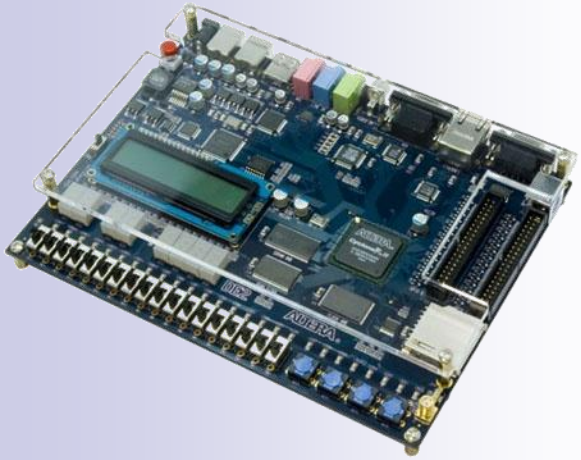
Logic Systems and Processors
cz:Logické systémy a procesory



Lecturer: Richard Šusta

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

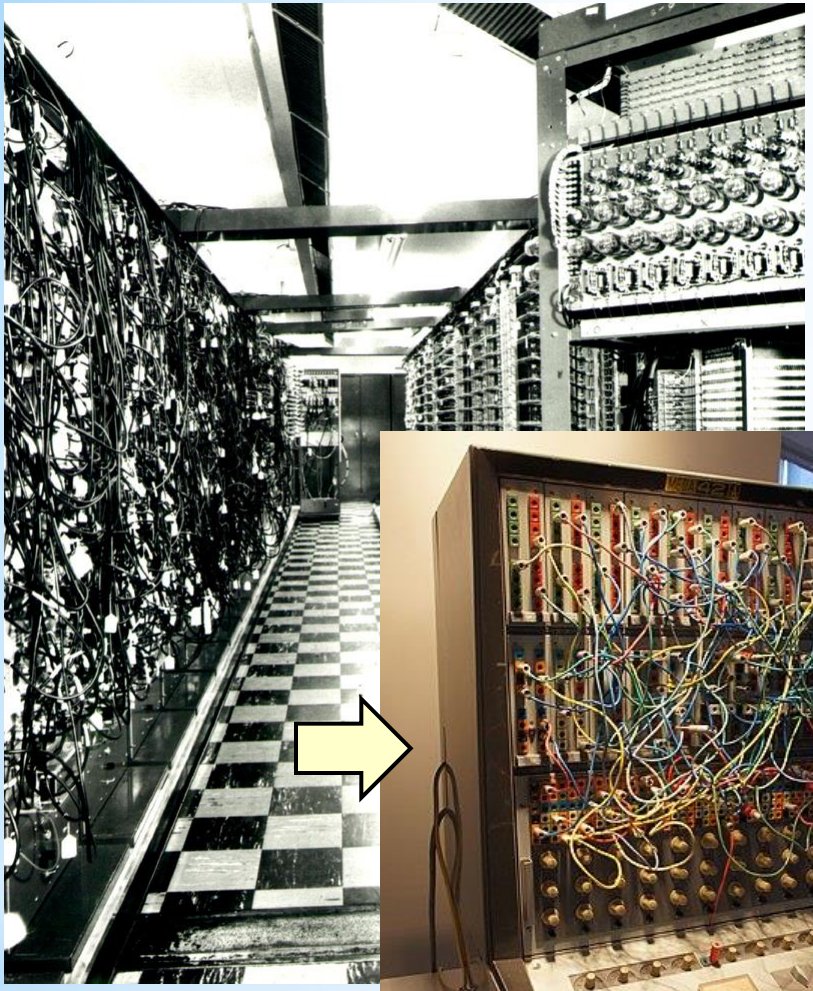
V1.1: Jan 7, 2025

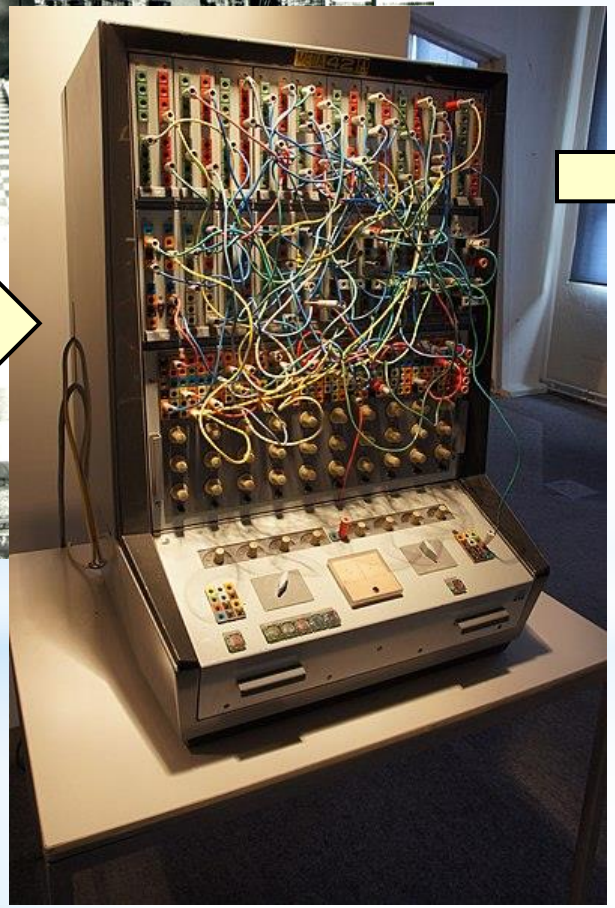
extended slides 45 and 71

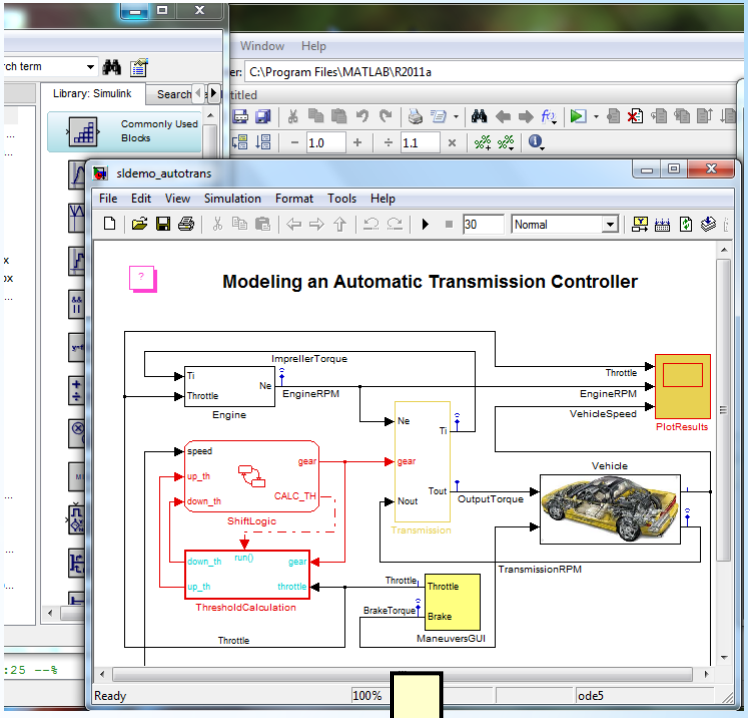
Pipeline ≈ Assembly Line

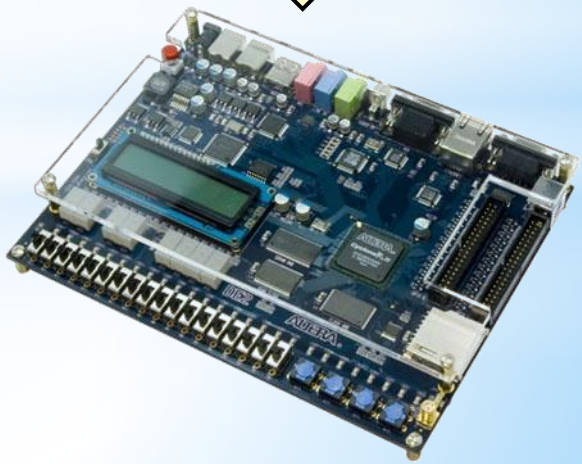
Charlie Chaplin - Factory Scene - Modern Times © Roy Export S.A.S. Scan, courtesy of Cineteca di Bologna

* Parallel Analog Operations



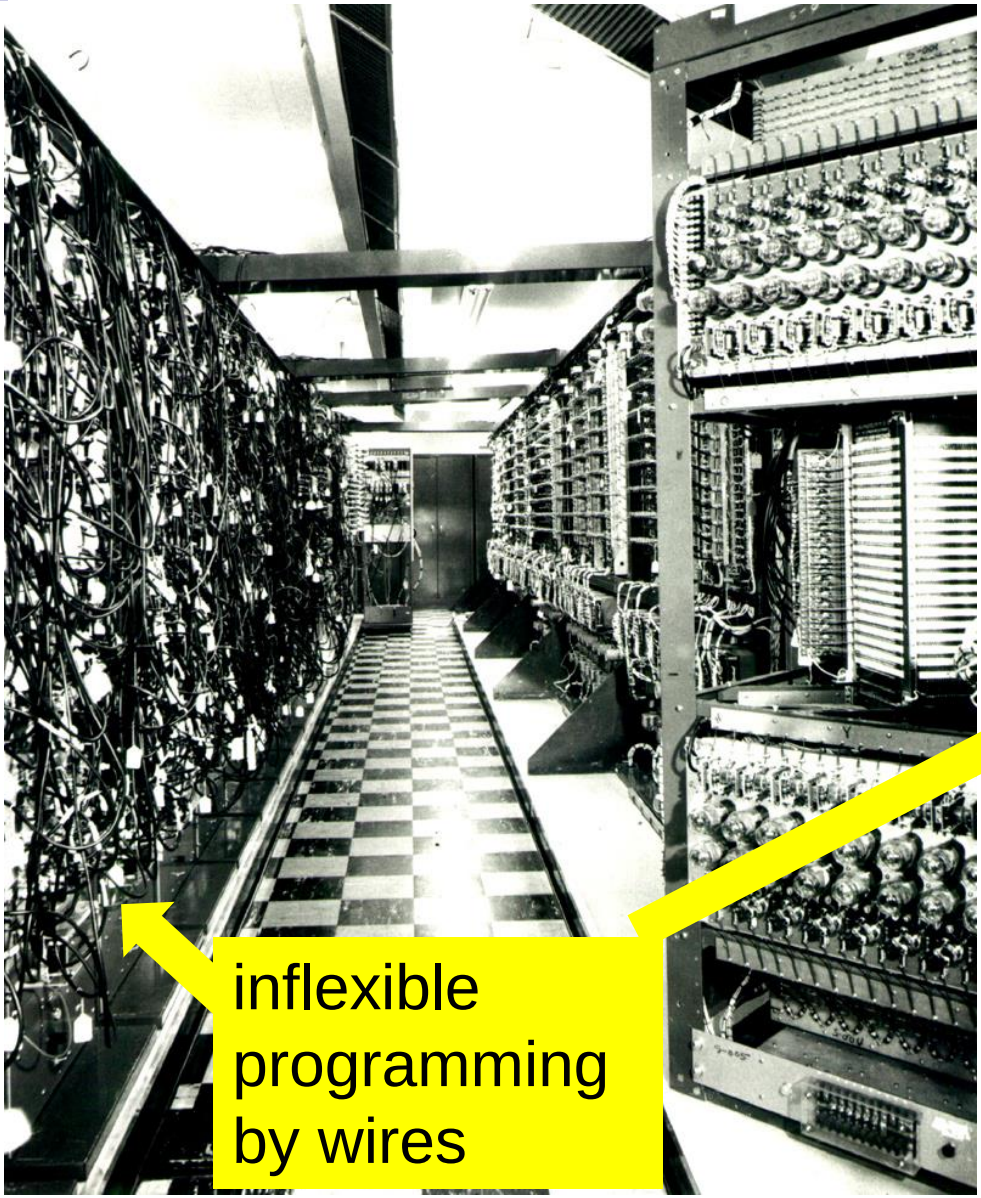






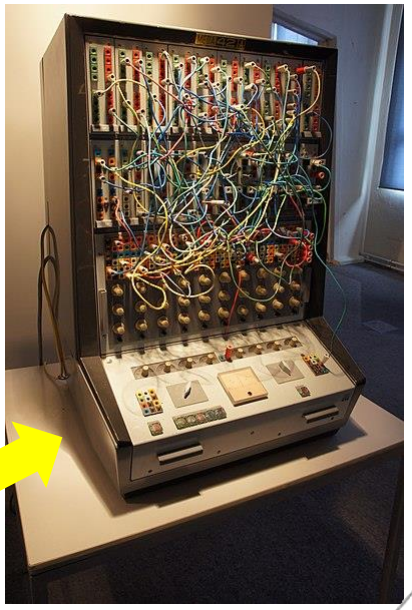
All images courtesy of Wikipedia

1943 MIT's Whirlwind Analog Computer

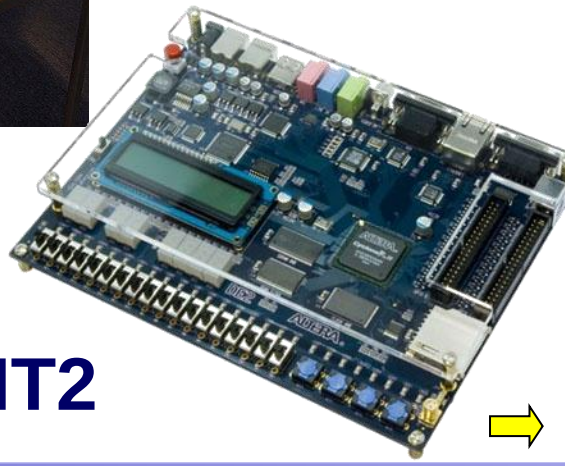


inflexible programming by wires

Designed as a flight simulator for training bomber crews, it simulated a realistic aerodynamic system operating in real time. But it was inaccurate and inflexible.



1980 MEDA Aritma Vysočany 1956-1987



2024 VEEK-MT2

1. Laplace transformation

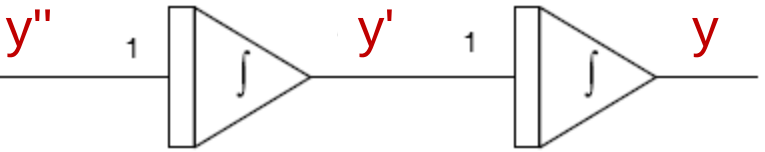
$$F(p)=1/(p^2 + 0.5\ p+1)$$

2. Its differential equations

$$y'' + 0.5\ y' + y = u(t)$$

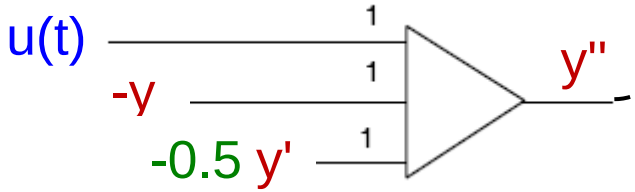
3. Simulink

$$y'' = u(t)-0.5\ y'- y$$

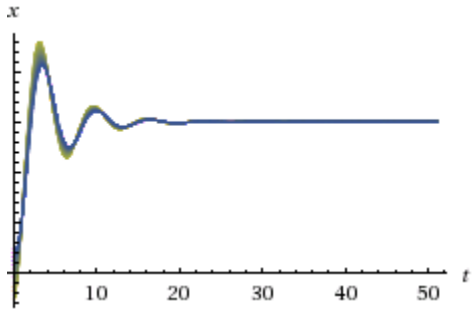


3a. Two integrators

3b. Summation member



3c. Response to unit jump
u(t)=0.3; y(0)=0; y'(0)=1



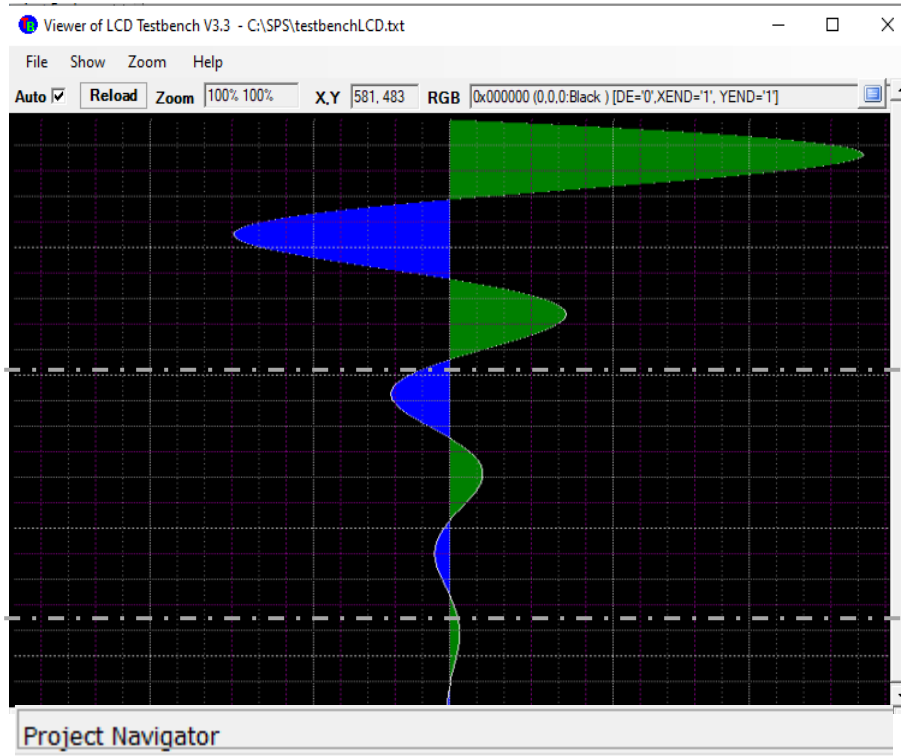
4. Numerical solution for unit jump u(t)=0.3 for t>=0

