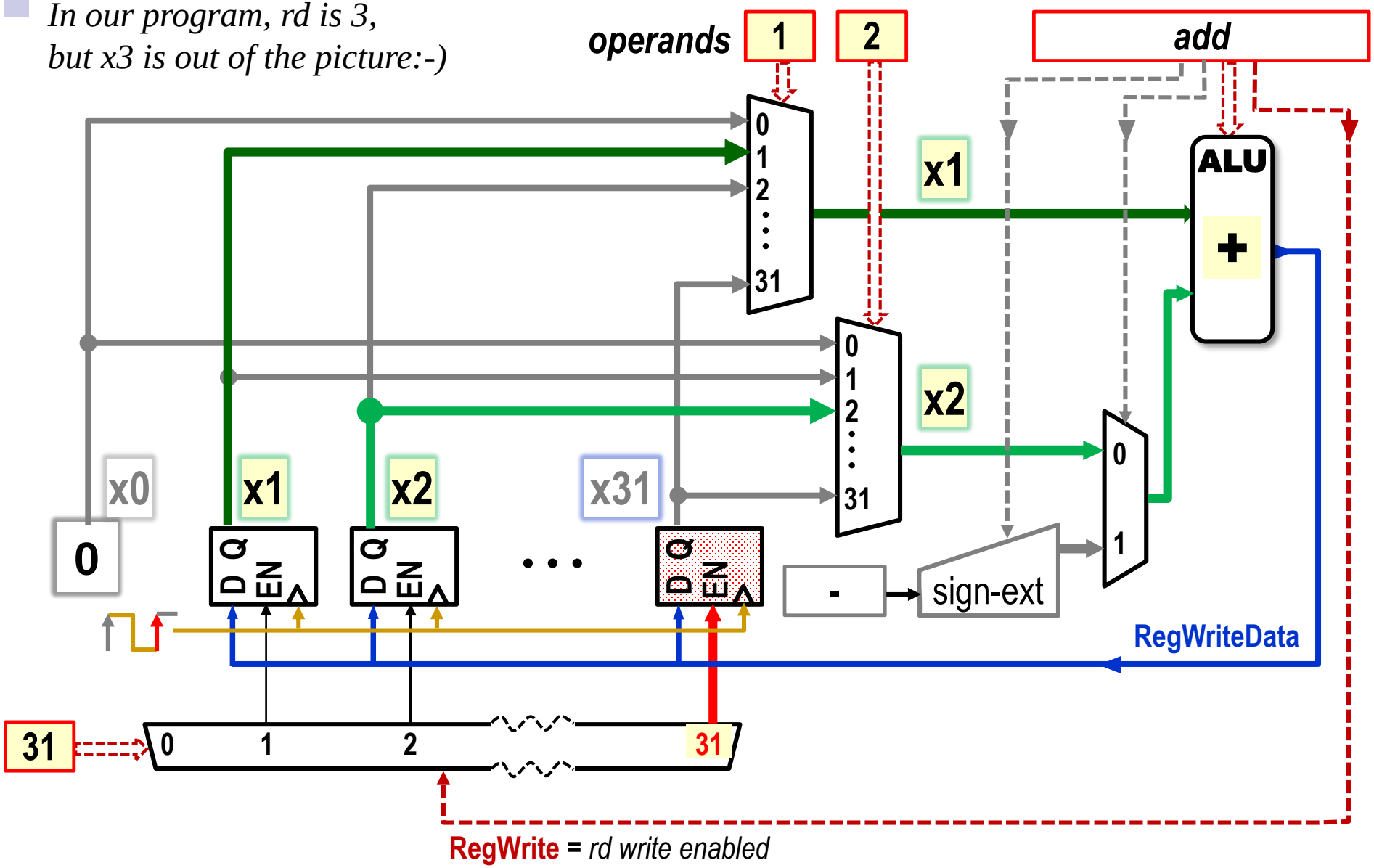


or x1, x0, x2 ≡ x1←x2



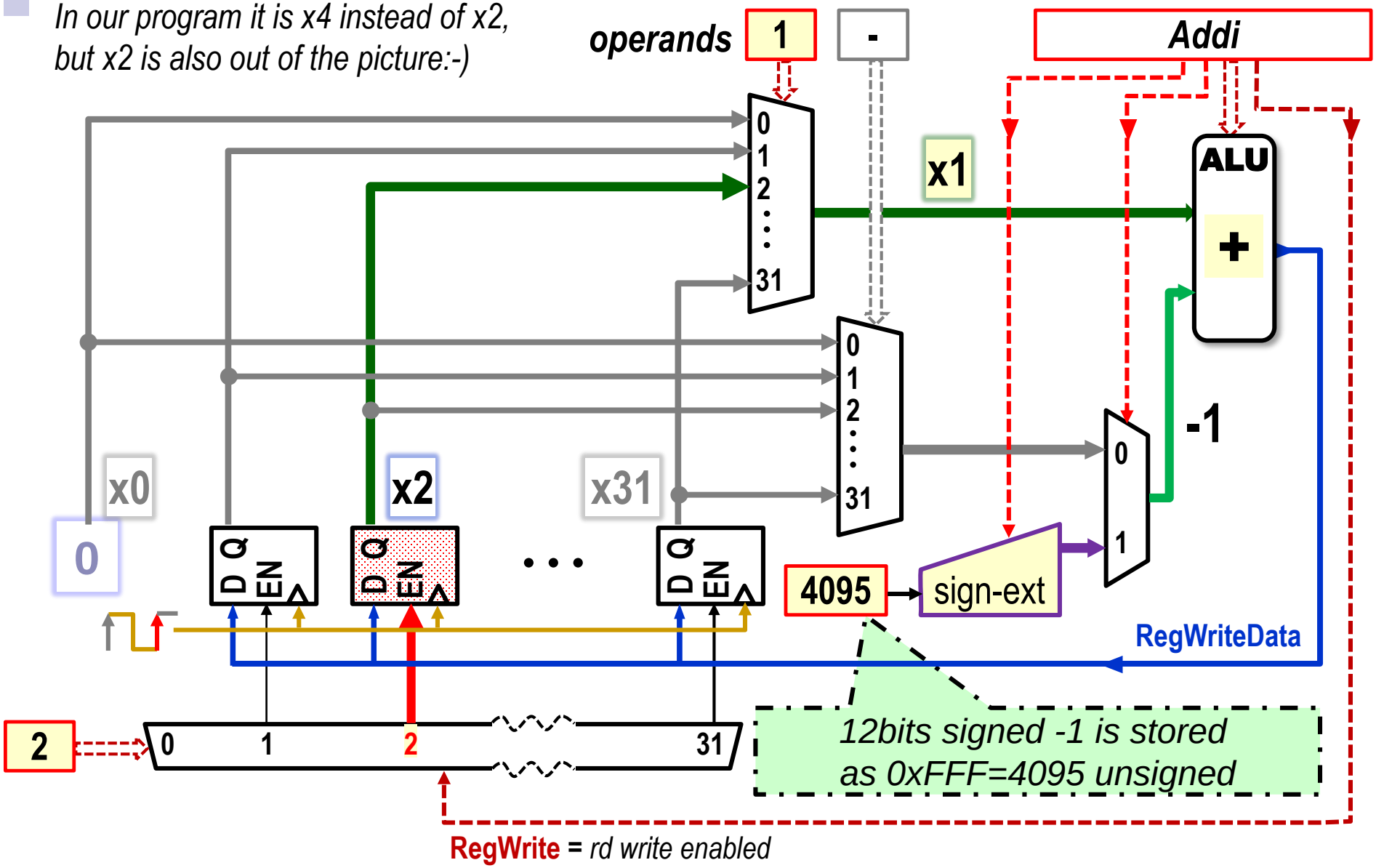
add x31,x1,x2 ≡ x31←x1+x2

In our program, rd is 3,
but x3 is out of the picture:-)