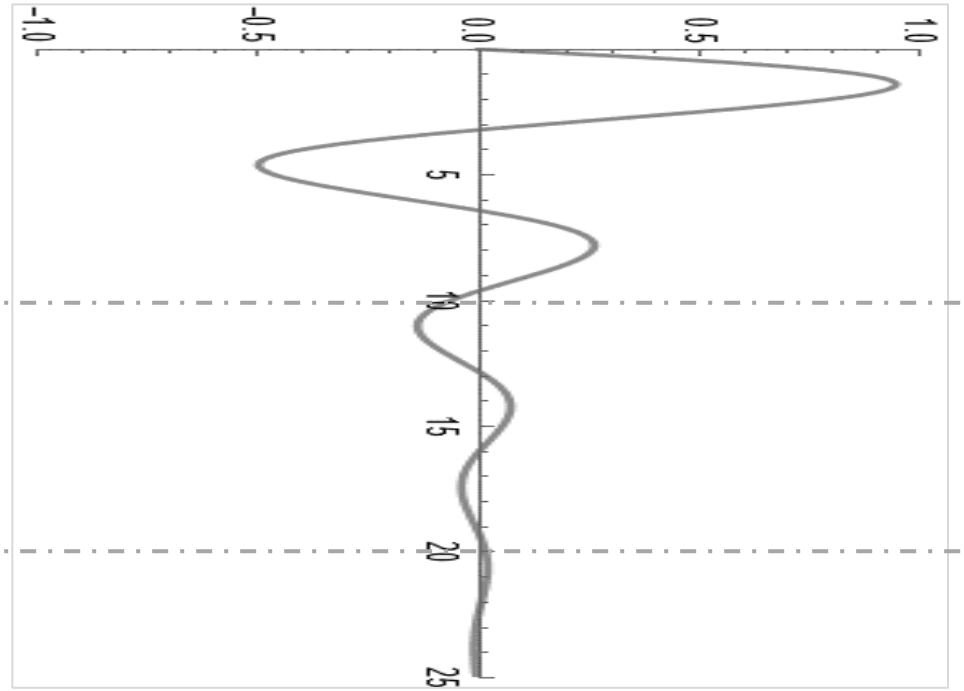
$$y(t)=0.955336\ e^{-0.25t} \sin(0.968246\ t) - 0.3\ e^{-0.25t} \cos(0.968246\ t)+0.3$$

[calculated using WolframAlpha]

Example: the VEEK-MT2

$$y''[t] + 0.4\, y'[t] + y[t] == 0,\ y'[0]==1.25\ ,\ y[0]==0$$





Project Navigator

Entity	Logic Cells	DSP 18x18
Cyclone IV E: EP4CE115F29C7		
Integrator	639 (1)	3
VeekMT2_LCDgenerator:inst	48 (47)	0
VeekMT2_LCDreg:inst1	44 (44)	0
LCDGraph:inst2	562 (168)	3
DifferentialEquation:integrator	400 (400)	2
lpm_mult:Mult0	0 (0)	1
lpm_mult:Mult1	0 (0)	1
lpm_mult:Mult0	0 (0)	1

$$y[t] \to 1.27578\, e^{-0.2\, t} \sin[0.979796\, t]$$

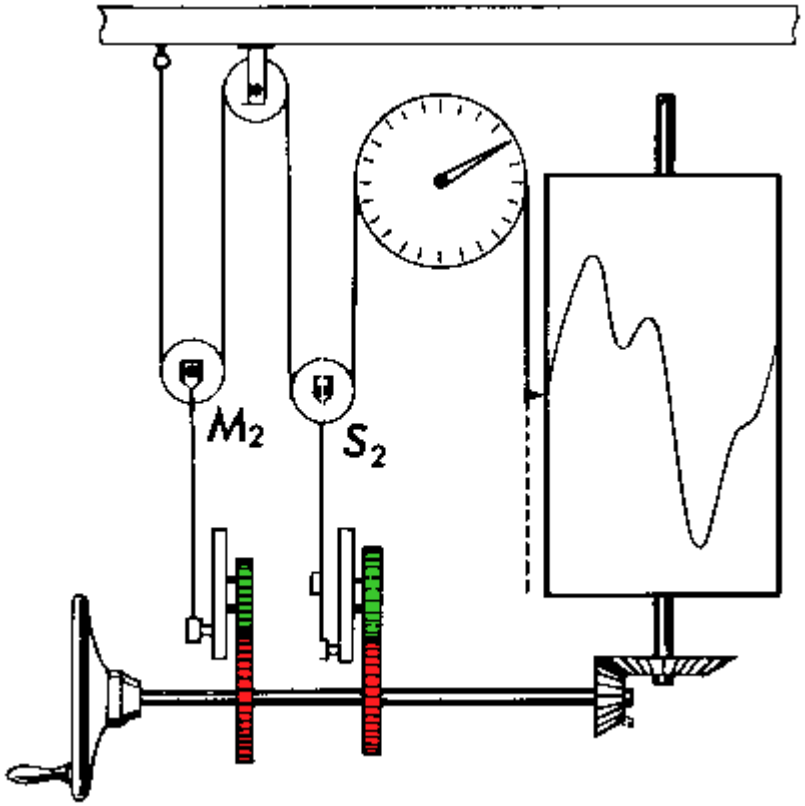
Wolfram Mathematica 14

In an FPGA, all elements of the iteration are emulated in parallel (concurrently), and so the response speed is independent of the number of elements, i.e., the sizes of modeled systems.

The famous mechanical integration solution

[Source: http://www.mit.edu/~klund/analyzer/]

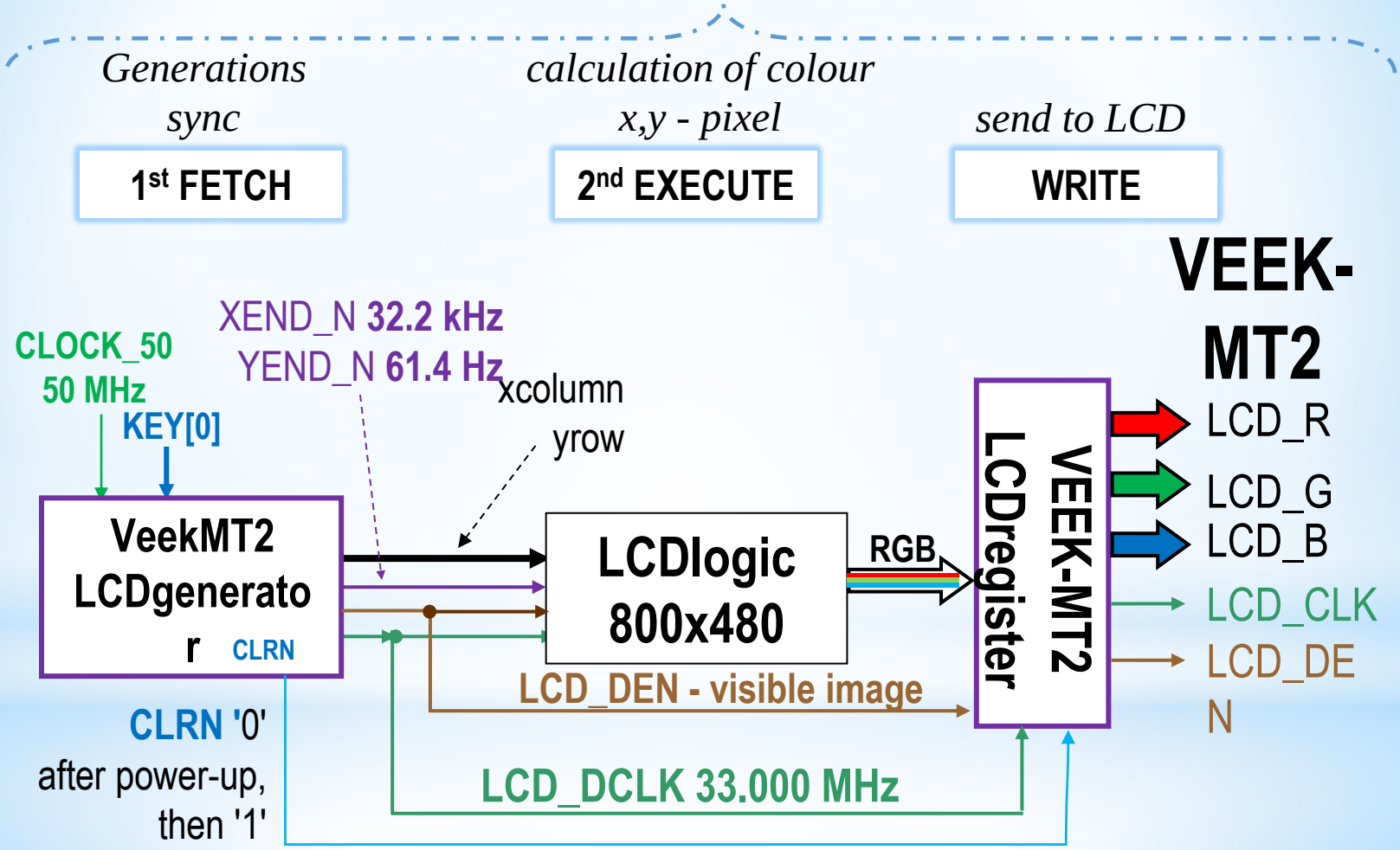
Vannevar Bush
Differential Analyzer

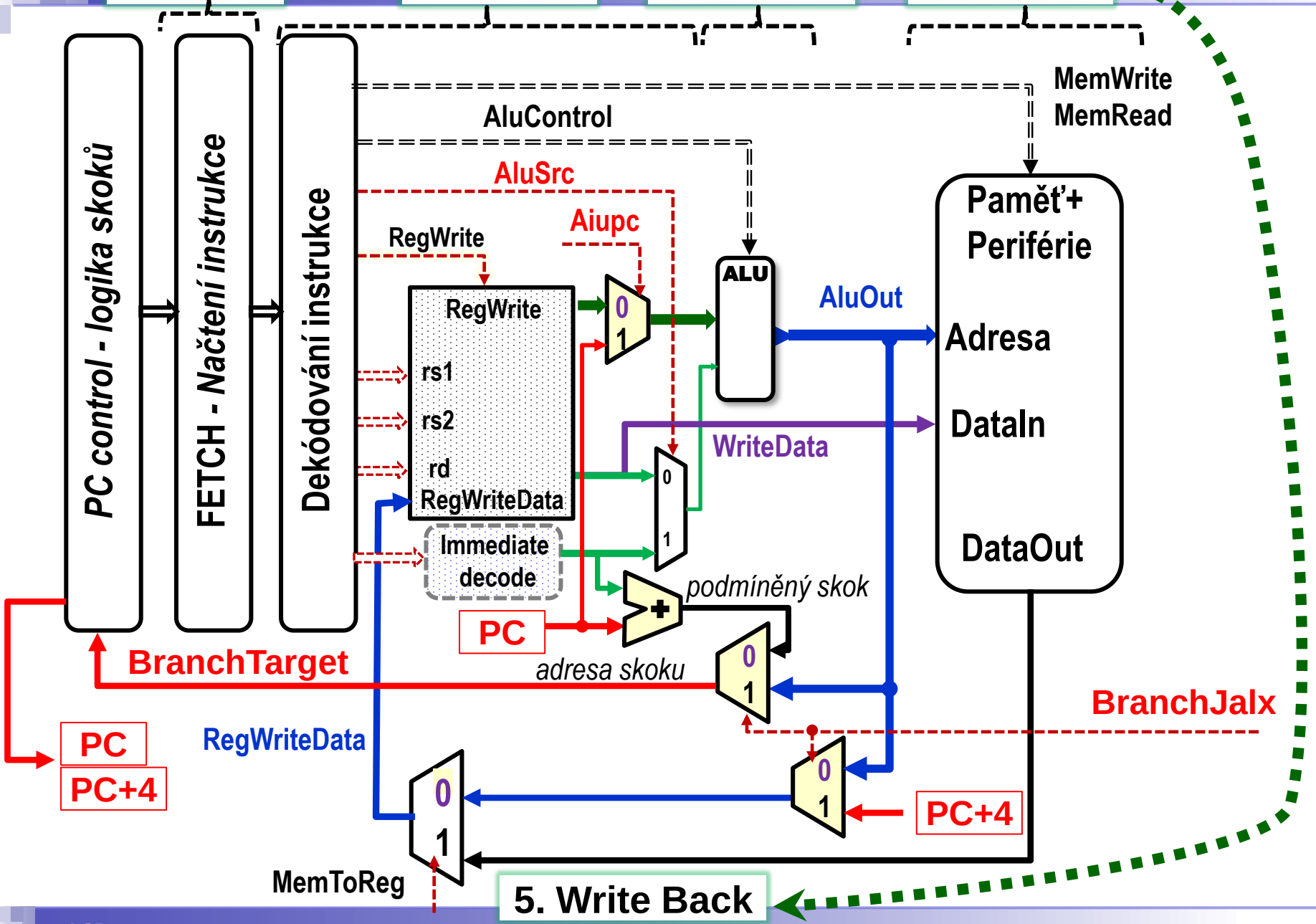


Variable-size friction wheels for simulating the behaviour of differential equations

Video:
http://www.mit.edu/~klund/analyzer/video3.mov

You know Pipeline from LCD it approximates analog processes




1. FETCH
2.DECODE
3. EXECUTE
4. MEMORY
5. Write Back
PC control - logika skoků
FETCH - Načtení instrukce
Dekódování instrukce
AluControl
MemWrite
MemRead
AluSrc
Aiupc
RegWrite
RegWrite
rs1
rs2
rd
RegWriteData
Immediate decode
ALU
AluOut
Paměť+ Periférie
Adresa
DataIn
DataOut
WriteData
podmíněný skok
PC
adresa skoku
BranchTarget
BranchJalx
PC
PC+4
RegWriteData
PC+4
MemToReg
0
1

What the processor does in degrees

- FETCH - fetches an instruction from memory.
- DECODE - the instruction is decoded and the operand values are added to it.
- EXECUTE - ALU (Arithmetic Logic Unit) performs an operation with the values prepared in the DECODE phase.
- MEMORY - possible work with memory, writing or reading the address calculated by ALU, if there is a branch then writing the new address to PC (Program Counter).
- WRITE-BACK - values are saved in registers

Complete rabbit assembler model AD 1202

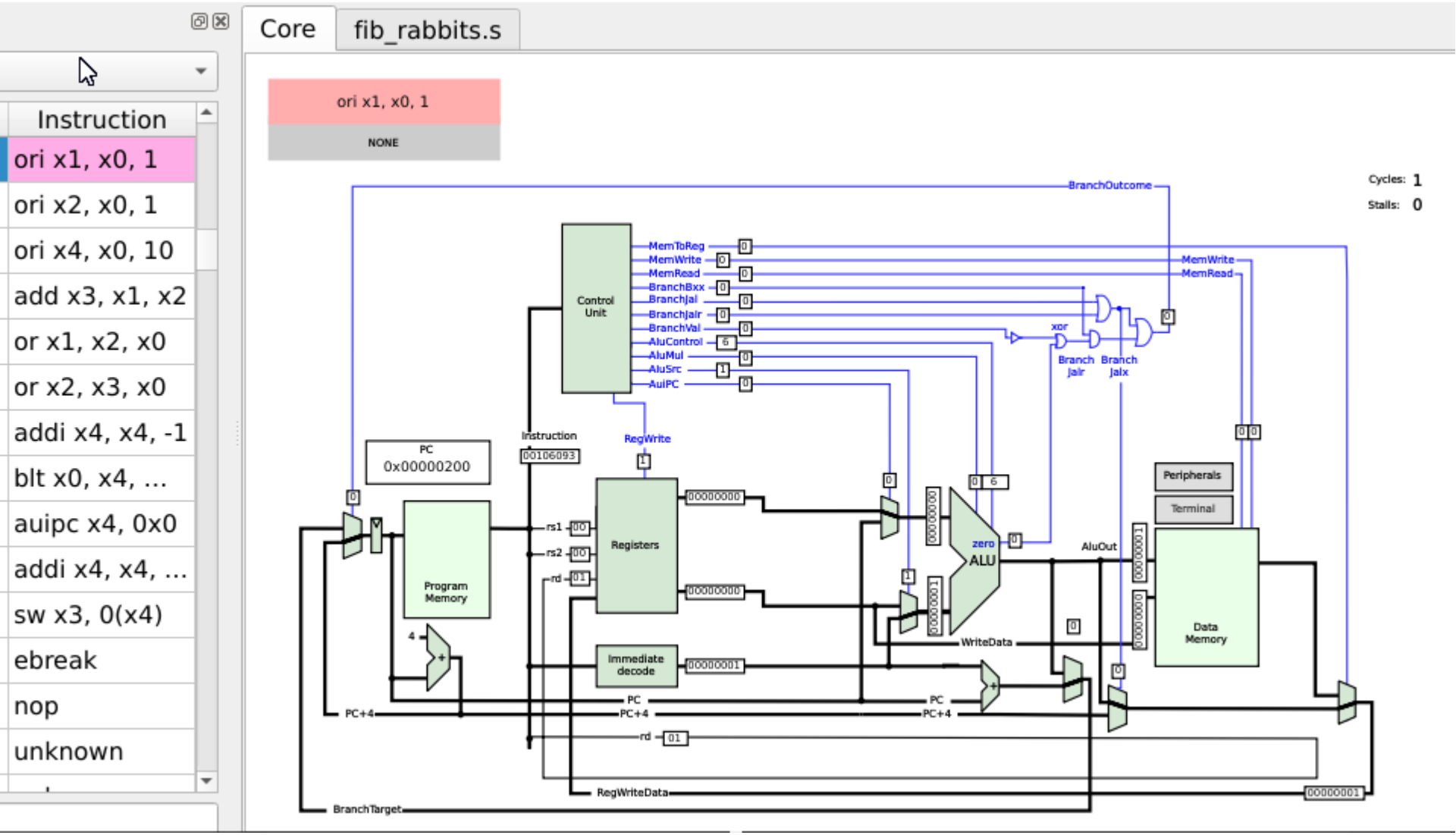
File
fib_rabbits.s

```
// 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
```

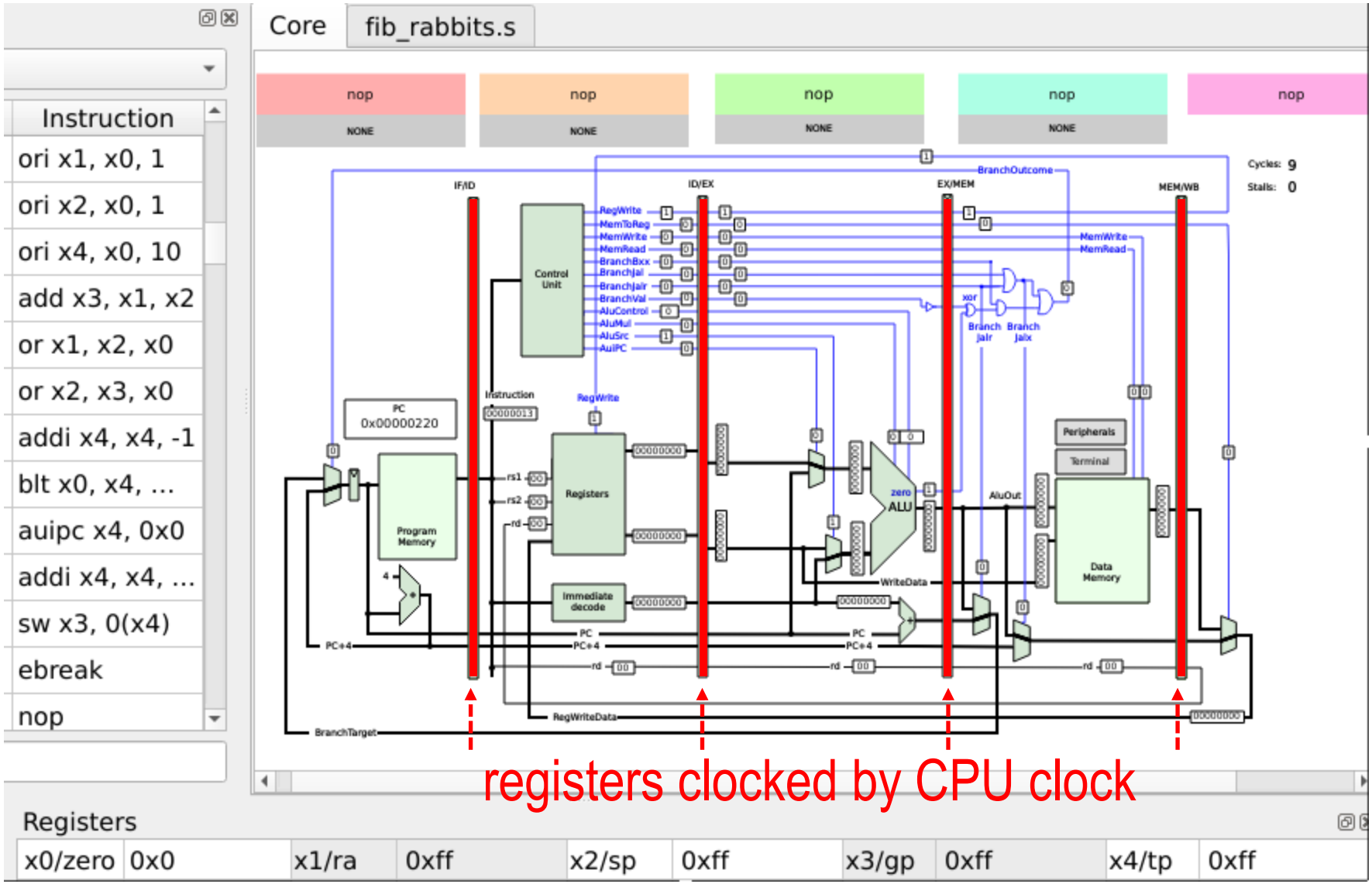
Registry	
zero	always 0
x1	step t-1
x2	step t
x3	step t+1
x4	counter

```
        int x1, x2, x3, x4;
        x1 = 1, x2 = 1; x4 = 10;
Loop:   x3 = x1 + x2;
        x1 = x2; x2 = x3;
        x4--;
        if(0<x4) goto Loop;
        Fib13=x3; //&Fib13=0x400
```

Single cycle RISC V simulator

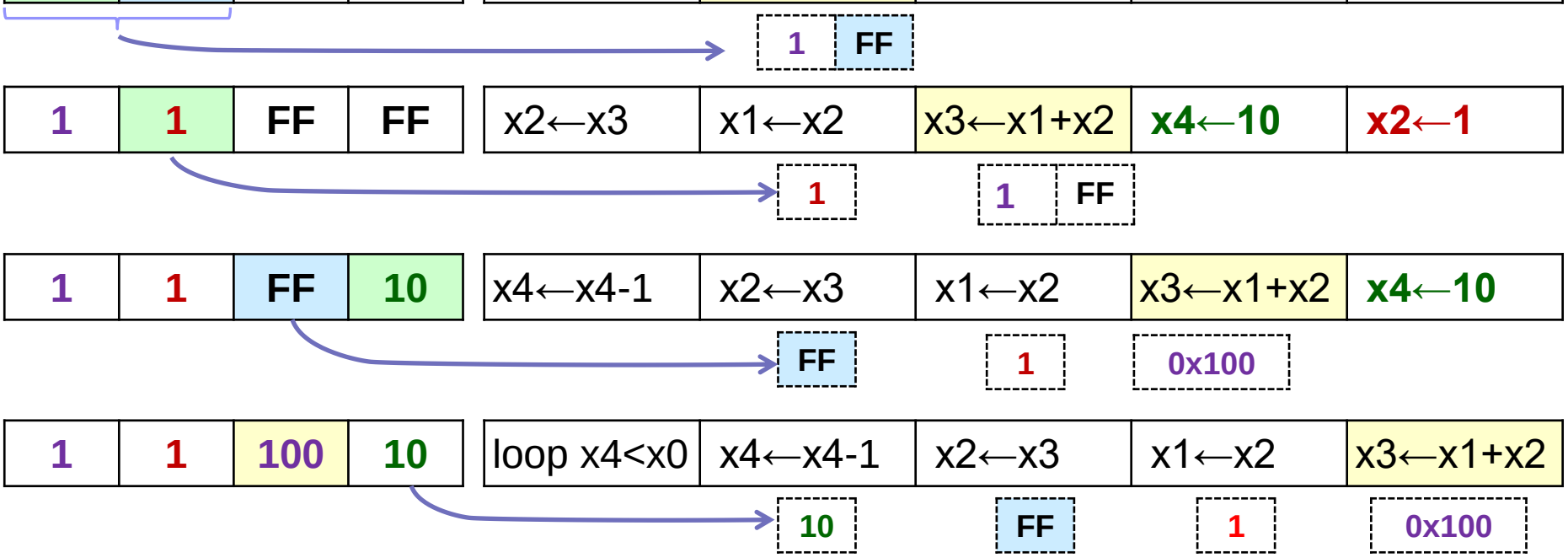


Pipeline Simulator RISC V

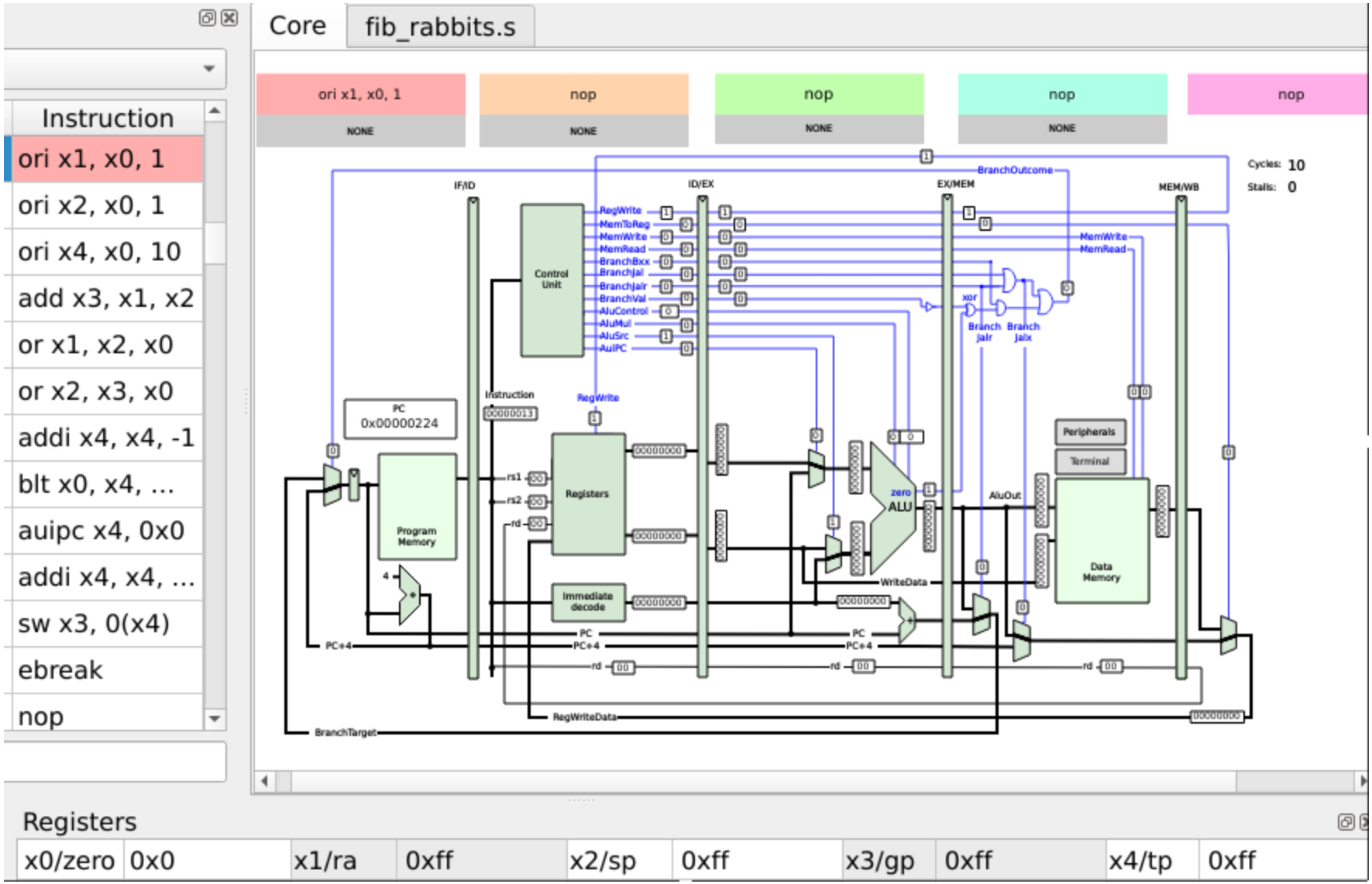


Data Hazards

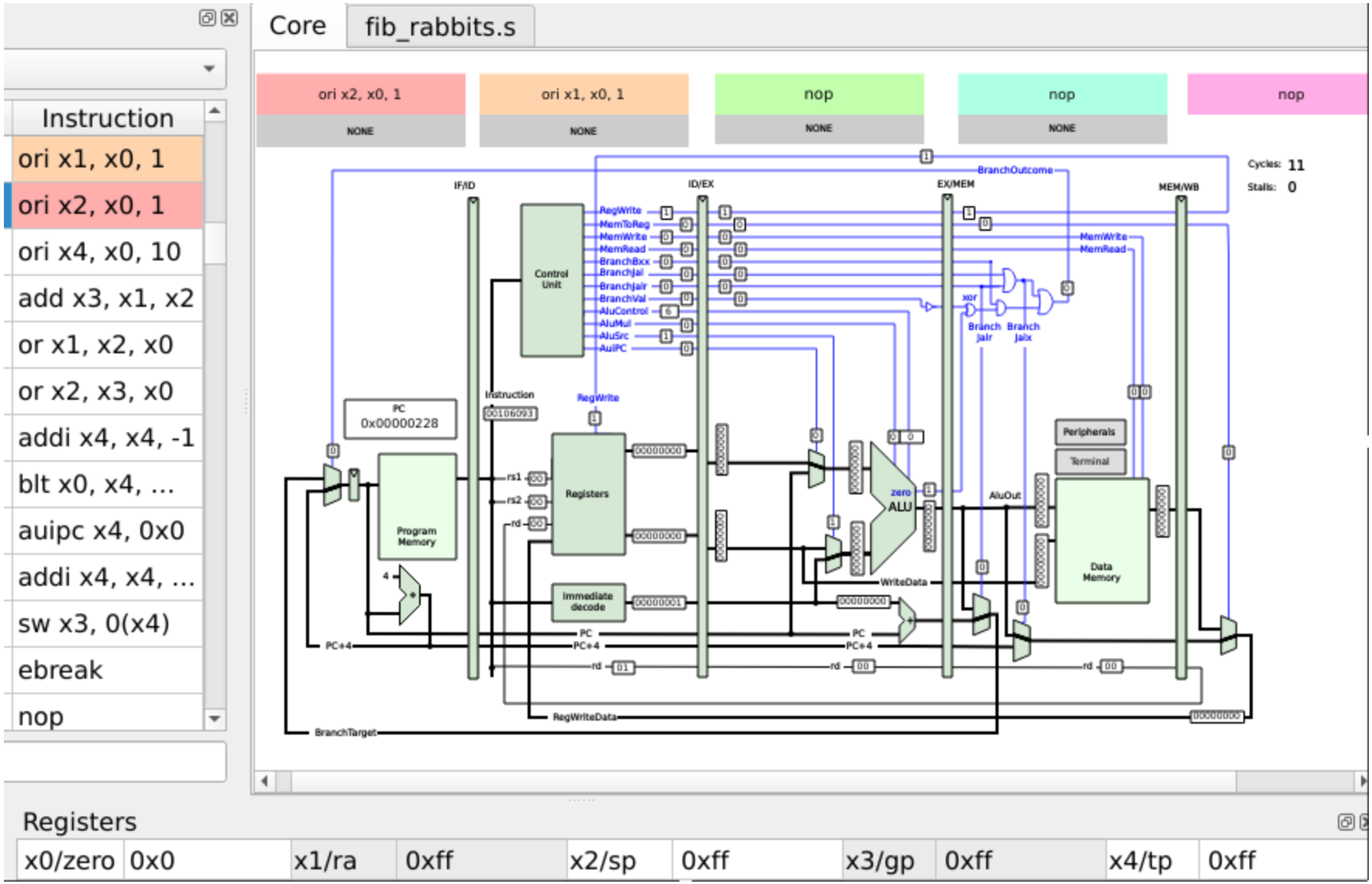
x1	x2	x3	x4	Fetch	Decode	Execute	Memory	Write-Back
FF	FF	FF	FF	x1←1				
FF	FF	FF	FF	x2←1	x1←1			
FF	FF	FF	FF	x4←10	x2←1	x1←1		
FF	FF	FF	FF	x3←x1+x2	x4←10	x2←1	x1←1	
1	FF	FF	FF	x1←x2	x3←x1+x2	x4←10	x2←1	x1←1
1	1	FF	FF	x2←x3	x1←x2	x3←x1+x2	x4←10	x2←1
1	1	FF	10	x4←x4-1	x2←x3	x1←x2	x3←x1+x2	x4←10
1	1	100	10	loop x4<x0	x4←x4-1	x2←x3	x1←x2	x3←x1+x2