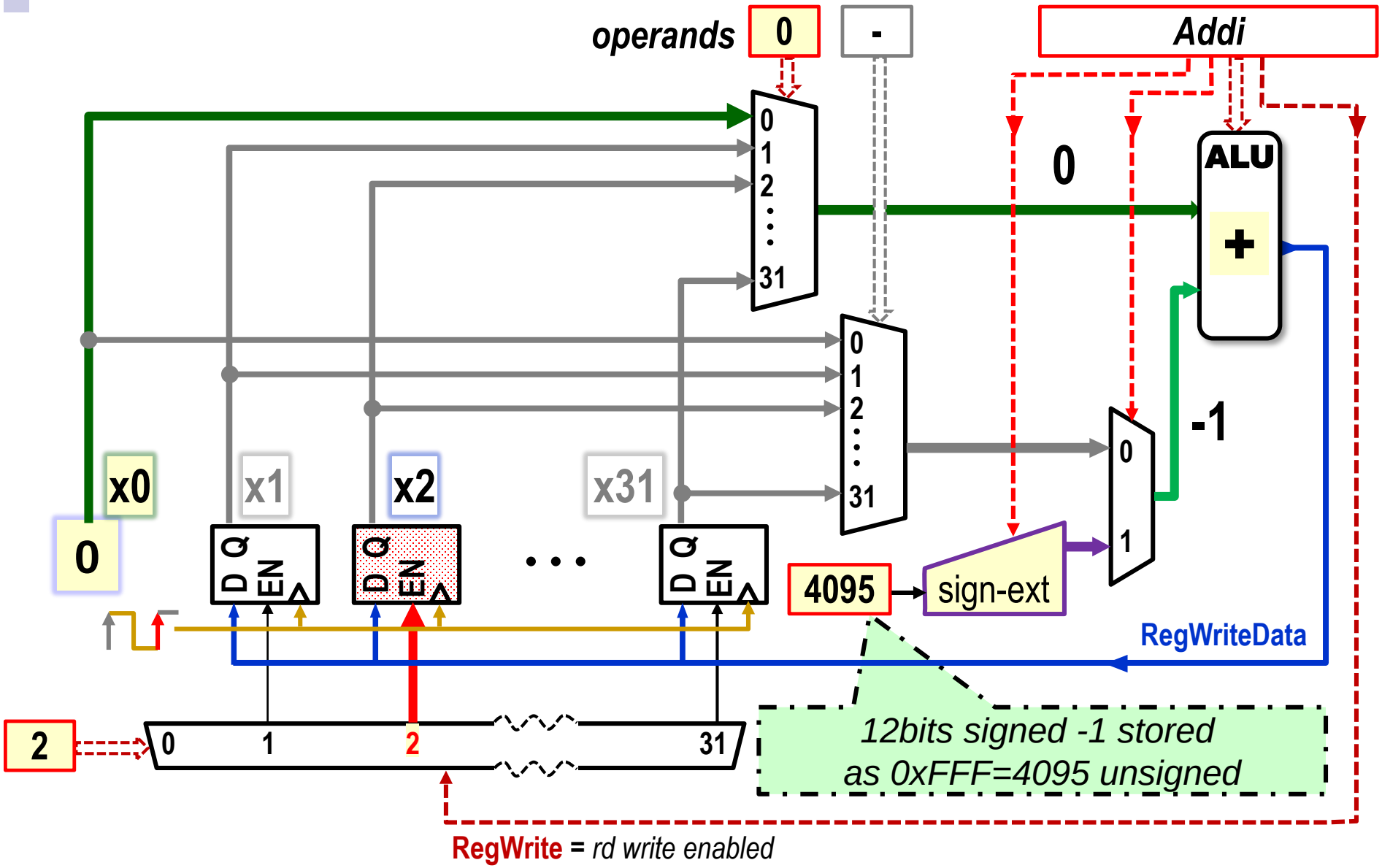


addi x2, x2, -1 ≡ x2←x2-1

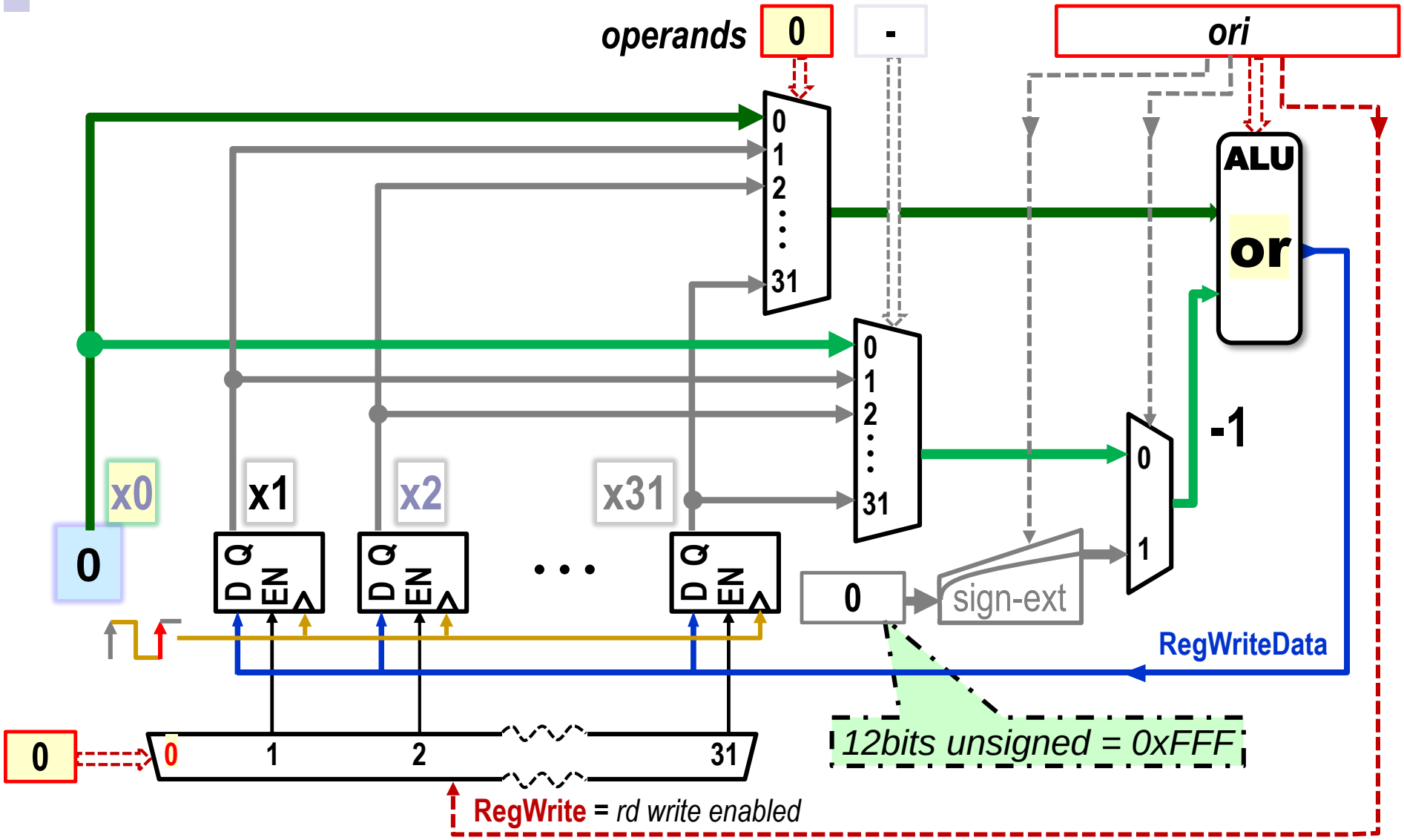
In our program it is x4 instead of x2, but x2 is also out of the picture:-)



addi x2, x0, -1 ≡ x2 ← -1

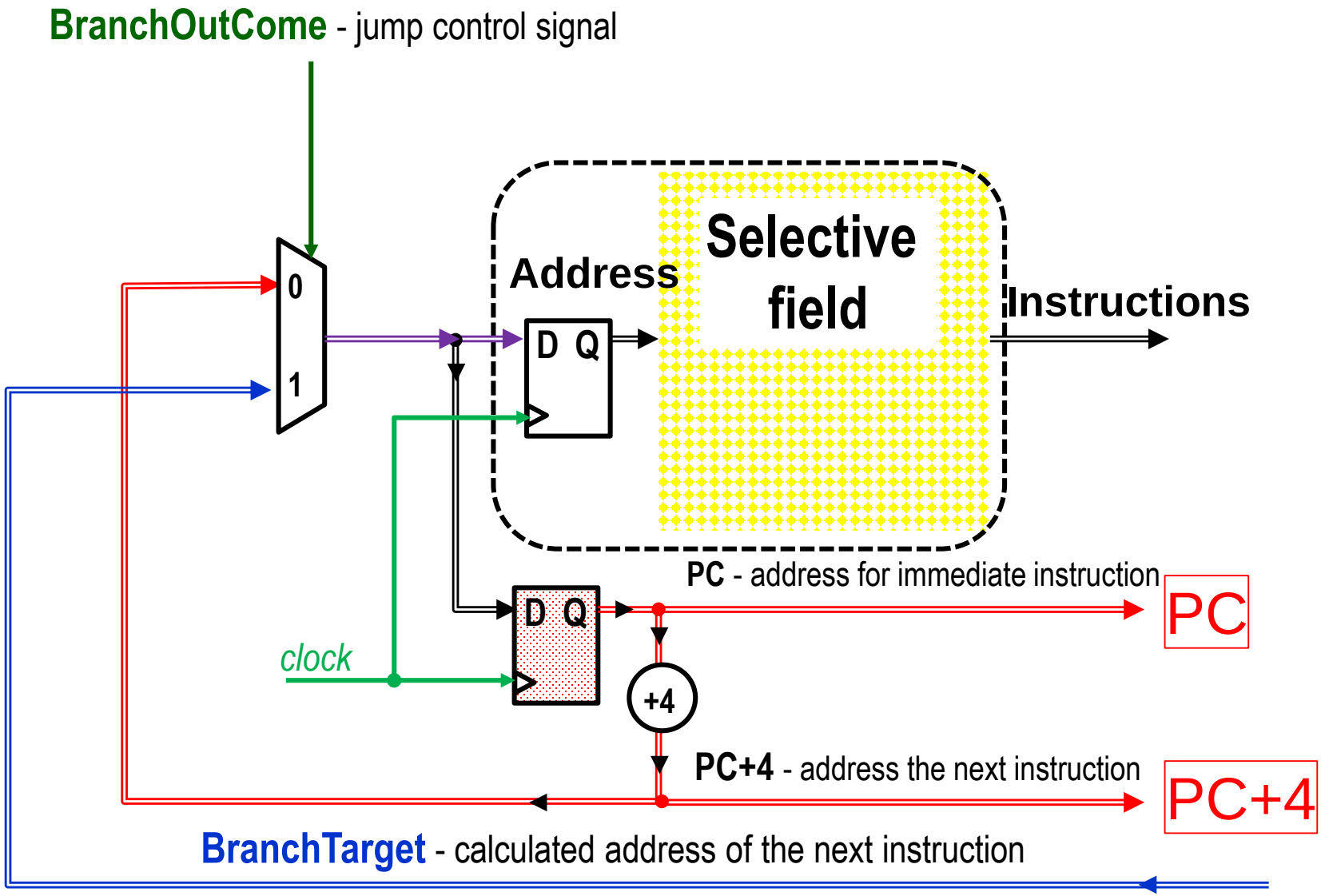


NOP = No Operation ≡ x0←x0+x0

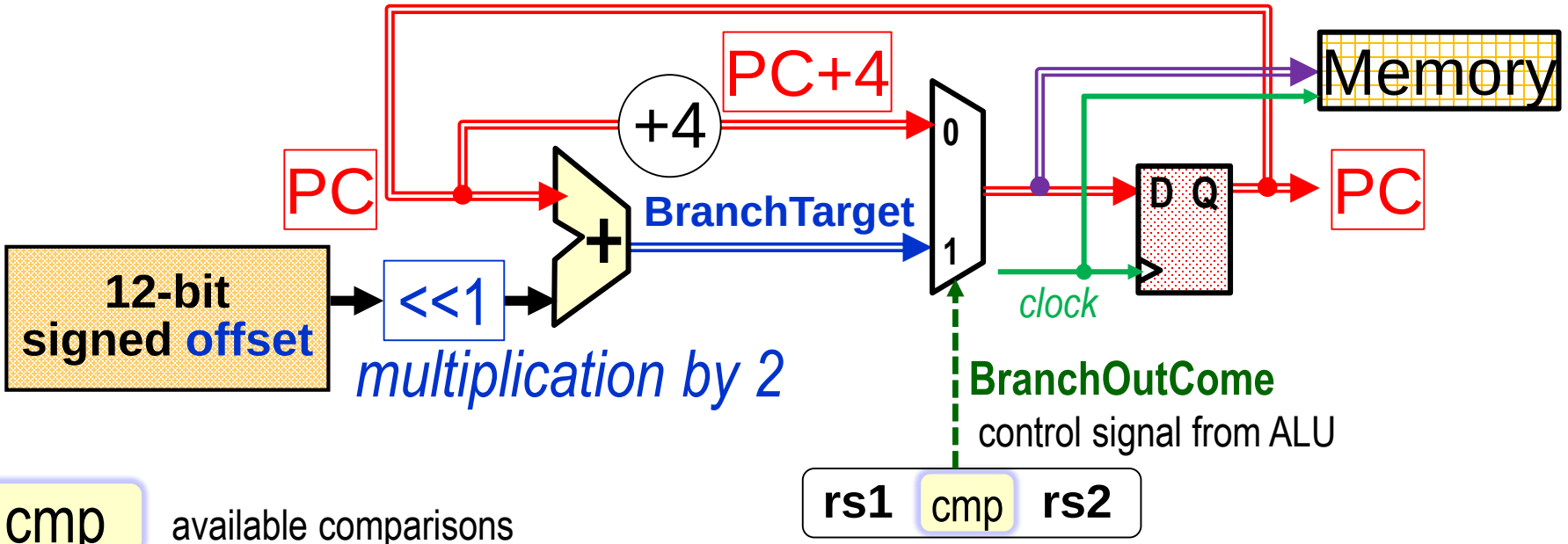


RISC V machine code **NOP** =0x13, code 0x00 is unknown → exception, exception unknown instruction

RICS V - PC Basic Cycle (Program Counter)



Conditional jumps in a simplified diagram



cmp available comparisons

== **BEQ** rs1, rs2, **offset**

!= **BNE** rs1, rs2, **offset**

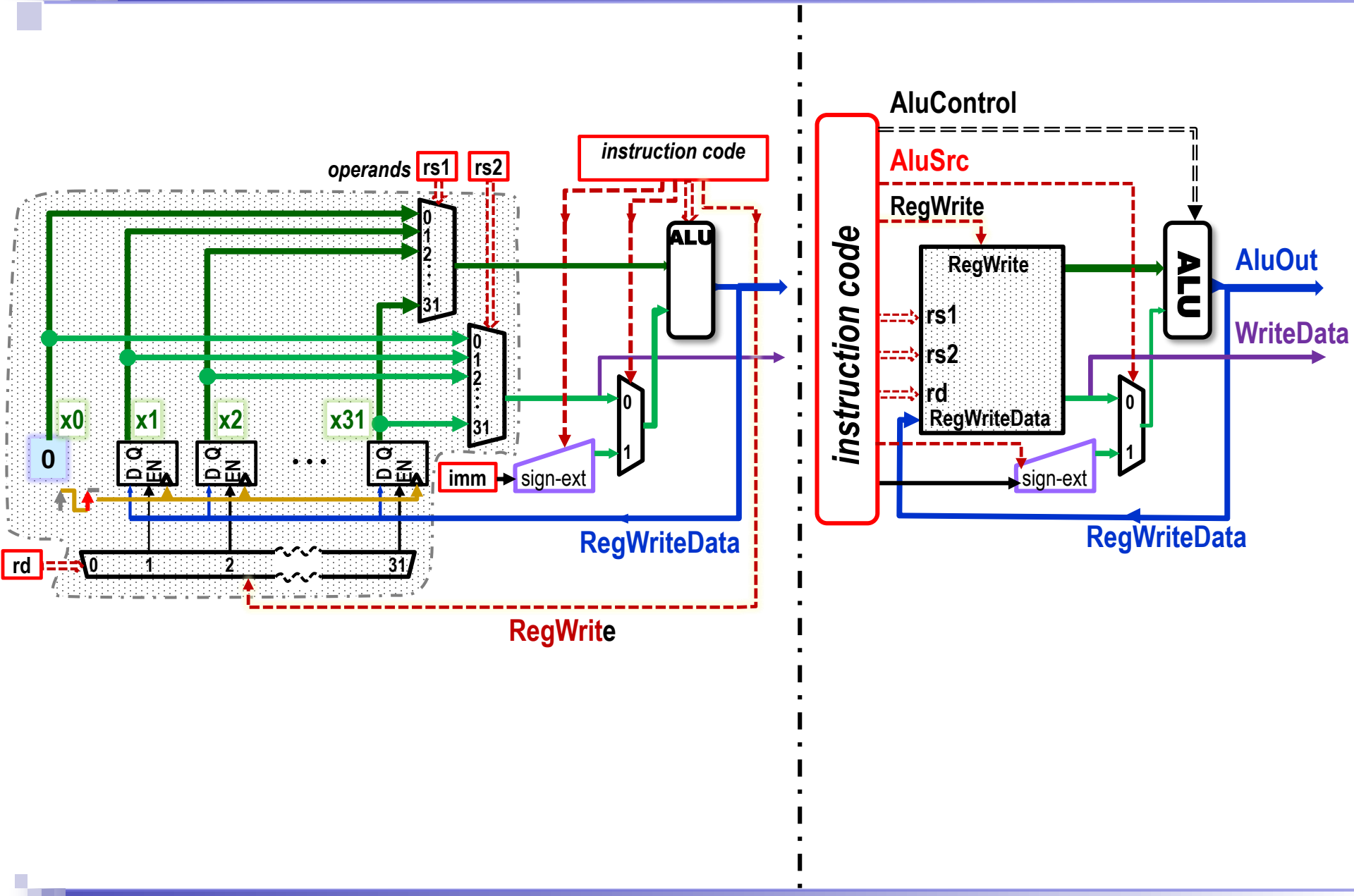
< **BLT** *signed*(rs1), *signed*(rs2), offset