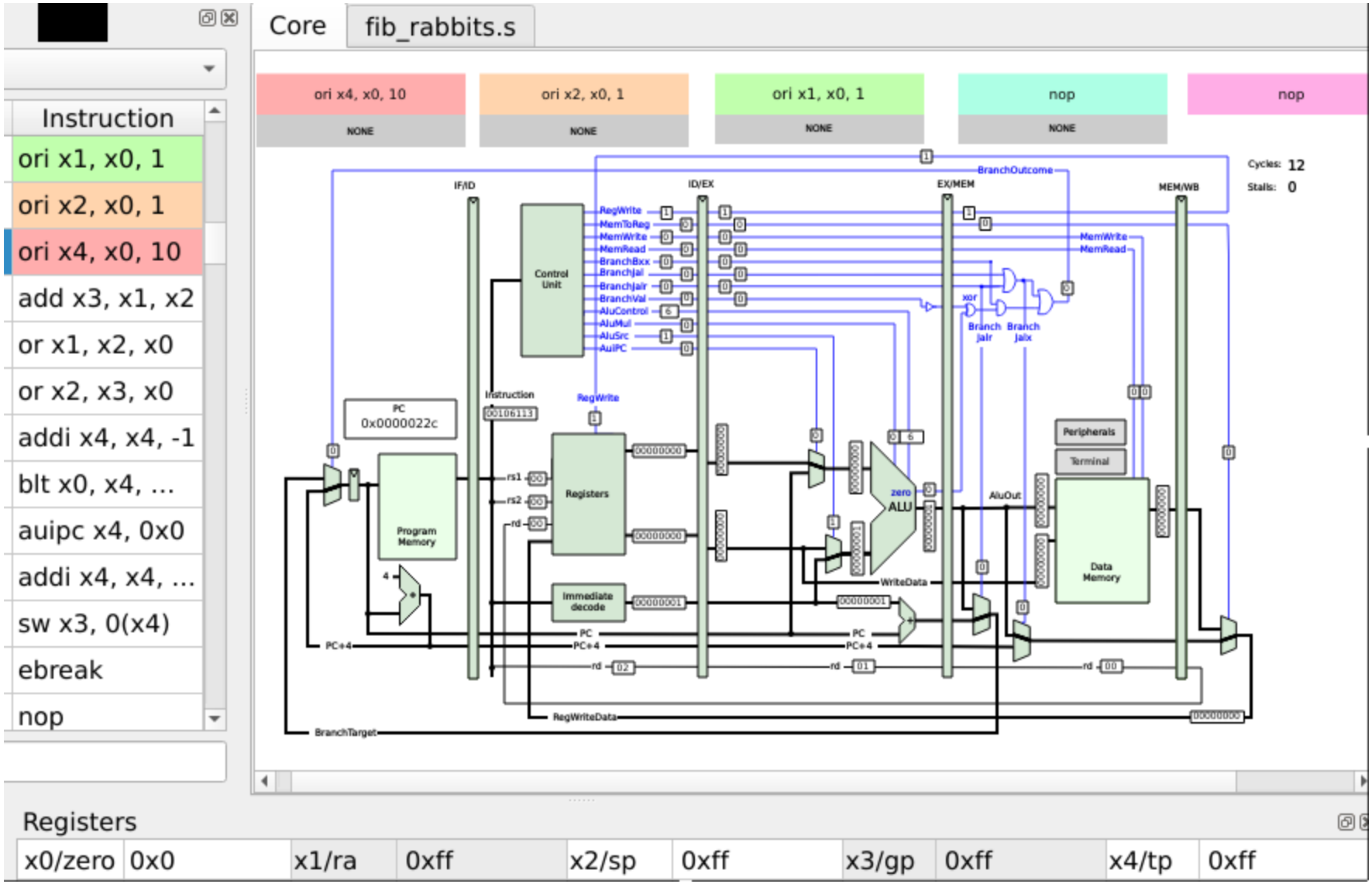
1st instructions in the pipeline



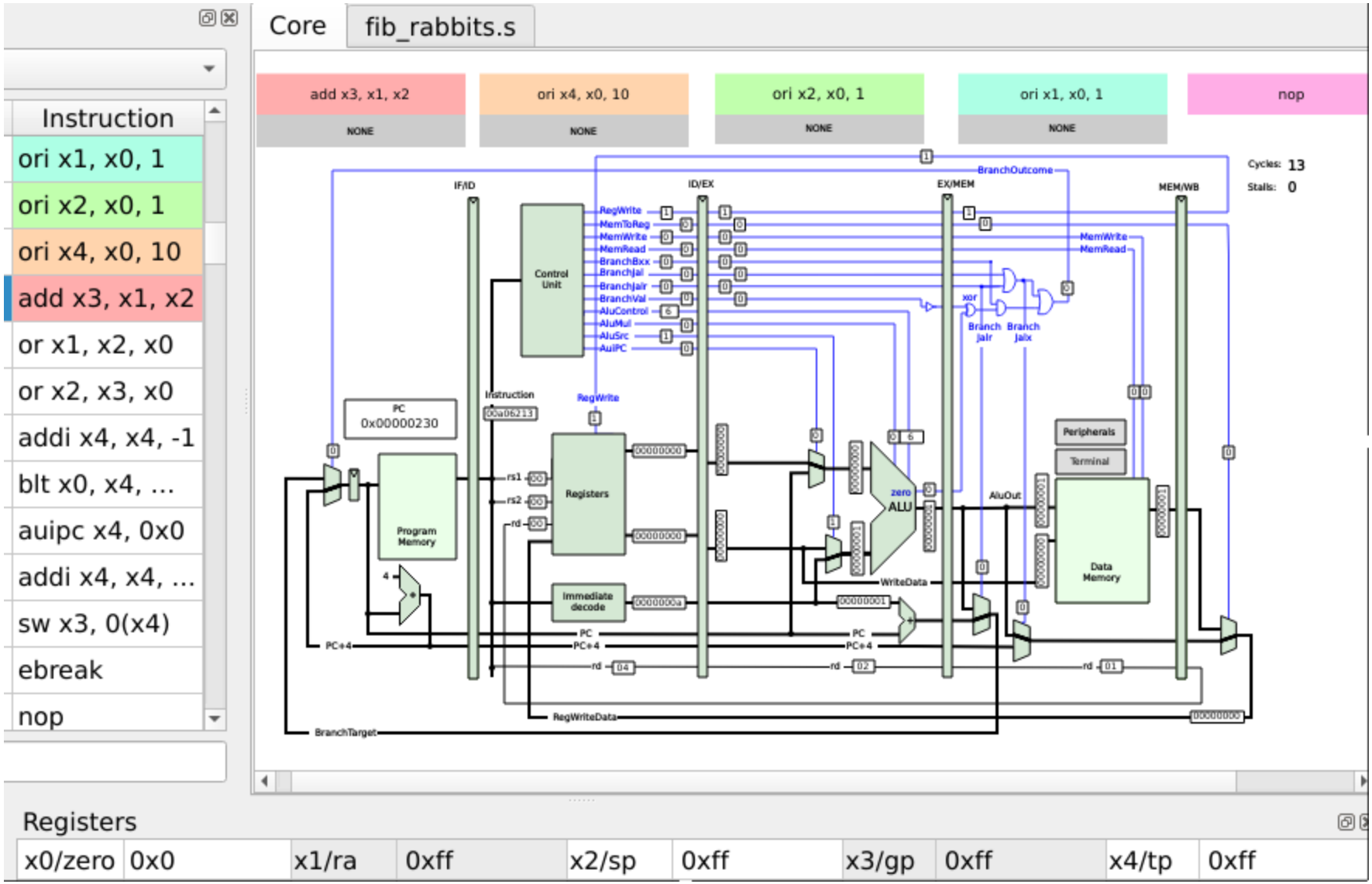
2nd instructions in the pipeline



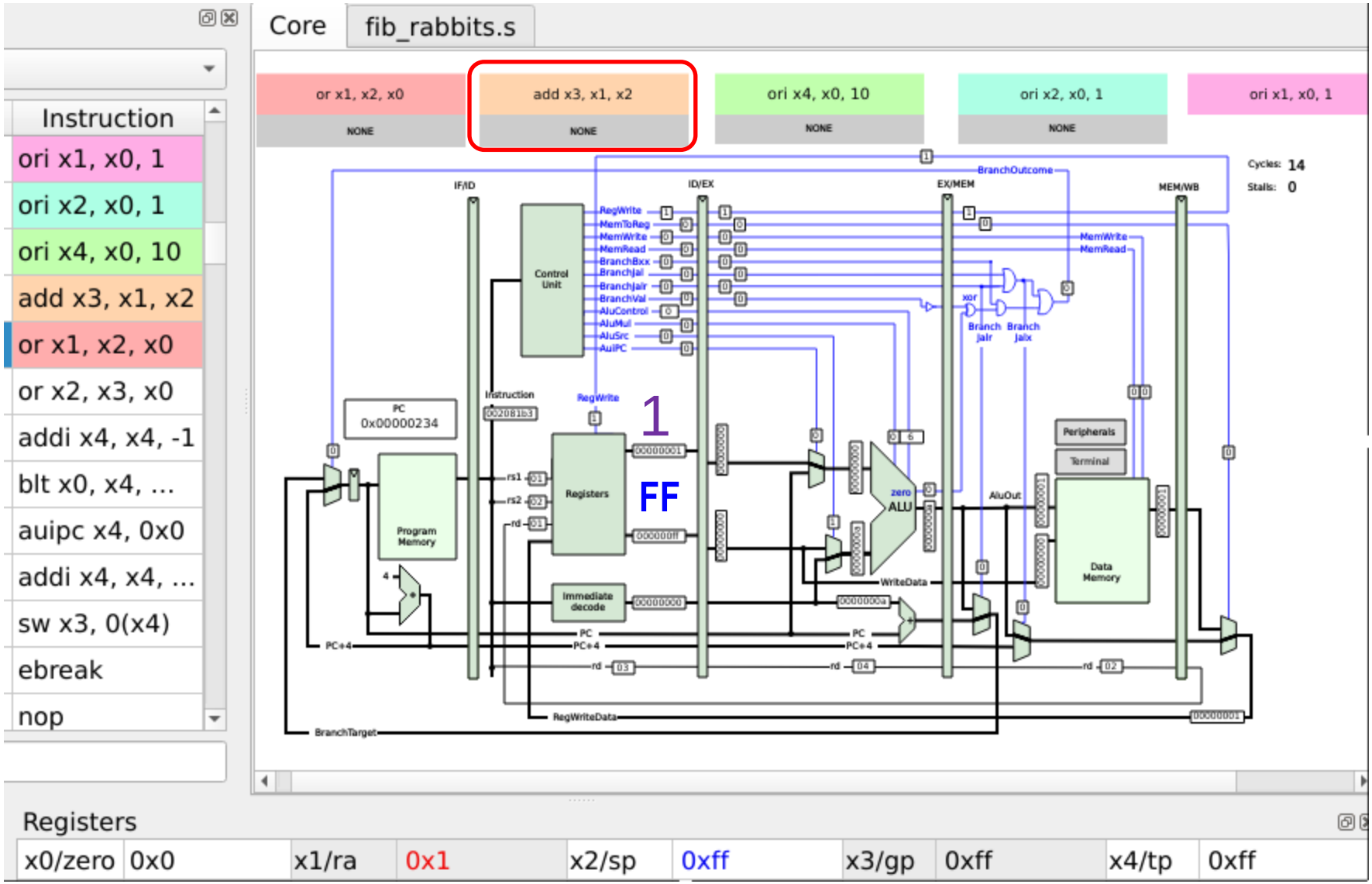
3rd instructions in the pipeline



4th instructions in the pipeline



5th instructions in the pipeline

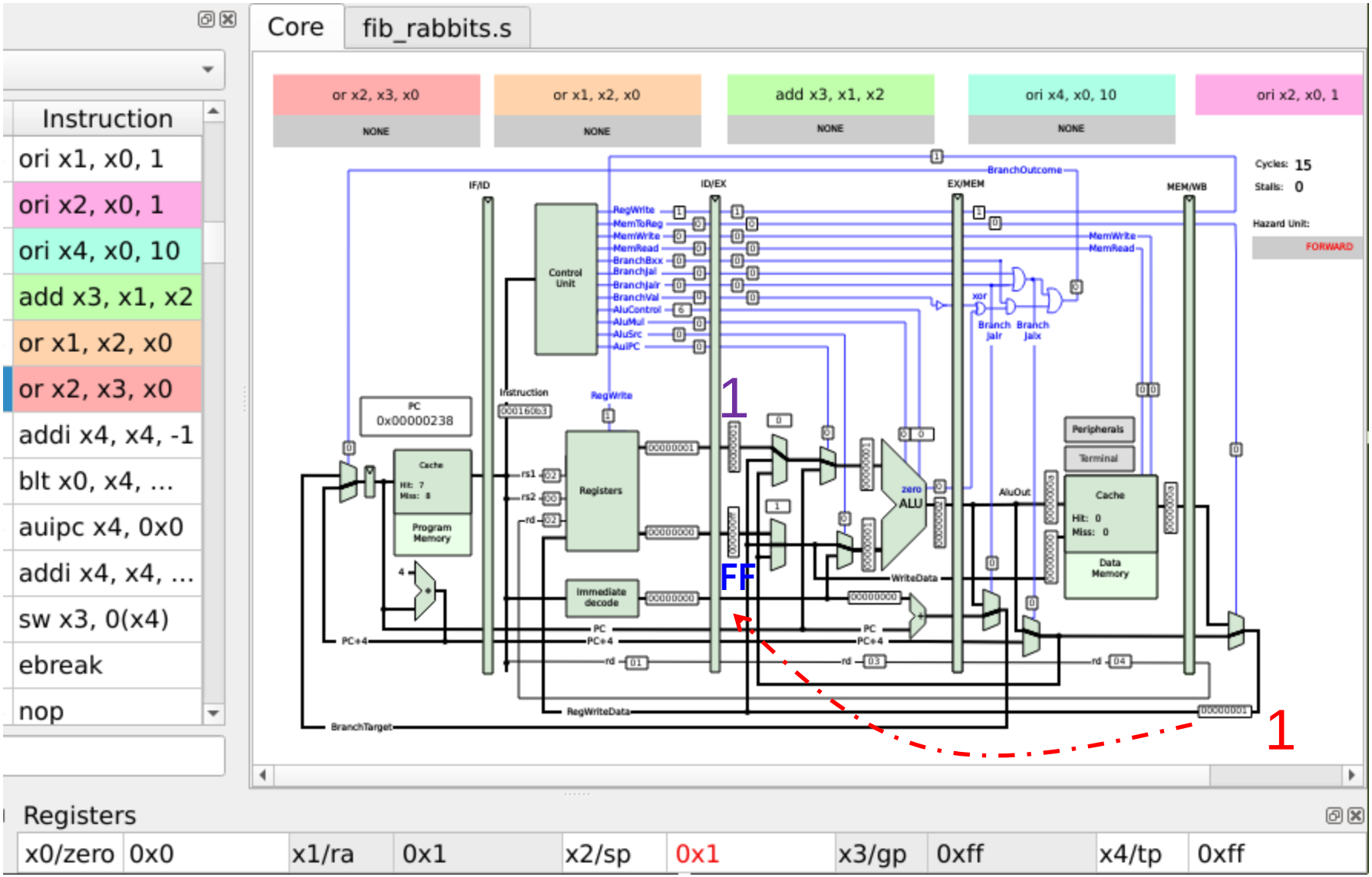


The add instruction got the operands in DECODE according to the immediate values in the registers

Treatment of data hazards by forwarding

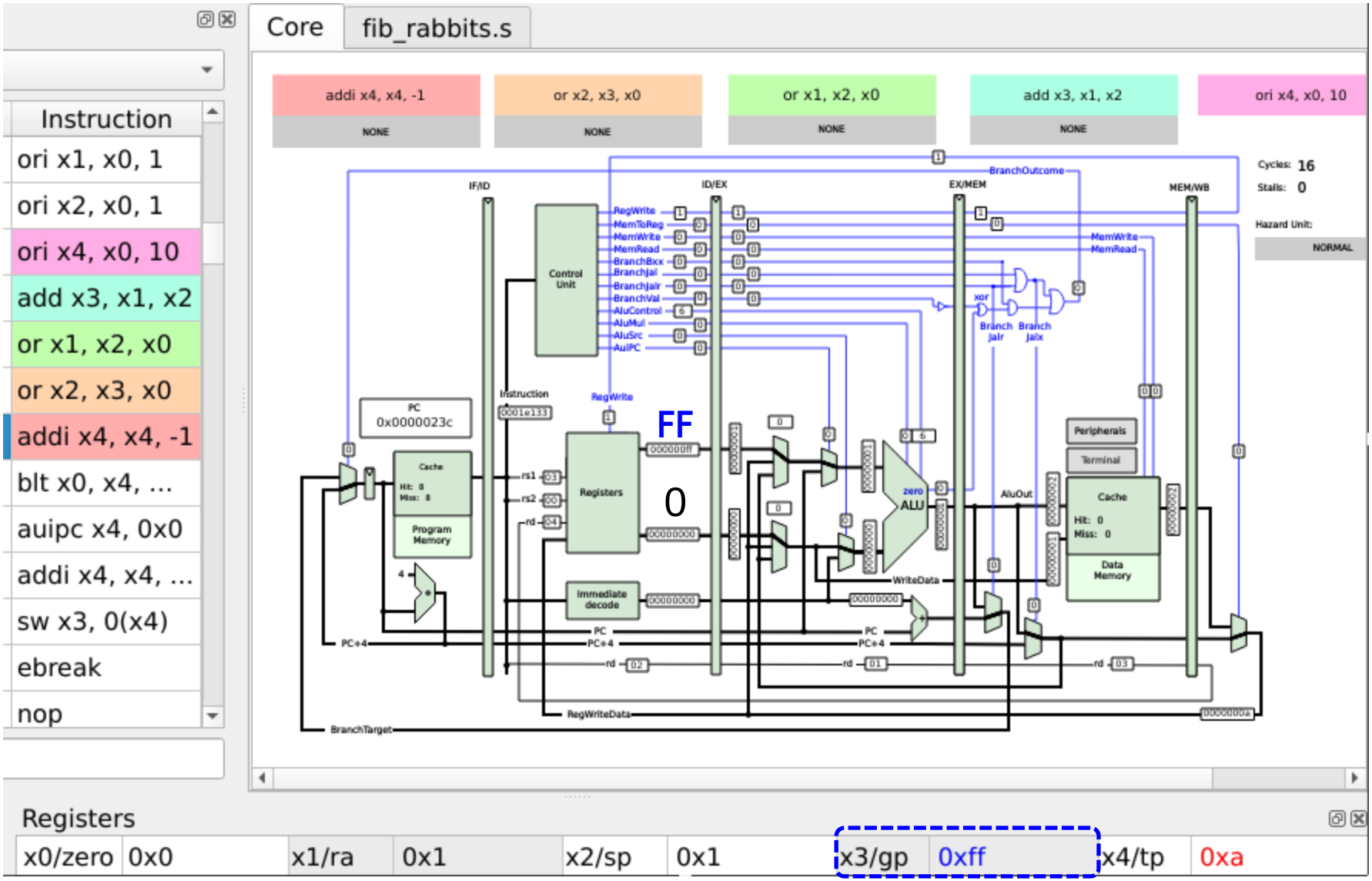
x1	x2	x3	x4	Fetch	Decode	Execute	Memory	Write-Back
FF	FF	FF	FF	x1←1				
FF	FF	FF	FF	x2←1	x1←1			
FF	FF	FF	FF	x4←10	x2←1	x1←1		
FF	FF	FF	FF	x3←x1+x2	x4←10	x2←1	x1←1	
1	FF	FF	FF	x1←x2	x3←x1+x2	x4←10	x2←1	x1←1
					1 FF			
1	1	FF	FF	x2←x3	x1←x2	x3←x1+x2	x4←10	x2←1
					1	1 1		forward
1	1	FF	10	x4←x4-1	x2←x3	x1←x2	x3←x1+x2	x4←10
					FF	1	2	
1	1	2	10	loop x4<x0	x4←x4-1	x2←x3	x1←x2	x3←x1+x2
					10	2	1	2 forward

6th instruction + input correction add x3,x1,x2



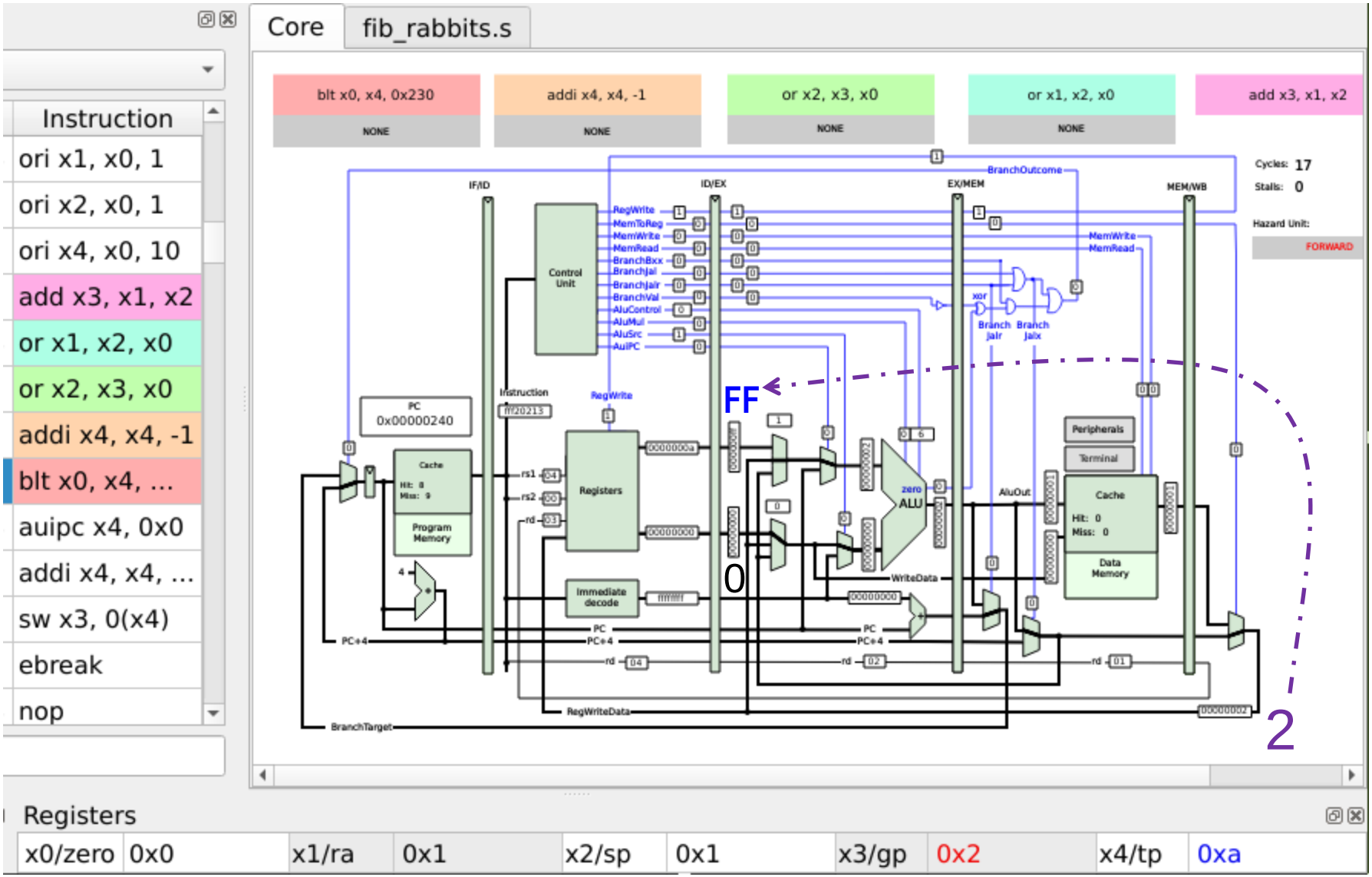
x1 value is sent from WR-BACK to the EXE stage and overwrites *0xFF*