< **BLTU** *unsigned*(rs1), *unsigned*(rs2), offset

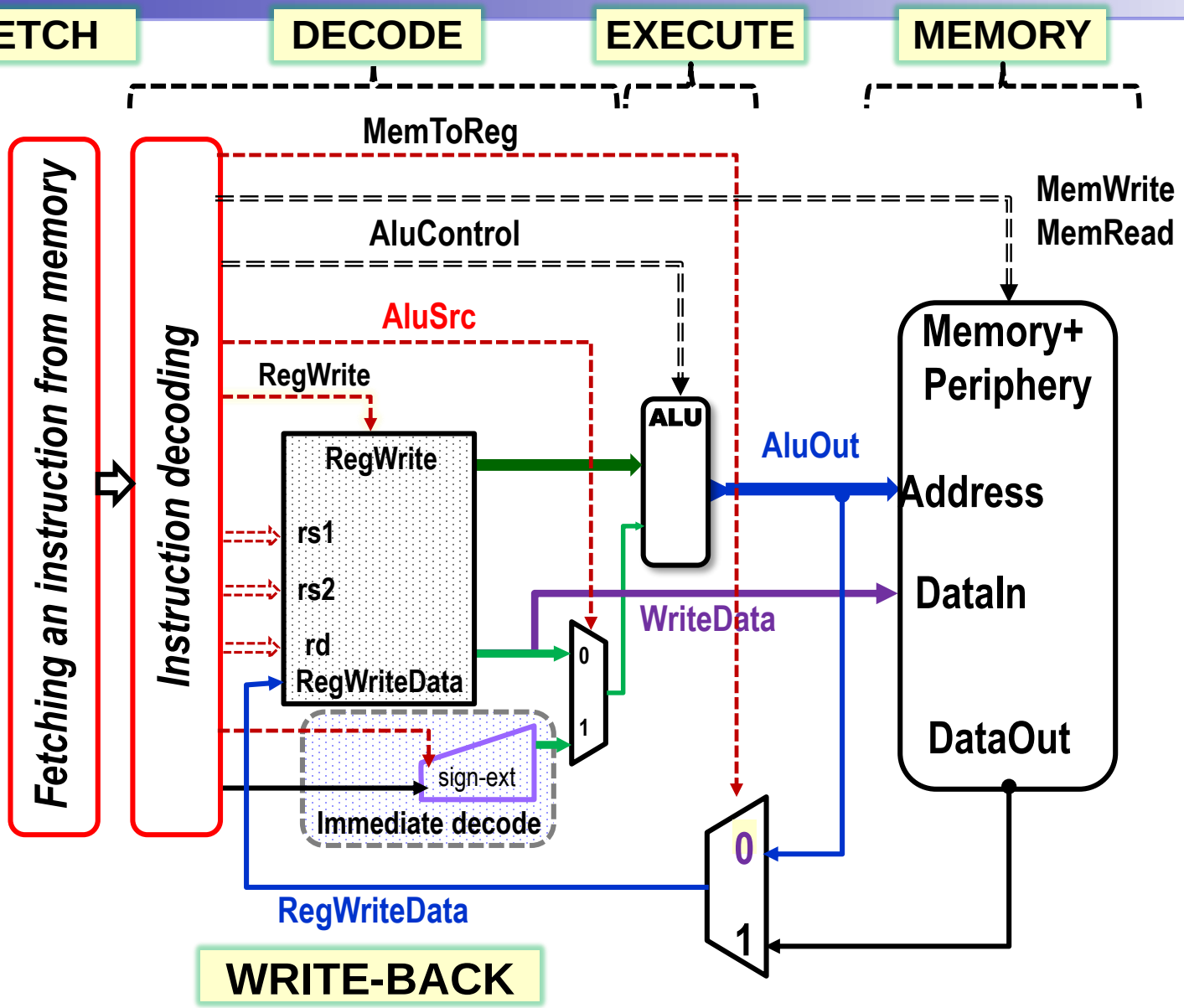
>= **BGE** *signed*(rs1), *signed*(rs2), offset

>= **BGEU** *unsigned*(rs1), *unsigned*(rs2), offset

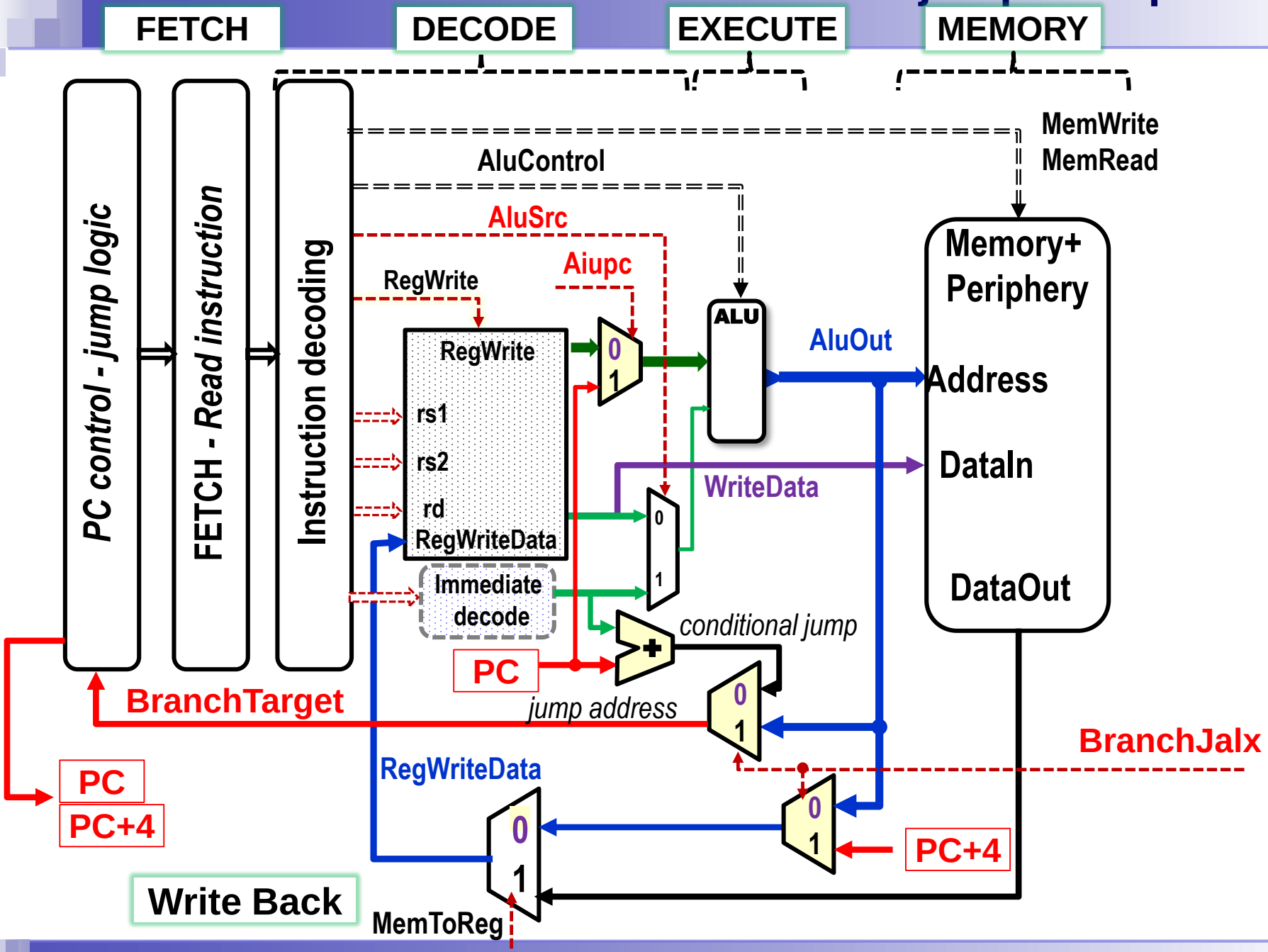
Download the register array to the block



Now we distinguish 5 phases of instruction processing

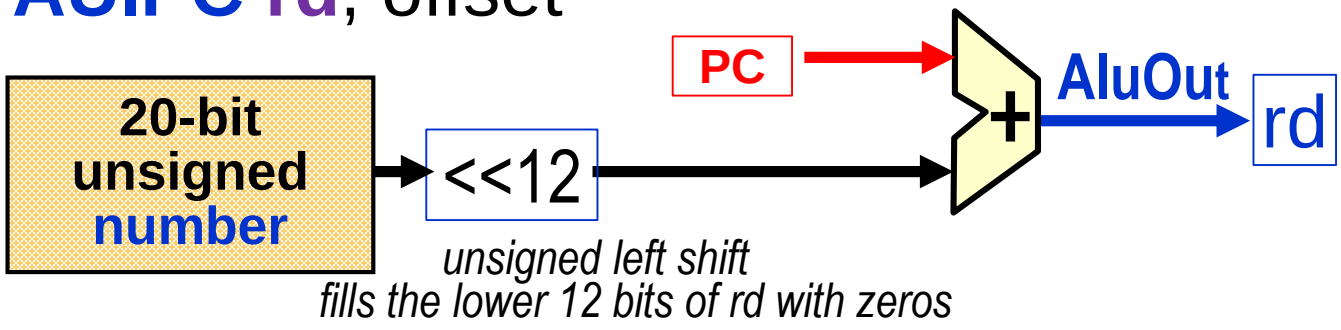


Add all jump multiplexers



Loading the label address

AUIPC rd, offset



x4

	Address	Instruction
0x220	0x00000220	auipc x4, 0x0
0x400	0x00000224	addi x4, x4, 480
	0x00000228	sw x3, 0(x4)