7th instruction



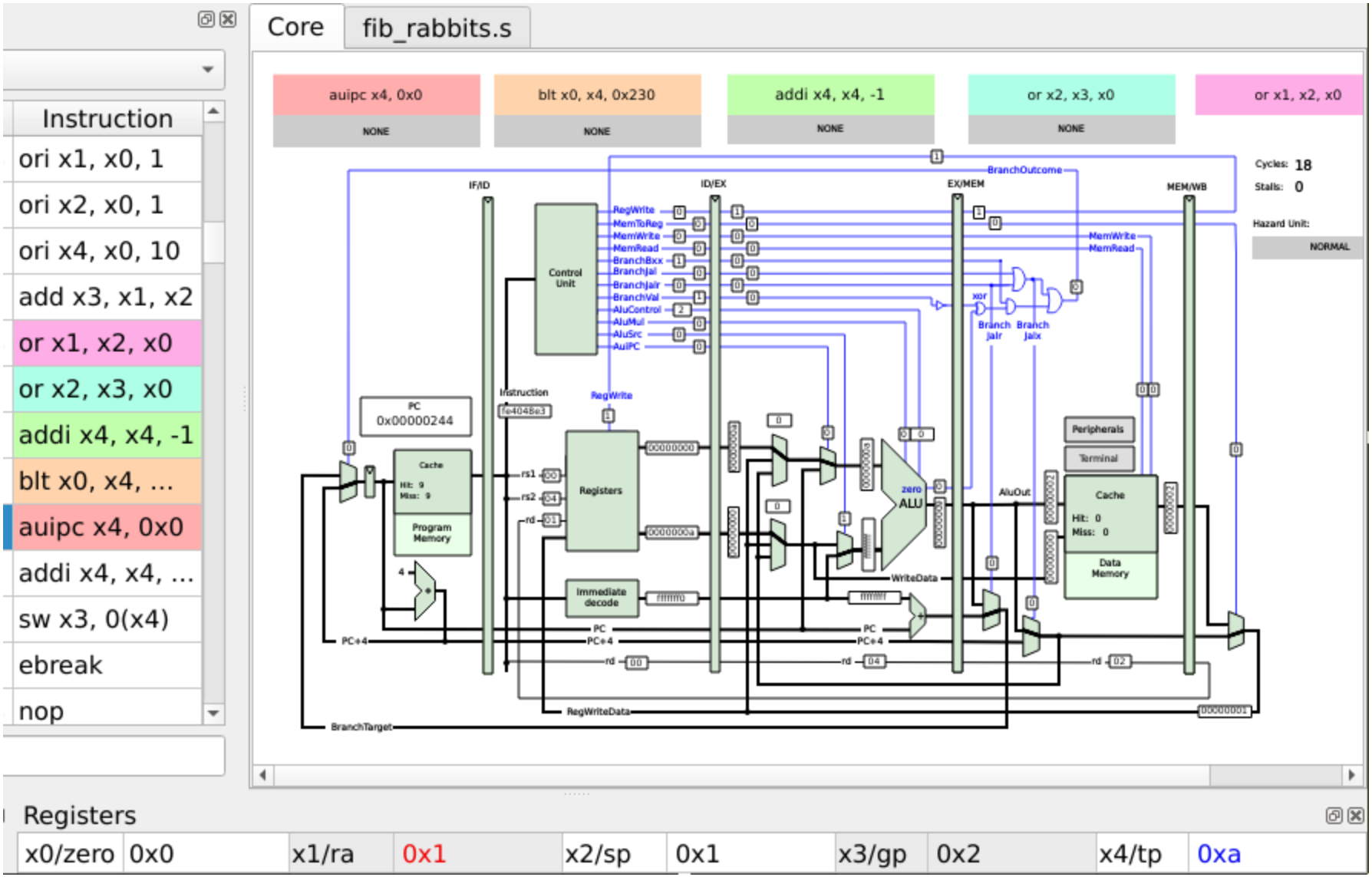
the operands were loaded again from the immediate values in the registers

8th instruction



x3 value is sent from WR-BACK to the EXE stage and overwrites *0xFF*

9th instruction + correct result or x2,x3,x0



Forwarding

Not only from Write-Back is transferred to the EXE stage,

but also from Memory, which occurs with a pair of instructions:

```
la x4, fib13 // x4=&fib13
sw x3, 0(x4) // *fib13=x3
```

translated to

```
auipc x4, 0x0 // x4← PC + 0
addi x4, x4, 480 // x4← x4 + 0x1E0
sw x3, 0(x4)
```

Here the value x4 is transferred twice from MEM to EXE.

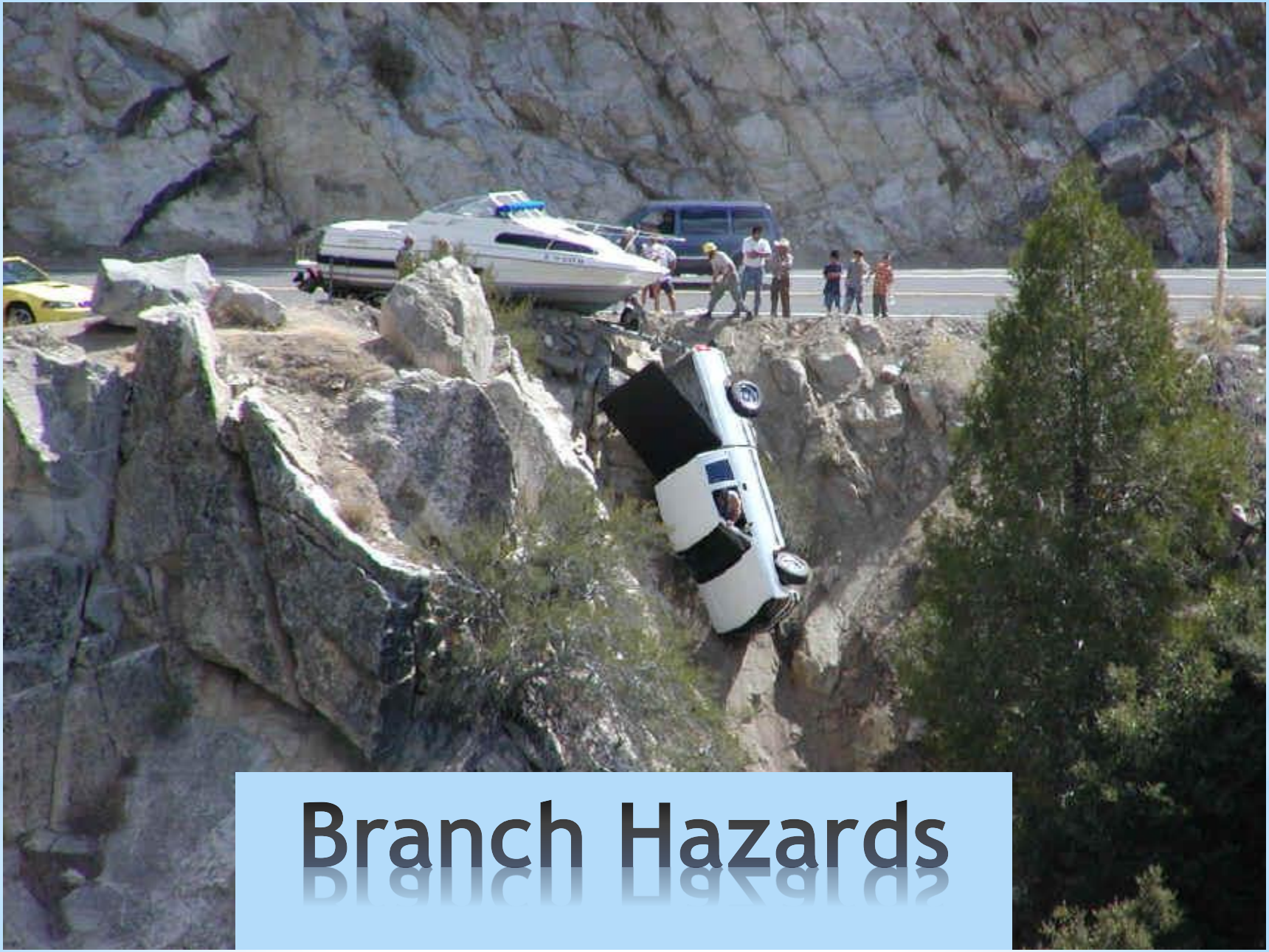
When do we care about data hazards?

- **We don't have to if we are translating for a large processor equipped with Hazard Unit.**
- However, we will treat them on smaller processors without a Hazard Unit, where the code is often corrected by the compiler:

```
_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
        nop // wait for x2
loop: add x3, x1, x2 // x3←x1 + x2
        or x1, x2, zero // x1←x2 or 0 ≡ x1 ← x2
        nop // wait for x3
        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;
        auipc x4, 0x0 // x4← PC + 0; after adding NOP, PC=0x204
        nop // wait for x4
        nop // wait for x4
        addi x4, x4, 476 // x4← x4 + 0x1DC
        nop // wait for x4
        nop // wait for x4
        sw x3, 0(x4) // *x4← x3
```

When do we care about data hazards?

```
_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
        nop // wait for x2
loop: add x3, x1, x2 // x3←x1 + x2 // loop optimized by our instruction reorder
        addi x4, x4, -1 // x4←x3 + -1 ≡ x4- -
        or x1, x2, zero // x1←x2 or 0 ≡ x1 ← x2
        or x2, x3, zero // x2←x3 or 0 ≡ x2 ← x3
        blt zero, x4, loop // if(0<x4) goto loop;
        auipc x4, 0x0 // x4← PC + 0; after adding NOP, PC=204
        nop // wait for x4
        nop // wait for x4
        addi x4, x4, 476 // x4← x4 + 0x1DC
        nop // wait for x4
        nop // wait for x4
        sw x3, 0(x4) // Fib13=x4; Fib13 address = 0x400
```

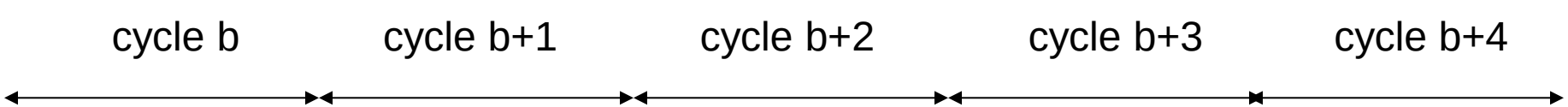


Source: http://www.spaceg.com/multimedia/collection/automobiles/ (Aug 20, 2014)

Branch Not Taken

Branch to Z
A
B
C
D
Z

Assume the heuristic: ***no branching****.*
The instructions continue to be loaded in their order

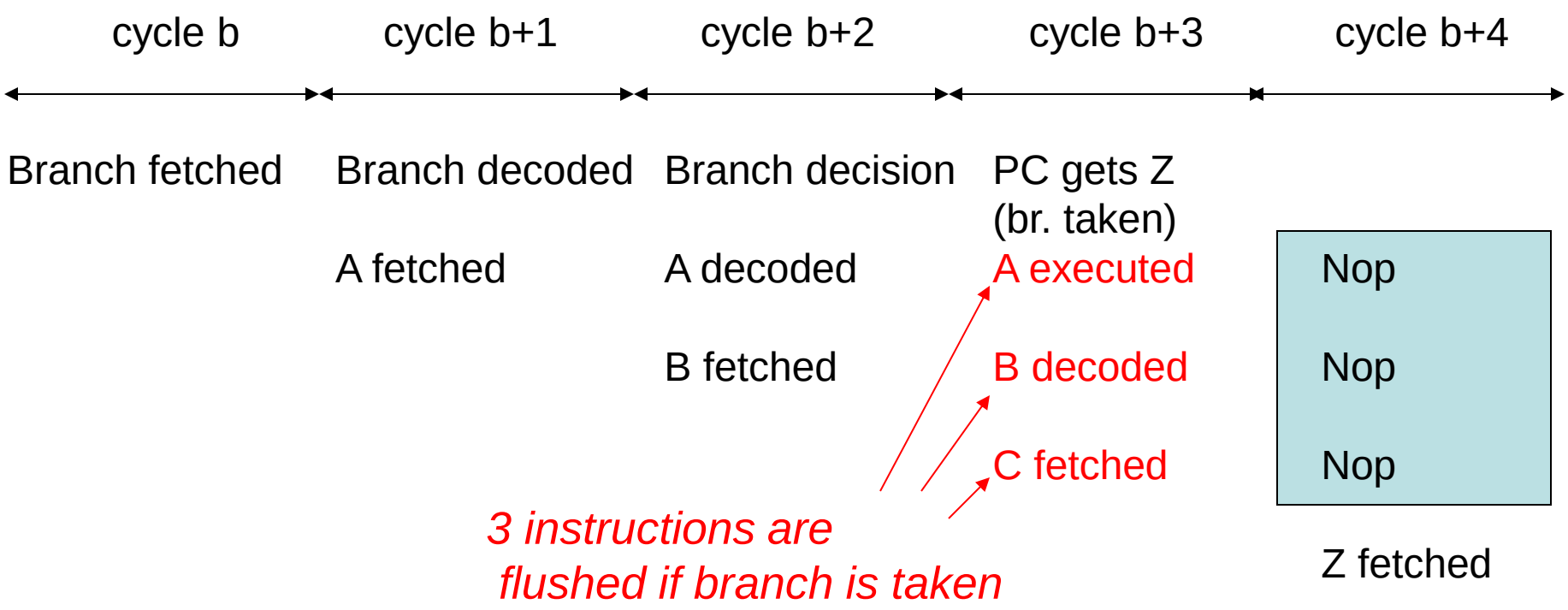


cycle b	cycle b+1	cycle b+2	cycle b+3	cycle b+4
Branch fetched	Branch decoded	Branch decision	PC unchanged **(br. not taken)**	
	A fetched	A decoded	A executed	A continues
		B fetched	B decoded	B executed
			C fetched	C decoded
				D fetched

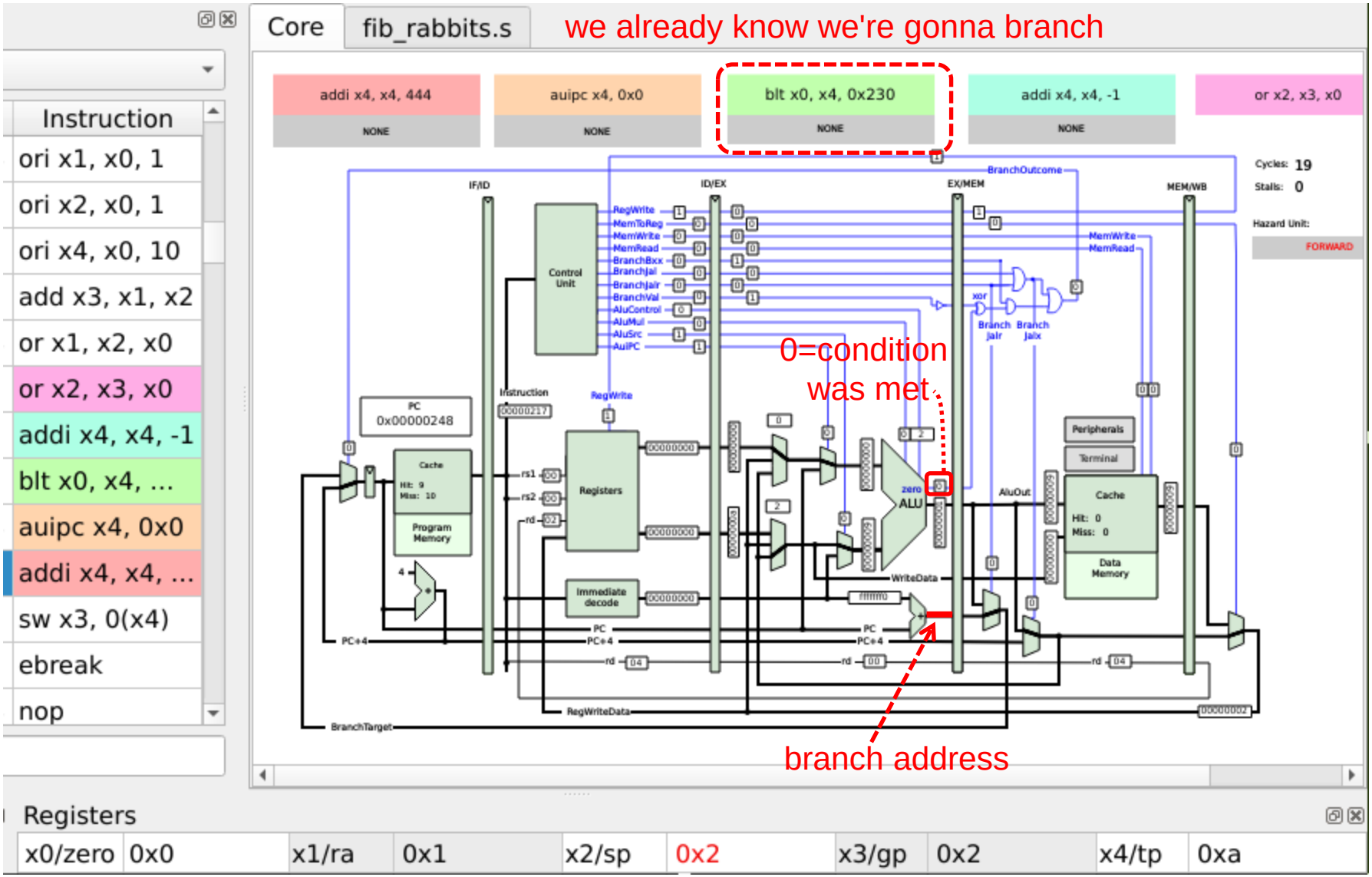
Branch Taken

If branching, branching is generated in the MEM stage. All work in progress is deleted and a new instruction is loaded.

Branch to Z
A
B
C
D
Z

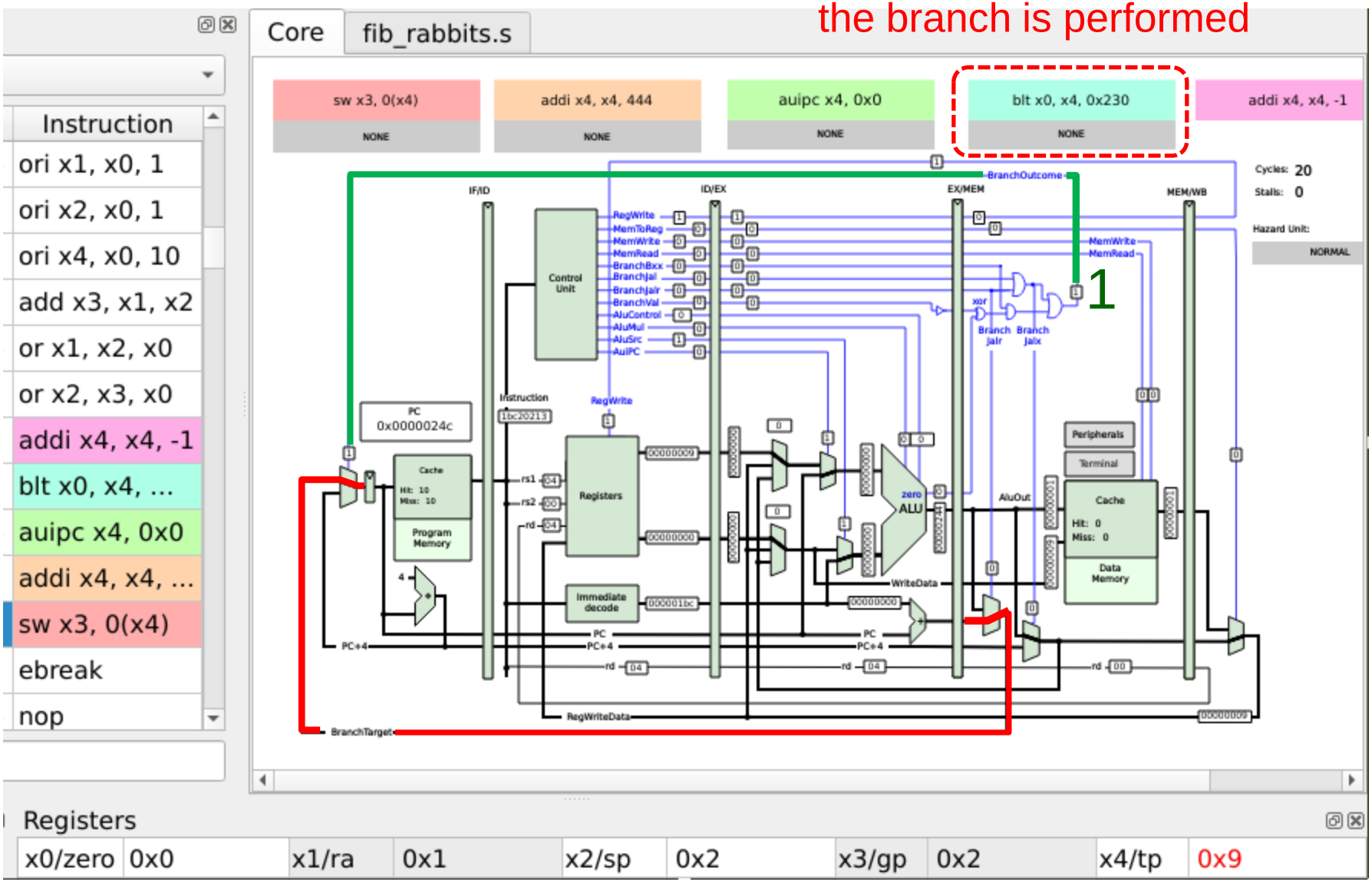


Branch Hazard

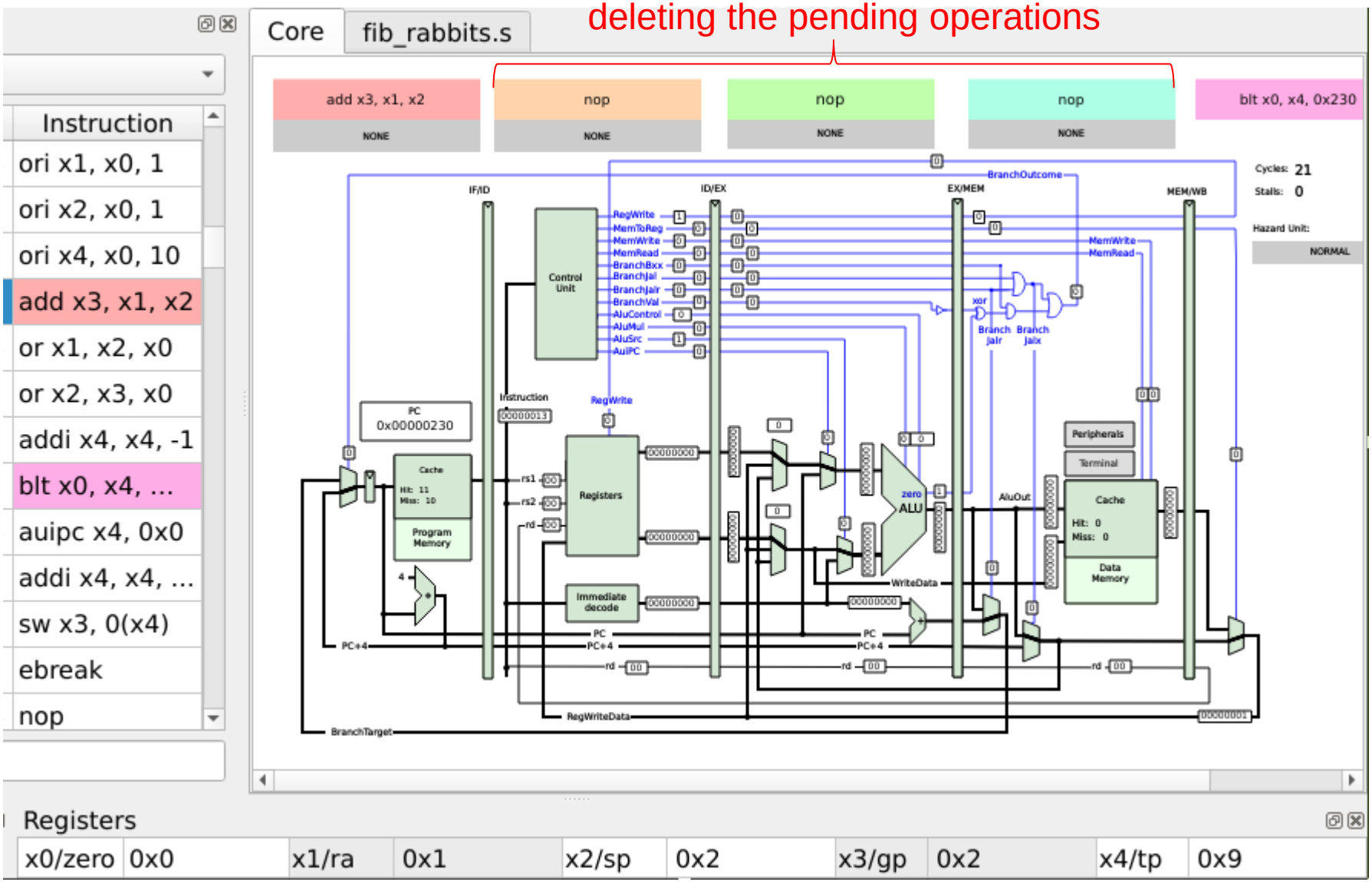


In the MEM stage, it will be decided whether to branch

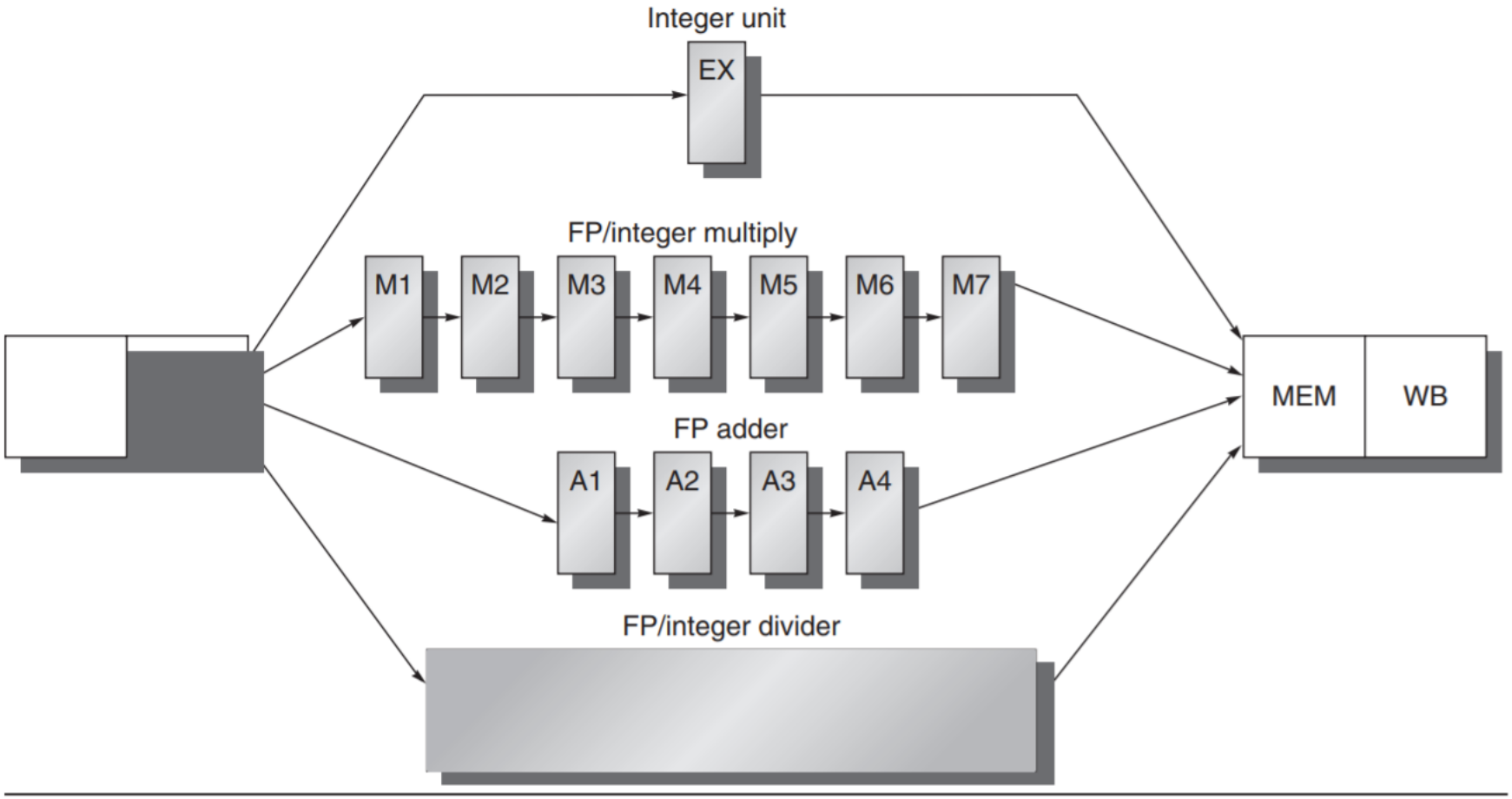
the branch is performed



Read instructions are flushed



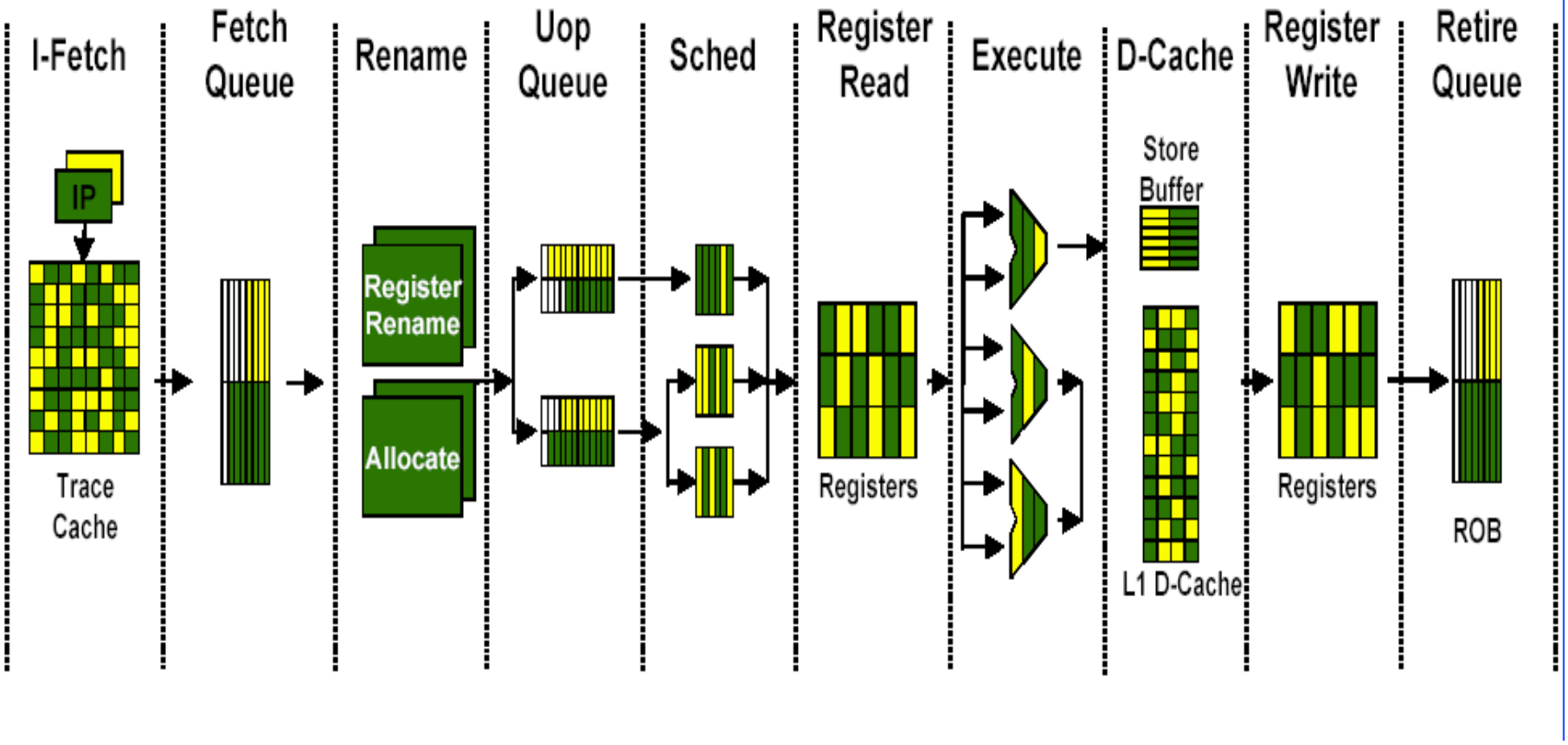
ALU can have multiple parallel data paths



Source of picture: J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.*

Bigger processors have longer pipelines

Pentium 4: Netburst Microarchitecture's



*Simplified picture, the pipeline actually has **20 steps**.*
The penalty for branching is high.