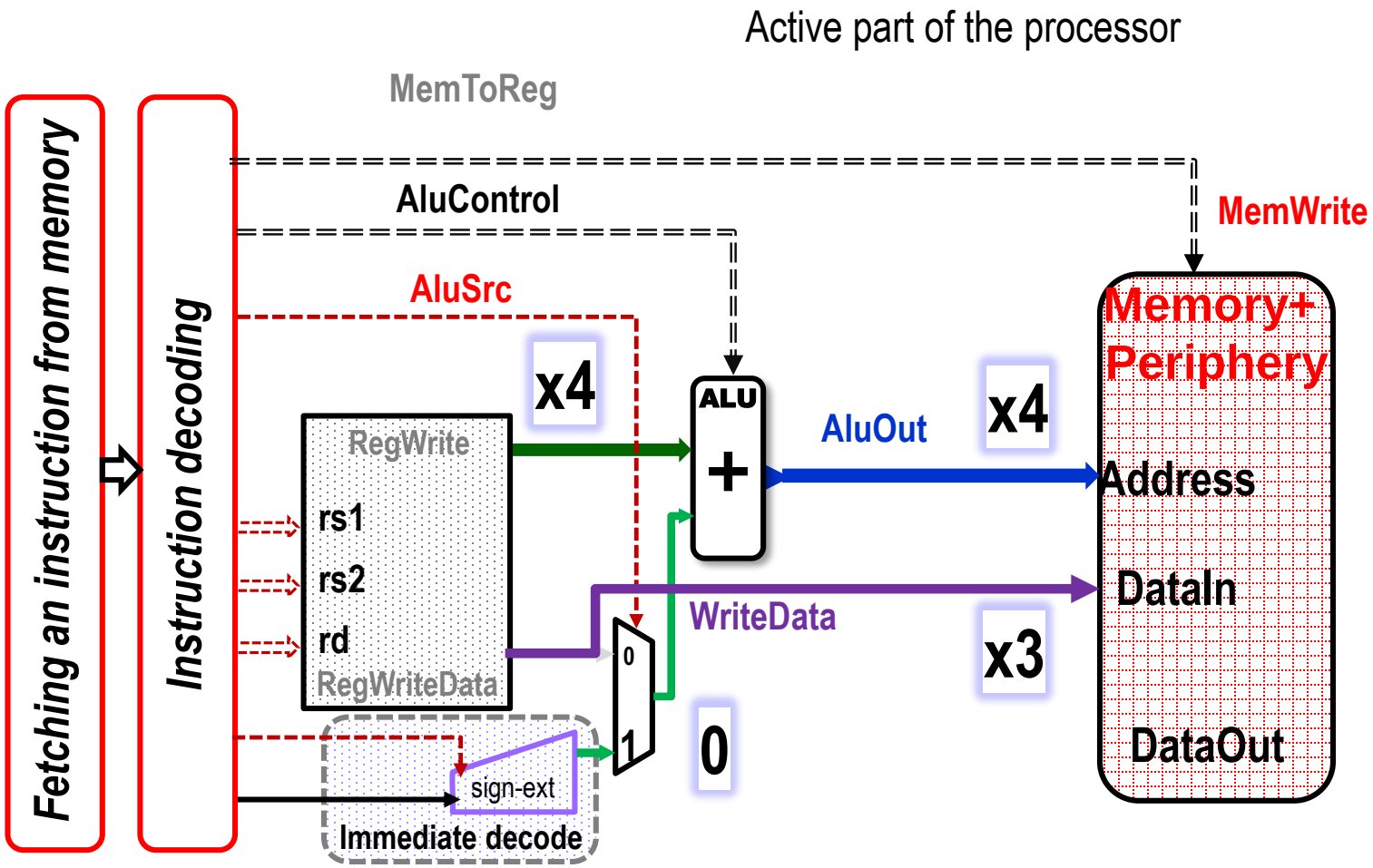
480=0x1E0

Load Address macroinstructions

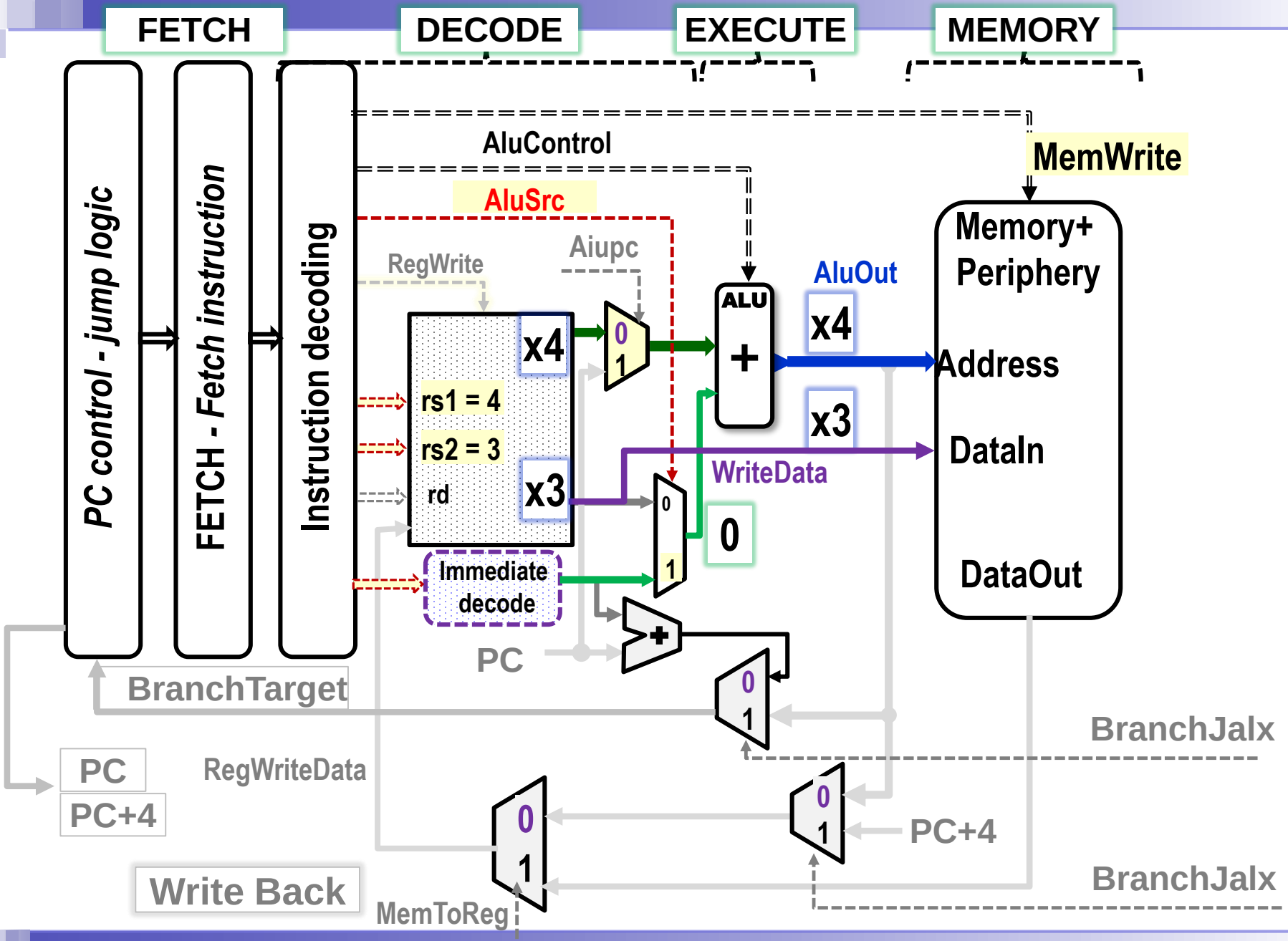
```
la x4, fib13 // x4=&fib10

sw x3, 0(x4) // *fib13=x3

.data // data segment
.org 0x400 // start address
fib13: .word 0 //32bit 0
```

Store Word

Simplified: sw x3, 0(x4) ≡ mem[x4]←x3



Total: sw x3, 0(x4) ≡ mem[x4]←x3

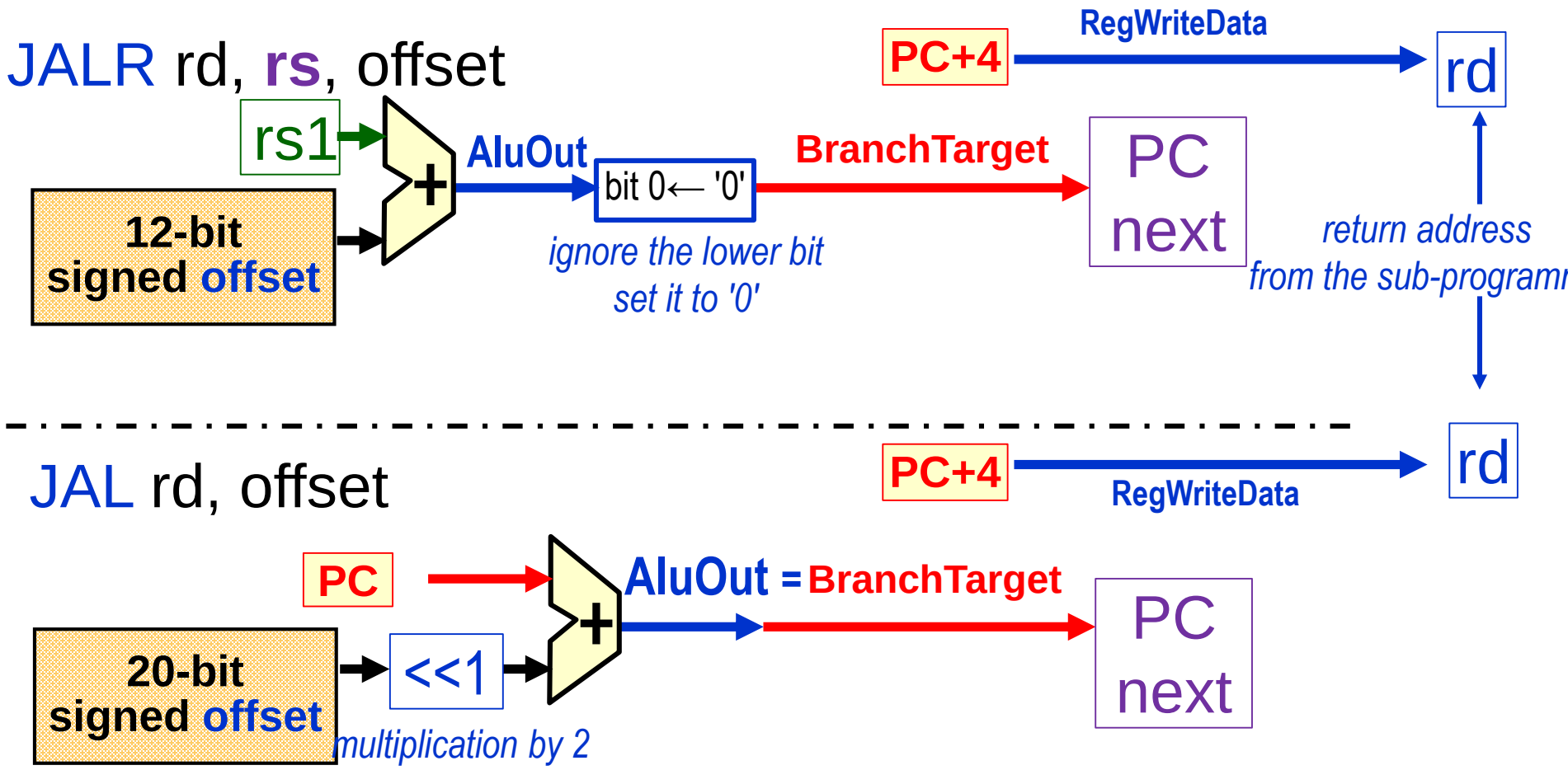


Complete rabbit assembler model AD 1202

Attached as a file
fib_rabbits.s

```
// Simulator directives to make interresting windows visible
#pragma qtmips show registers
.globl _start // entry point
.option norelax // no optimization of codes or instruction orders
.text // begin block of instructions
_start: ori x1, zero, 1 // x1←0 or 1
        ori x2, zero, 1 // x2←0 or 1
        ori x4, zero, 10 // x4←0 or 10
loop:
        add x3, x1, x2 // x3←x1 + x2
        or x1, x2, zero // x1←x2 or 0 ≡ x1 ← x2
        or x2, x3, zero // x2←x3 or 0 ≡ x2 ← x3
        addi x4, x4, -1 // x4←x3 + -1 ≡ x4- -
        blt zero, x4, loop // if(0<x4) goto loop;
        la x4, fib13 // x4=&fib10
        sw x3, 0(x4) // *fib13=x3
stop: ebreak // break point
        nop
.data // begin block of data
.org 0x400 // start address of data segment
fib13: .word 0 // 32bit word initilized to 0
```

Unconditional jumps



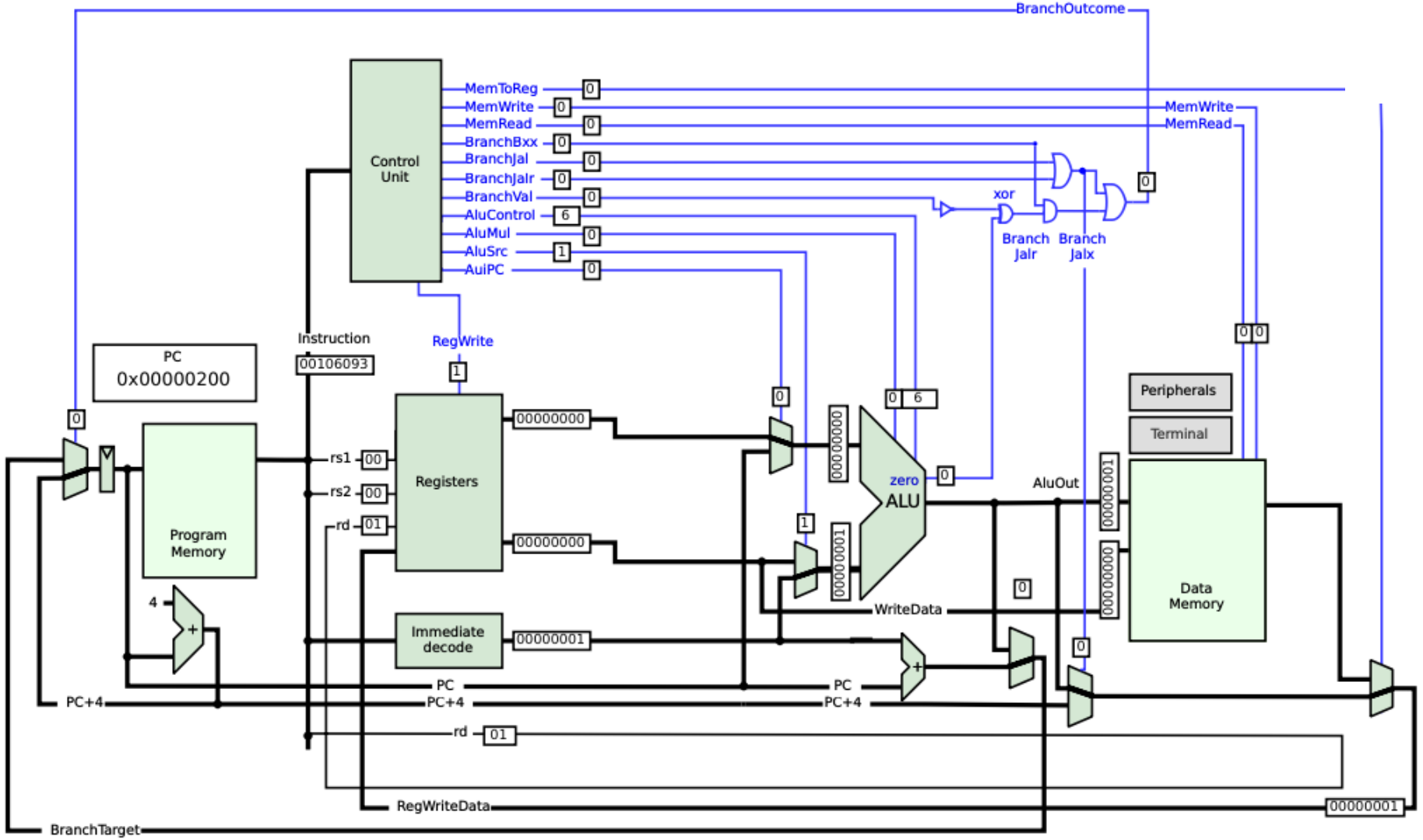
Unconditional jumps

- All unconditional jump instructions use PC-relative addressing that supports position-independent code. The JALR instruction is defined to allow a jump anywhere in the 32-bit absolute address range by applying no more than two instructions.
- The LUI instruction can read the top 20 bits of the destination address into rs1 and then JALR adds the lower bits.
- Similarly, the AUIPC and JALR pair allow jumping anywhere in the 32-bit relative address range.
- Unlike the conditional branch instruction, **the JALR instruction does not treat 12-bit offsets as multiples of 2 bytes** to simplify processor hardware.
- In practice, most uses of JALR will either have sr1=x0, i.e. zero, or be paired with LUI or AUIPC, so the slight reduction in range is not significant.
- The JALR instruction ignores the lowest bit of the computed target address, allowing auxiliary information to be stored in the lowest bit of the function pointers. Although there is a slight loss of control of incorrect addresses in this case, in practice jumps to the wrong instruction address will usually raise an exception.
- When using rs1=x0, the JALR can implement a subroutine call with a single instruction ranging from 2 kB before to 2 kB after the immediate address of the instruction currently being processed, which will facilitate quick use of a small subroutine library.
- **The JAL and JALR instructions will generate an instruction misalignment exception if the destination address is not aligned to a four-byte boundary.**

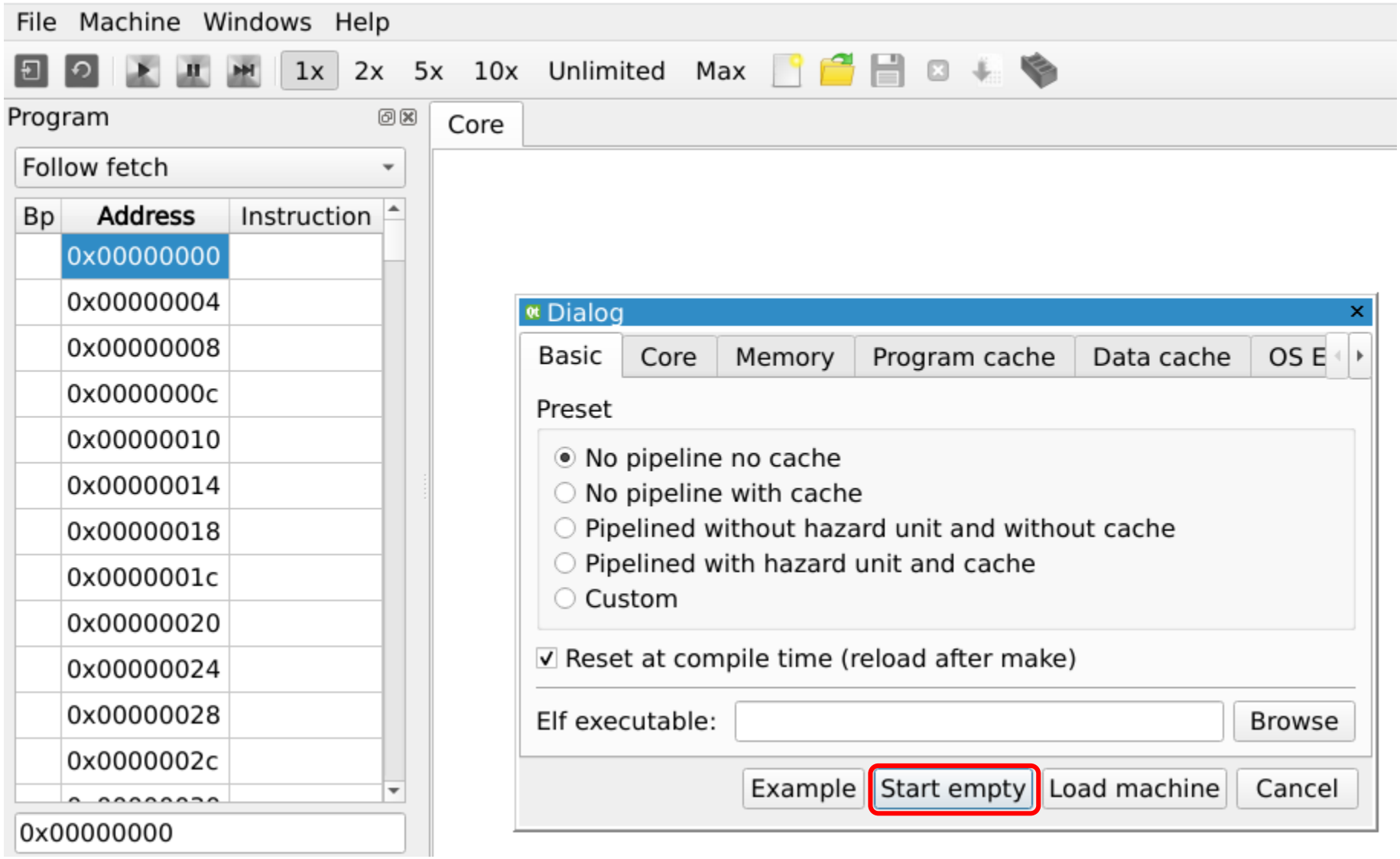
AndrewWaterman, Krste Asanovic:
The RISC-V Instruction Set Manual, Volume I: User-Level ISA Document Version 2.2, page 16