What is the length of pipelines? – approx. …

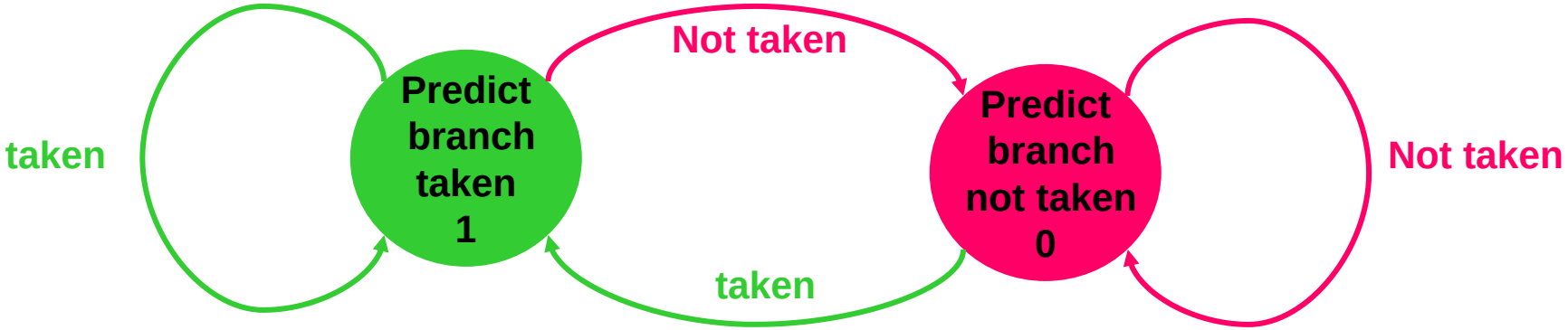
5	1993	P5 (Pentium)
10	1995	P6 (Pentium 3)
14	1995	P6 (Pentium Pro)
20	2000	NetBurst (Willamette, 180 nm) - Celeron, Pentium 4:
20	2002	NetBurst (Northwood, 130 nm) - Celeron, Pentium 4+ HT
31	2004	NetBurst (Prescott, 90 nm) - Celeron D, Pentium 4 ExEd
31	2005	NetBurst (Cedar Mill, 65 nm)
31	2006	NetBurst (Presler 65 nm) – Pentium D
14	**2020-**	**generation Intel I3, I5, I7 , 10 and following**
5	2005	ARM 8-9: **5**
13	2005	Cortex A8: **13**
5-25	2010	Cortex A15: **15-25**
8-10	2011	Cortex A7: **8-10**
10	**2021-**	**Cortex X2, X3**

- The Optimum Pipeline Depth for a Microprocessor: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.4333&rep=rep1&type=pdf

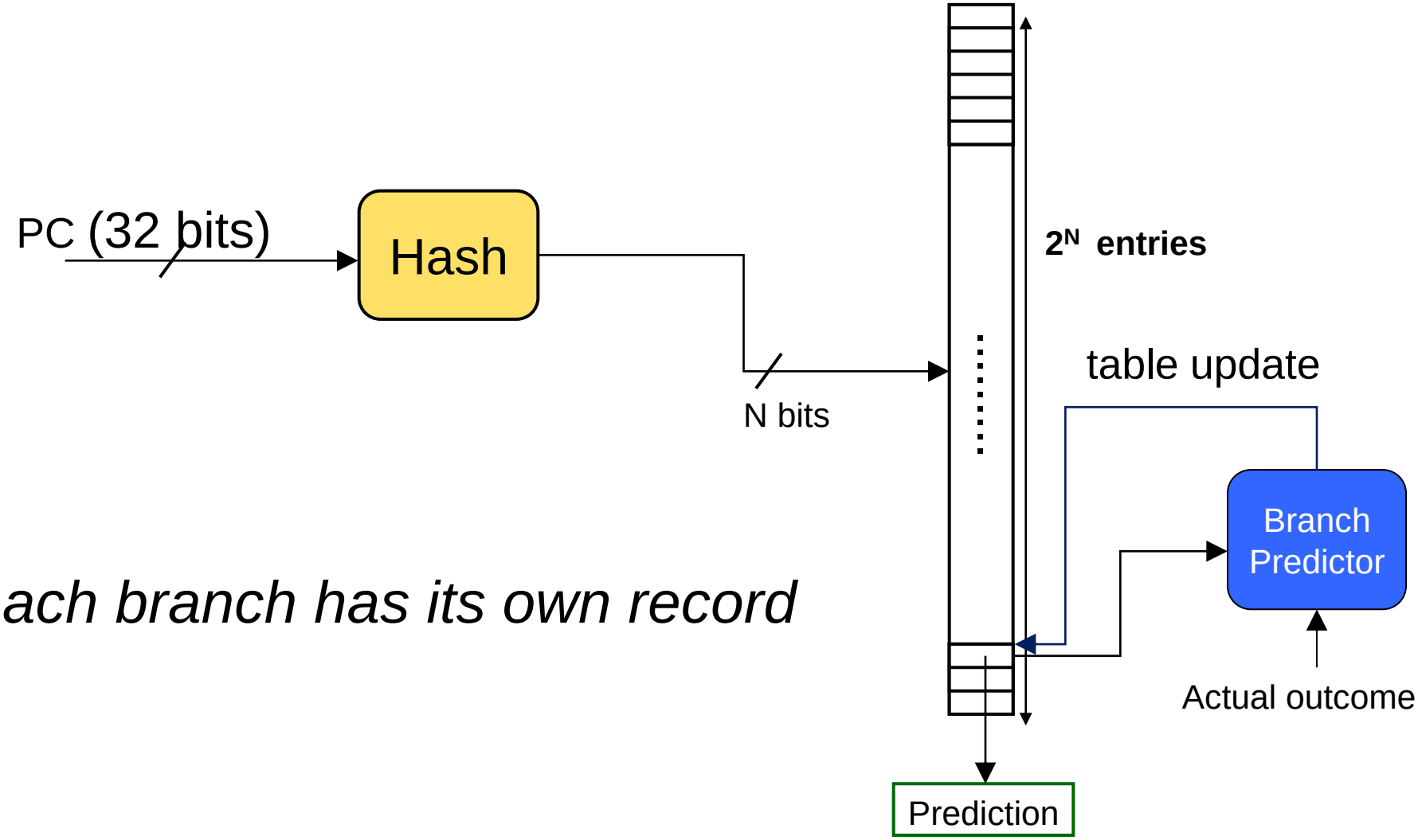
Prediction of Branches

1-bit prediction

- 1-bit prediction:
  has 1 "history bit" that remembers what was done,
  when the instruction was last executed:
  - History bit = 1, branch was previously **Taken**
  - History bit = 0, branch was previously **Not taken**

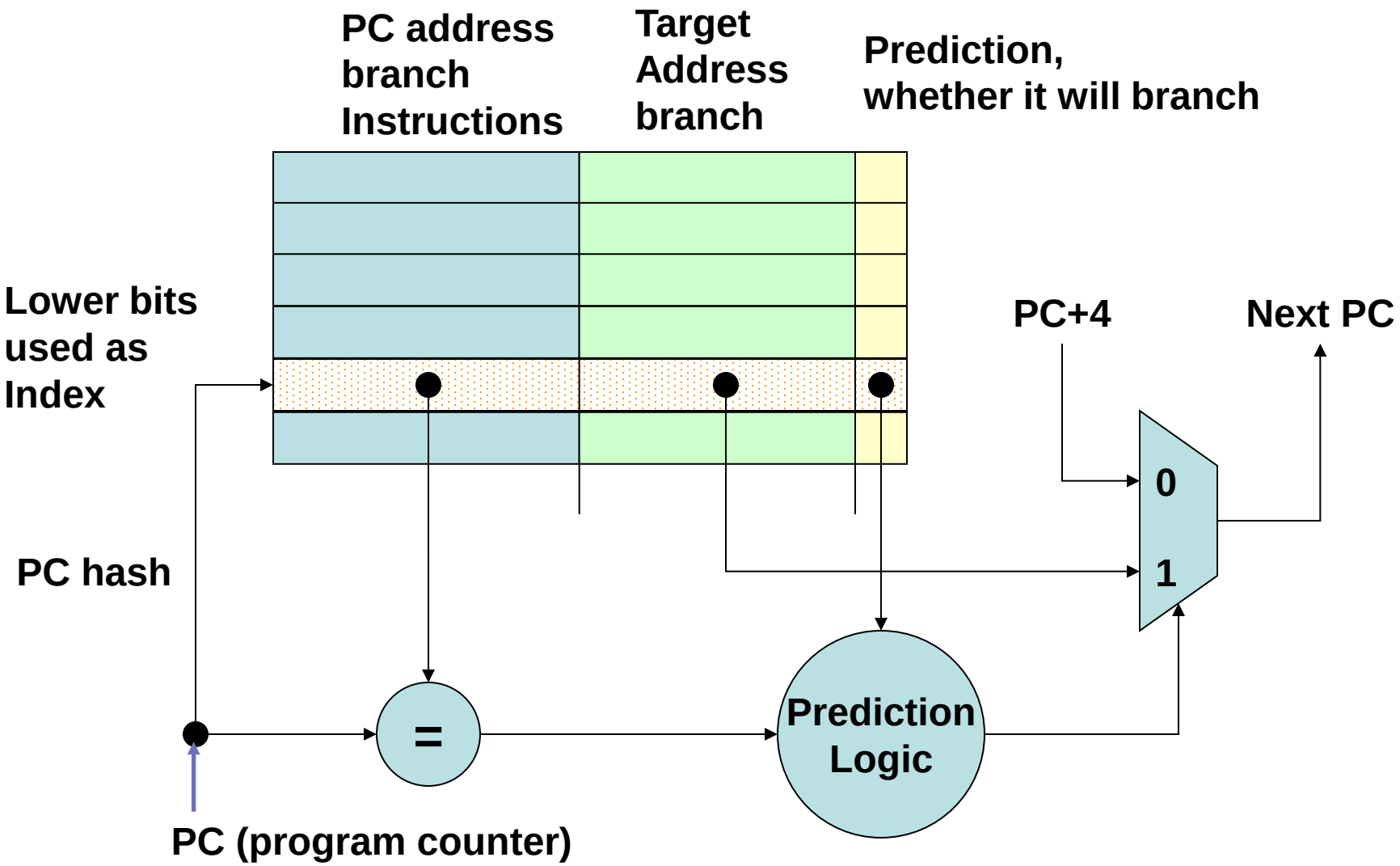


Normal organization of the prediction table

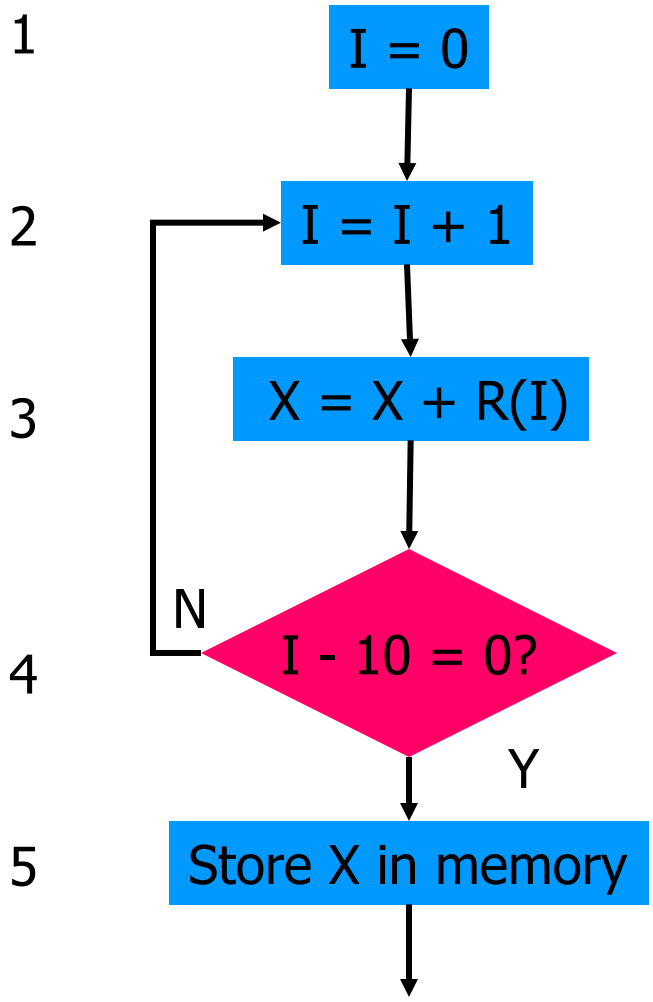


each branch has its own record

Branch Prediction



Branch Prediction for a Loop



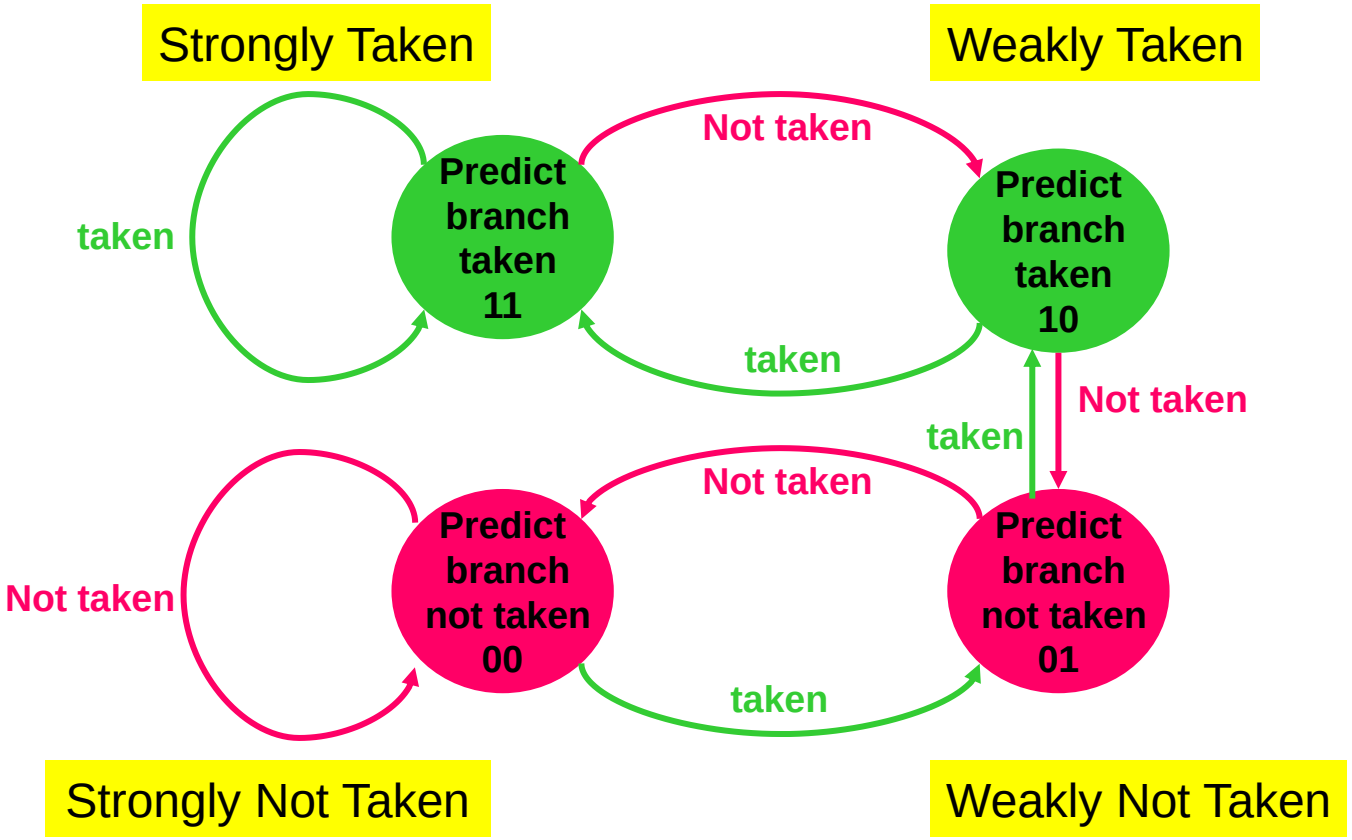
Execution of Instruction 4

Execu-tion seq.	Old Hist. bit	Next instr.			New hist. bit	Predi ction
		Befor e.	I	Act.		
1	0	5	1	2	1	Bad
2	1	2	2	2	1	Good
3	1	2	3	2	1	Good
4	1	2	4	2	1	Good
5	1	2	5	2	1	Good
6	1	2	6	2	1	Good
7	1	2	7	2	1	Good
8	1	2	8	2	1	Good
9	1	2	9	2	1	Good
10	1	2	10	5	0	Bad

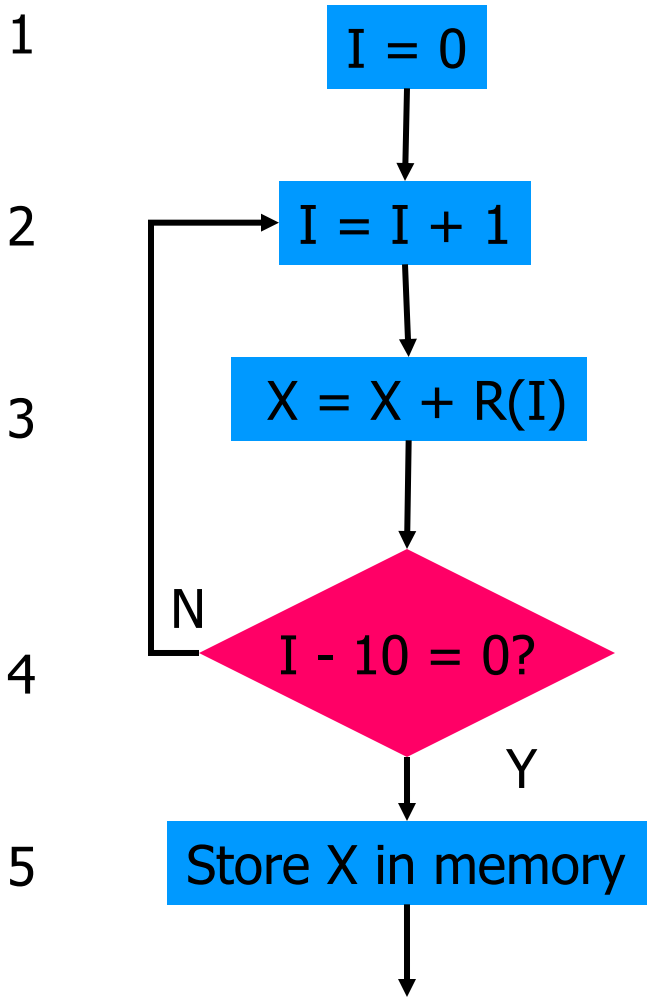
bit = 0 *branch not taken*, bit = 1 *branch taken*.

2-bit predictor

- It is also called "**saturating counter!**.



Branch Prediction for a Loop



Execution of Instruction 4

Execu-tion seq.	Old Befor e.Buf	Next instr.			New pred. Buf	Predi ction
		Befor e.	I	Act.		
1	10	2	1	2	11	Good
2	11	2	2	2	11	Good
3	11	2	3	2	11	Good
4	11	2	4	2	11	Good
5	11	2	5	2	11	Good
6	11	2	6	2	11	Good
7	11	2	7	2	11	Good
8	11	2	8	2	11	Good
9	11	2	9	2	11	Good
10	11	2	10	5	10	Bad

Example 1

```
for (int i = 0; i < 100; i++)
{
   for (int j = 0; j < 10; j++)
   {
     // some code
   }
}
```

Number of wrong predictions:

1-bit predictor: outer loop 2 + inner loop 100*2=202

2-bit predictor: outer loop 1 + inner loop 100*1=101

Example 2

```
int arr[] = { 1, 2, 3, -10, -20, 30, 100, -200, -300};
int count = 0;
for (int i = 0; i < sizeof(arr)/sizeof(int); i++)
{ if (arr[i] < 0) count++; } // assembler if(arr[i]>=0) goto labelNext
```

Number of wrong predictions:

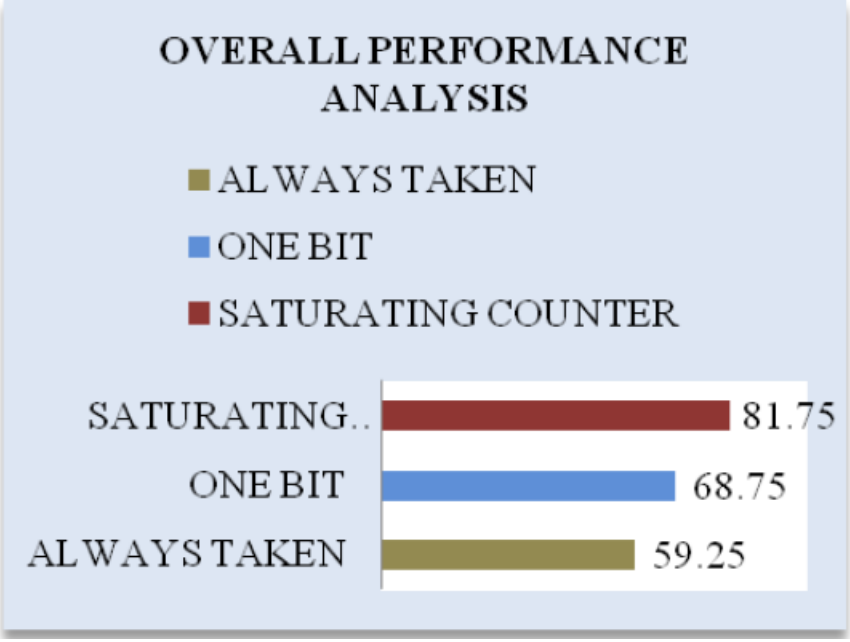
1-bit init NT predictor: outer loop 2 + inner loop 4

2-bit init WNT predictor: outer loop 2 + inner loop 6

2-bit init WT predictor: outer loop 1+ inner loop 5

Number	1	2	3	-10	-20	30	100	-200	-300
Assembler Needs	T	T	T	NT	NT	T	T	NT	NT
1bit Predictor	NT	T	T	T	NT	NT	T	T	NT
2bit Predictor WNT	WNT	WT	T	T	WT	WNT	WT	T	WT
2bit Predictor WT	WT	T	T	T	WT	WNT	WT	T	WT

Some Available Statistics



The larger numbers in the picture mean better prediction

Source : https://ieeexplore.ieee.org/document/6918861
H. Arora, S. Kotecha and R. Samyal, "Dynamic Branch Prediction Modeller for RISC Architecture," *2013 International Conference on Machine Intelligence and Research Advancement*, Katra, 2013, pp. 397-401.