Complete RISC V 32I in simulator

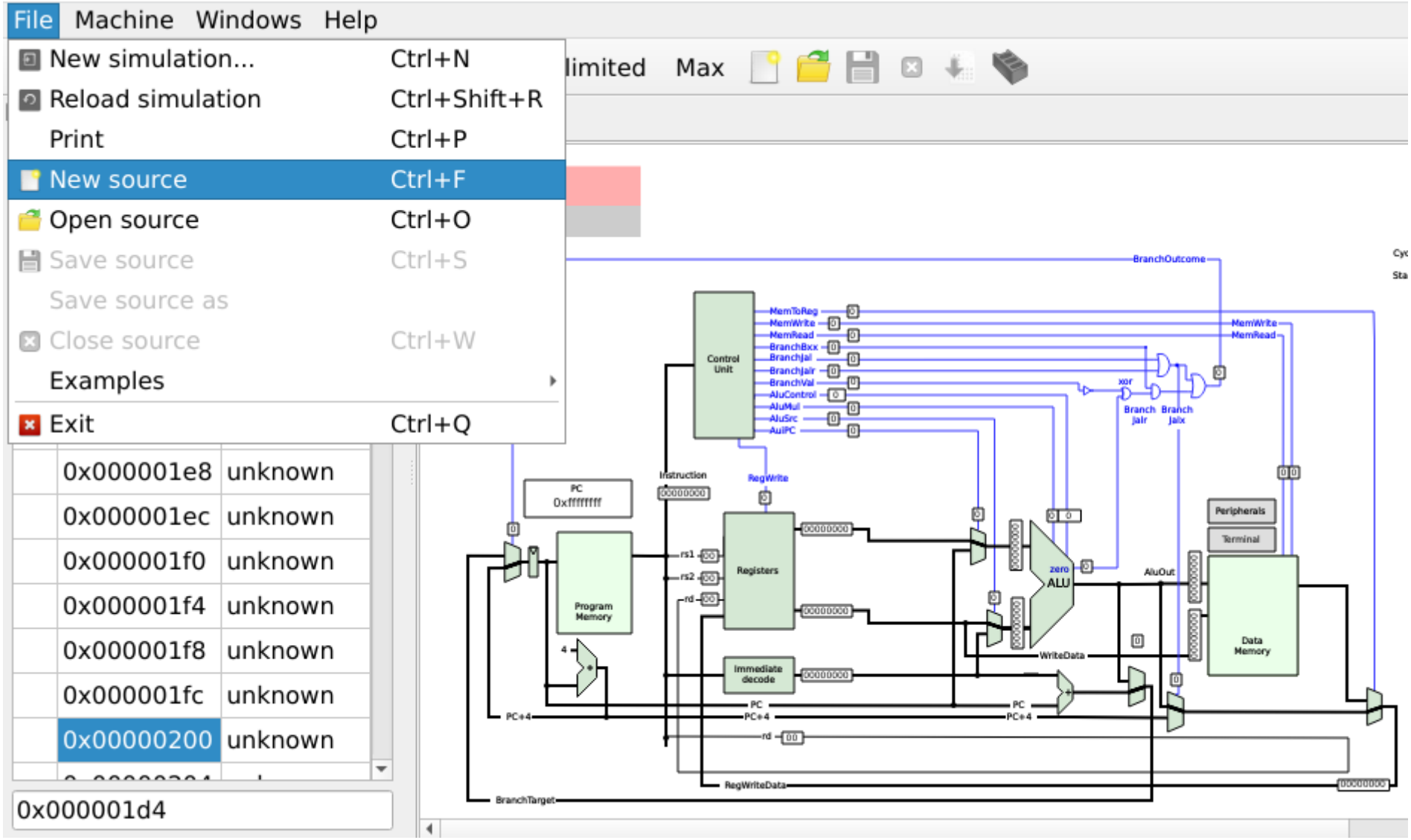
On-line RISC V: https://comparch.edu.cvut.cz/qtrvsim/app/
or offline download at https://comparch.edu.cvut.cz/ ,
For Windows, choose build : qtrvsim-mingw32-v0.9.5.zip



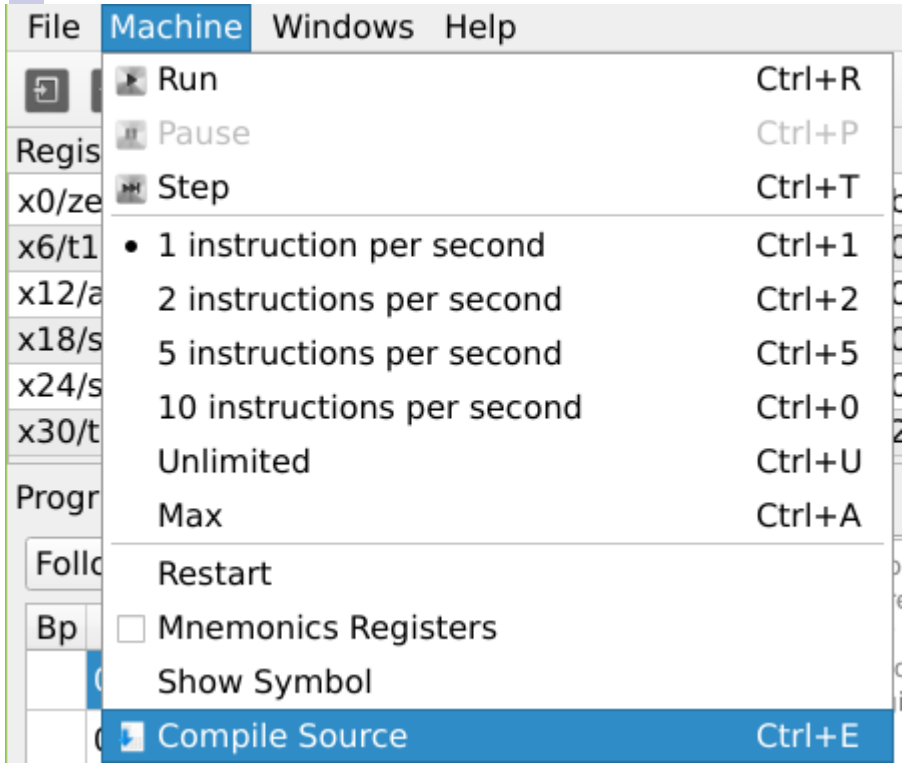
RISC V simulator



We create a file



Simulator Editor and Translation




```
// Simulator directives to make interresting windows visible
#pragma qtmips show registers
.globl _start       // entry point
.option norelax     // no optimalization of codes or instruction orders
.text               // begin block of instructions
_start:   ori  x1, zero, 1      // x1?0 or 1
          ori  x2, zero, 1      // x2?0 or 1
          ori  x4, zero, 10     // x4?0 or 10
loop:
          add  x3, x1, x2       // x3?x1 + x2
          or  x1, x2, zero      // x1?x2 or 0 =  x1 ? x2
          or  x2, x3, zero      // x2?x3 or 0  =  x2 ? x3
          addi  x4, x4, -1      // x4?x3 + -1 = x4--
          blt   zero, x4, loop  // if(0<x4) goto loop;
          la    x4, fib13       // x4=&fib10
          sw   x3, 0(x4)        // *fib13=x3
stop:     ebreak                // break point
          nop
.data         // begin data segment
.org 0x400    // start address of data segment
fib13:        .word  0   // 32bit word initilized to 0
```

If you are using the online version, it is recommended to copy it into another text editor and save it there. The web application cannot overwrite the file, it still saves it as a new download. The offline version saves by default.

For Copy Paste

```
// Simulator directives to make interesting windows visible
#pragma qtmips show registers
.globl _start    // entry point
.option norelax  // no optimization of codes or instruction reorders
.text            // begin block of instructions
_start:  ori  x1, zero, 1     // x1=0 or 1  - first
         ori  x2, zero, 1     // x2=0 or 1 - second
         ori  x4, zero, 10    // x4=0 or 10 - limit
loop:    add  x3, x1, x2      // x3=x1 + x2 - next
         or   x1, x2, zero    // x1=x2 or 0;  x1 = x2
         or   x2, x3, zero    // x2=x3 or 0;  x2 = x3
         addi x4, x4, -1      // x4=x3 + -1 = x4--
         blt  zero, x4, loop  // if(0<x4) goto loop;
         la   x4, fib13       // x4=&fib10
         sw   x3, 0(x4)       // *fib13=x3stop:
         ebreak               // break point
         nop
.data            // begin data segment
.org 0x400       // start address of data segment
fib13:  .word  0   // 32bit word initialized to 0
```

Run

Run

1 step

simulation speeds

File Machine Windows Help

1x 2x 5x 10x Unlimited Max

Registers

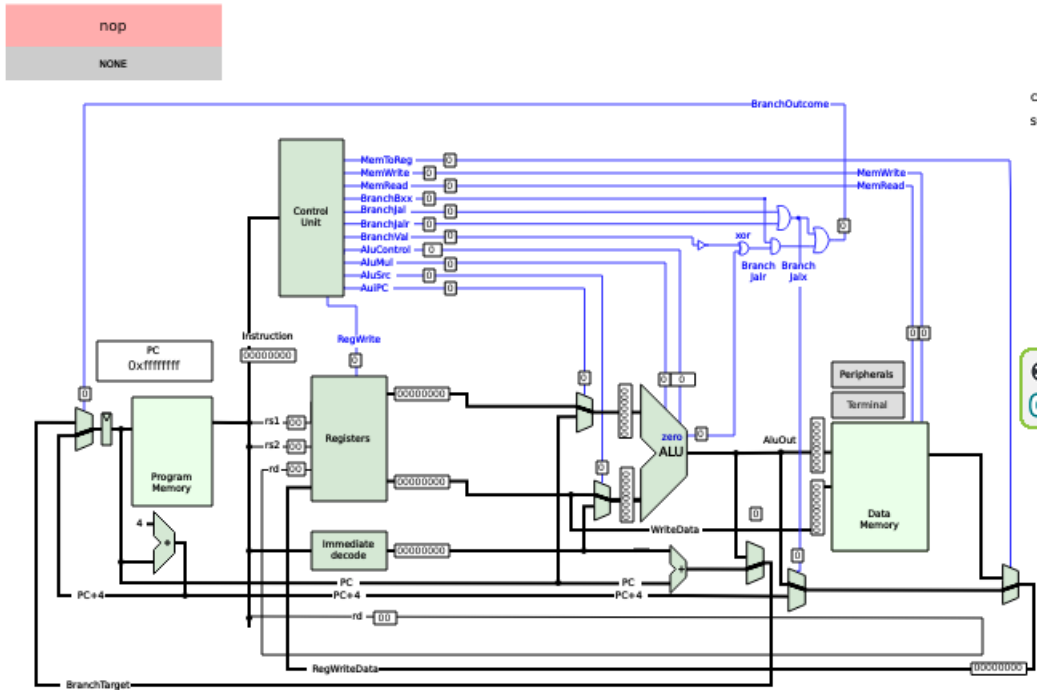
x0/zero	0x0	x1/ra	0x0	x2/sp	0xbfffff00	x3/gp	0x0	x4/tp	0x0	x5/t0	0x0
x6/t1	0x0	x7/t2	0x0	x8/s0	0x0	x9/s1	0x0	x10/a0	0x0	x11/a1	0x0
x12/a2	0x0	x13/a3	0x0	x14/a4	0x0	x15/a5	0x0	x16/a6	0x0	x17/a7	0x0
x18/s2	0x0	x19/s3	0x0	x20/s4	0x0	x21/s5	0x0	x22/s6	0x0	x23/s7	0x0
x24/s8	0x0	x25/s9	0x0	x26/s10	0x0	x27/s11	0x0	x28/t3	0x0	x29/t4	0x0
x30/t5	0x0	x31/t6	0x0	pc	0x200						

Program

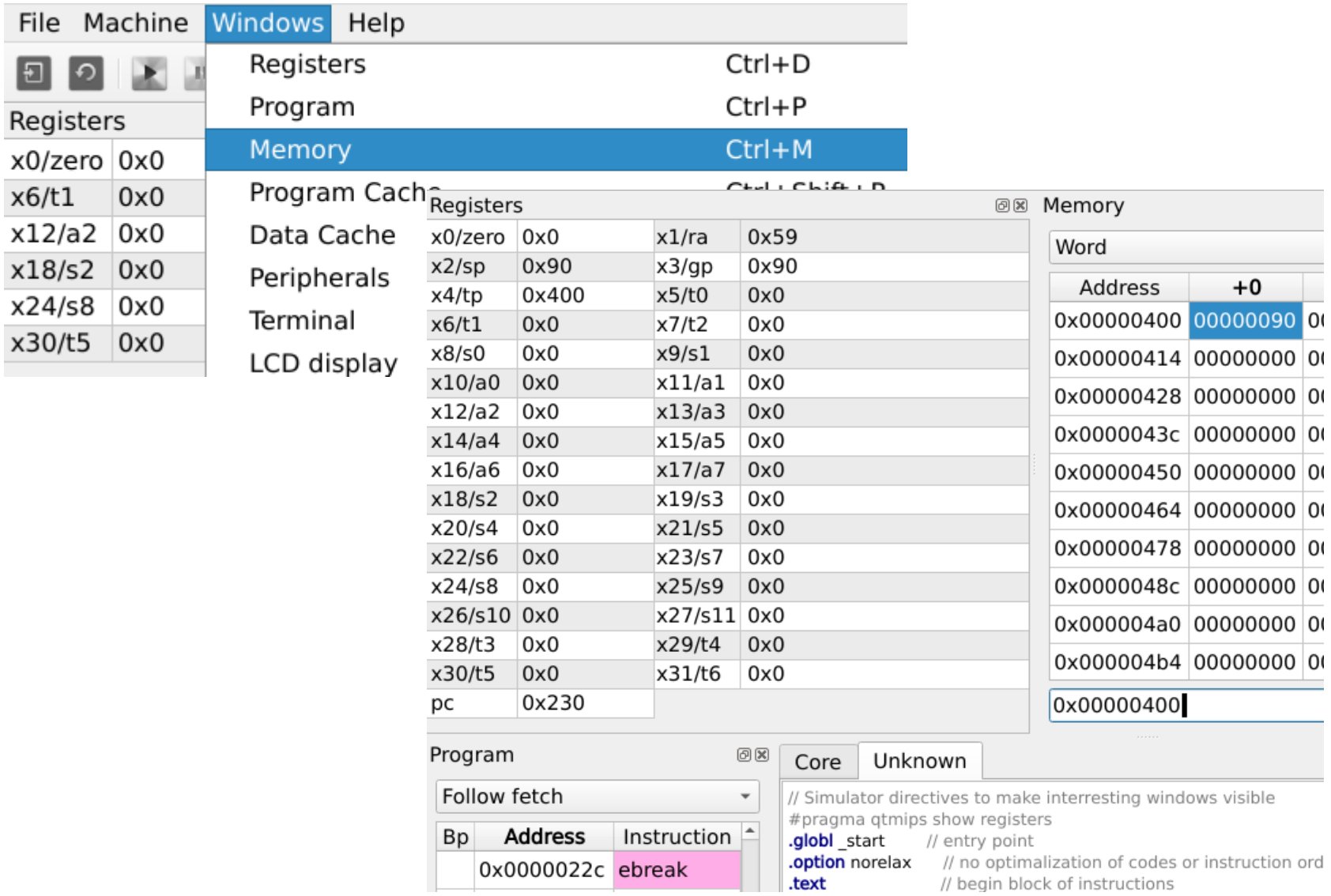
Follow fetch

Bp	Address	Code	Instruction
	0x000001fc	00000000	unknown
	0x00000200	00106093	ori x1, x0, 1
	0x00000204	00106113	ori x2, x0, 1
	0x00000208	00a06213	ori x4, x0, 10
	0x0000020c	002081b3	add x3, x1, x2
	0x00000210	000160b3	or x1, x2, x0
	0x00000214	0001e133	or x2, x3, x0
	0x00000218	fff20213	addi x4, x4, -1
	0x0000021c	fe4048e3	blt x0, x4, 0x20c
	0x00000220	00000217	auipc x4, 0x0
	0x00000224	1e020213	addi x4, x4, 480
	0x00000228	00322023	sw x3, 0(x4)
	0x0000022c	00100073	ebreak

Core Unknown



Memory display



About the RISC V simulator from its authors

•Dupák, J.; Píša, P.; Štepanovský, M.; Kočí, K.
QtRVSim - RISC-V Simulator for Computer Architectures
Classes In: Embedded world Conference 2022.
Haar: WEKA FACHMEDIEN GmbH, 2022. p. 775-778.
ISBN 978-3-645-50194-1.

- Available online https://comparch.edu.cvut.cz/publications/ewC2022-Dupak-Pisa-Stepanovsky-QtRvSim.pdf