Correlated or Correlating Predictors

They consider previous branchs

```
if (x==2) // branch b1
                    ...
if (y==2) // branch b2
                    ...
if(x!=y) { ... } // branch b3 depends on the
                    results of b1 and b2
```

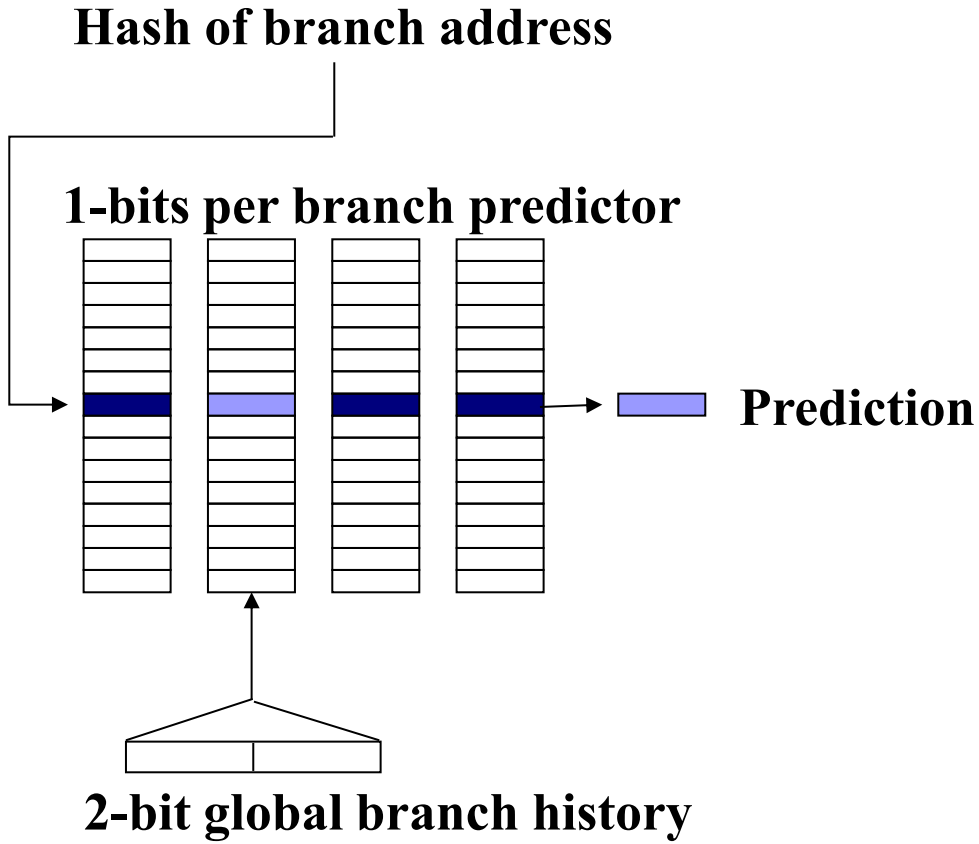
(2,1) Correlated predictor

We switch between the results of 4 predictors: **P00 | P01 | P10 | P11**

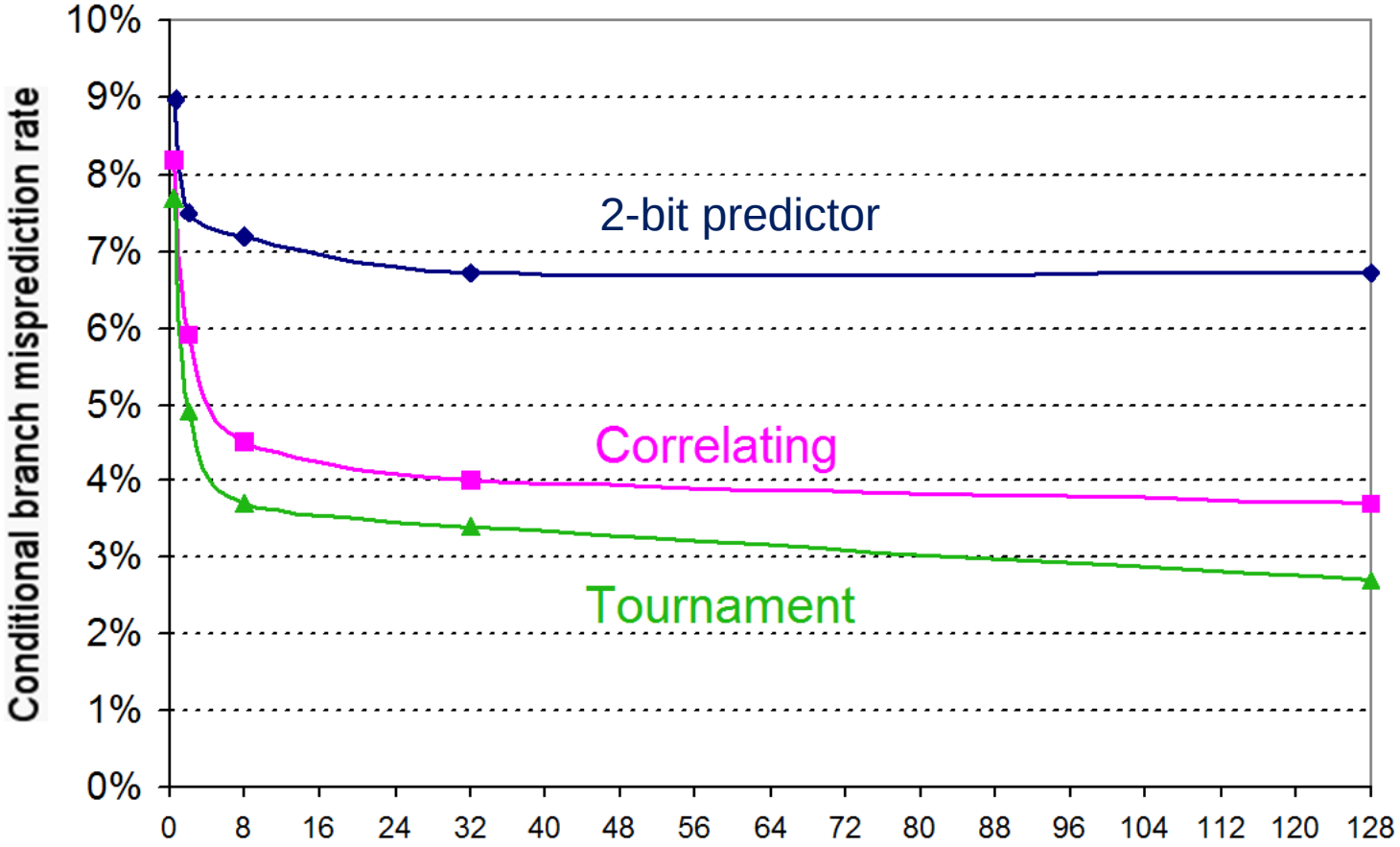
The predictor is used when	**P00**	**P01**	**P10**	**P11**
1. the previous branch was	Not taken	Not taken	Taken from	Taken from
2. the previous branch was	Not taken	Taken from	Not taken	Taken from

A (2,1) correlation predictor
- (**2**,**1**) means 2^{2} =4 predictors, **1** bit each
- **2** indicates that the result of the previous 2 branchs is selected.

- Example (2,1) predictor



A little bit of statistics again



Tournament - multiple predictors of different types. They all predict, but the result of the one with the most previous successes is taken. They are used in large processors, but the manufacturers hide their exact principles.

Branches can sometimes be avoided

There are many tricks on the web suitable for time-critical loops. For example, the absolute value of a 32-bit signed integer x can be calculated without jumps.

Code with *unpredictable branch* dependable on data

C code	***RISC V if x in t1***	***Comment***
if(x<0) x=-x;	**blt** x1, zero, **Skip1**	// if(tmp==0) goto Skip
	sub x1, zero, x1	// x = - x;
Skip1:	...	

Faster C code	***RISC V if x in x1***	***Comment***
int tmp = x>>31;	sra x2, x1, 31	// signed right shift; tmp = x<0 ? -1 : 0
x ^= tmp;	xor x1, x1, x2	// 1st compliment of x, if tmp=-1 (0xFFFFFFFF)
x -= tmp;	sub x1, x1, x2	// add +1 if tmp = -1

Note: For RISCV, we save only 1 instruction out of 3 that are deleted from the pipeline.
If we compile C code for a processor with a longer pipeline, the misprediction will be more expensive.

```
int CountMinus(int* arr, int len)
{ int count = 0;
  for (int i = 0; i < len; i++) { if (arr[i] < 0) count++; }
  return count;
}
```

```
int CountMinusNoBranch(int* arr, int len)
{
  int count = 0;
  for (int i = 0; i < len; i++) { count -= (arr[i]>>31); }
  return count;
}
```

Visual Studio 2022 C++ Compiler

In DEBUG mode (), CountMinusNoBranch () *is translated to code, which will be 3* faster than* CountMinus().

In RELEASE mode, both variants run equally fast because the compiler has replaced

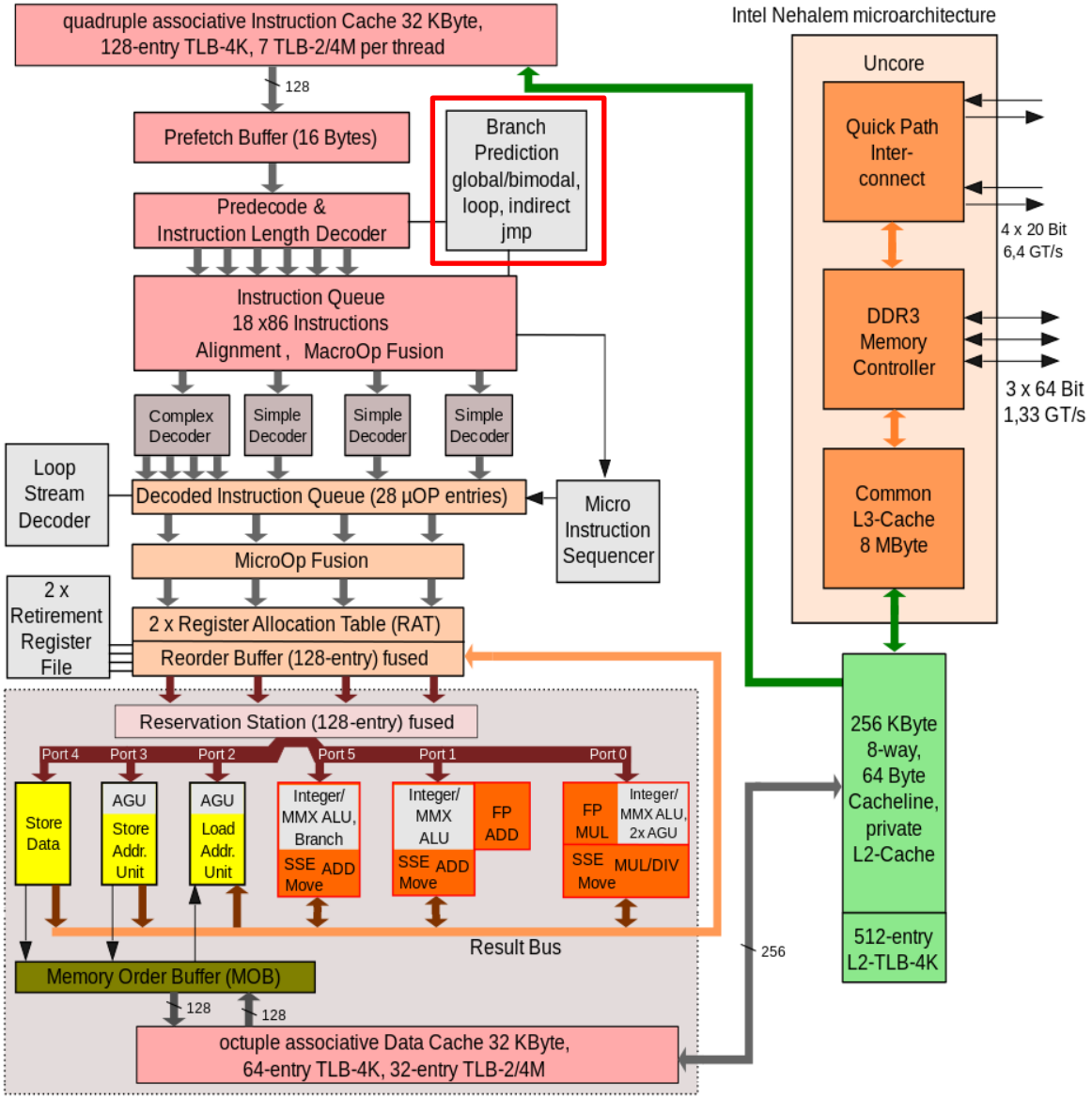
```
if (arr[i] < 0) count++;
```

during jump-free optimizations

```
count -= (arr[i]>>31);
```

However, it cannot handle more complex data-dependent jumps. It's always worth considering whether your algorithm could be modified so that there aren't so many of them.

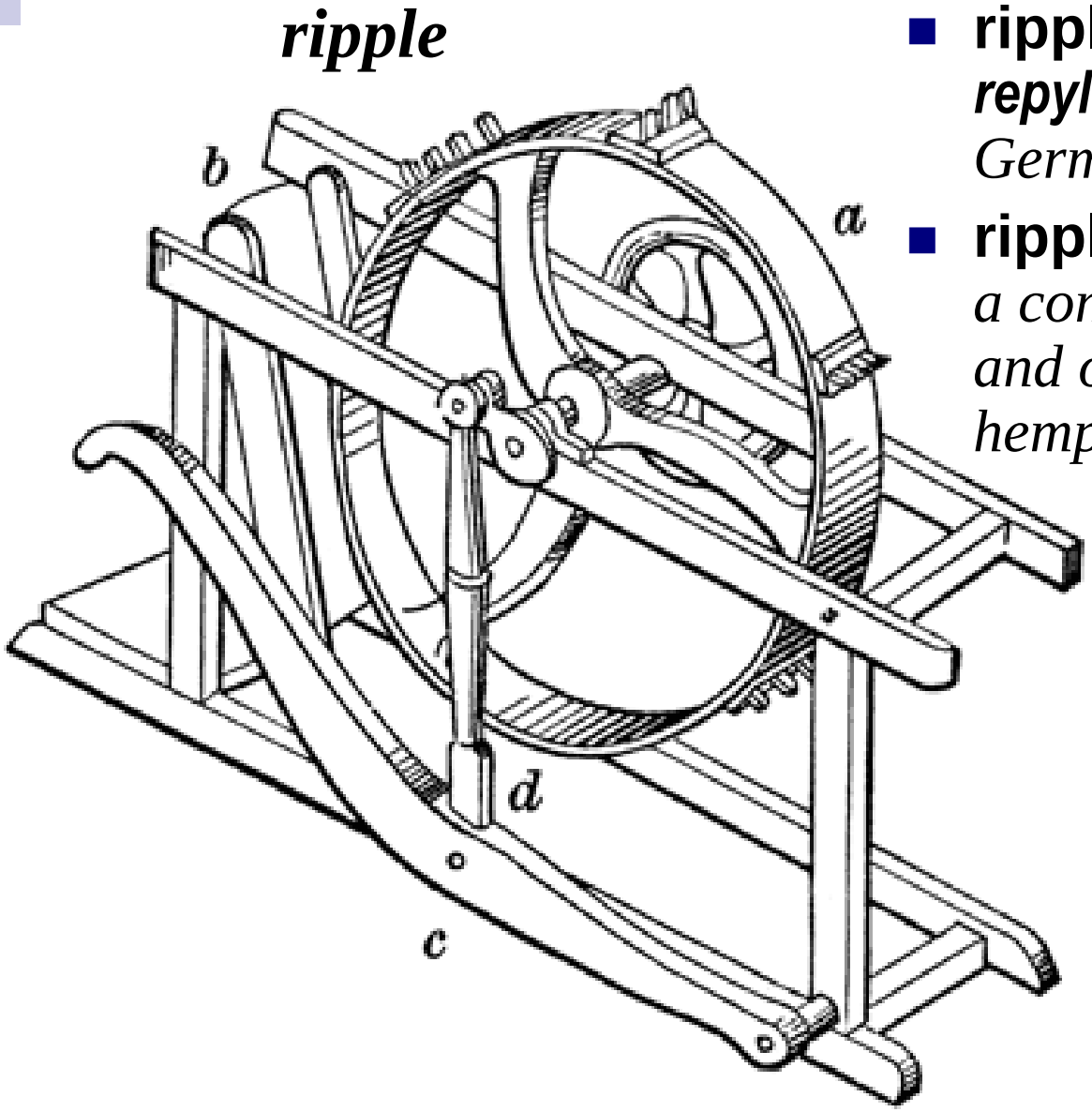
Motivation for the lecture - Intel Nehalem (Core i7) - 2008



- http://en.wikipedia.org/wiki/File:Intel_Nehalem_arch.svg

ALU and Ripple Effect

Ripple (1st meaning)



- **ripple** *from Middle English.* ***repylle*** *akin to Old High German* ***riffila=*** *saw*
- **ripple** *a large instrument like a comb for removing seeds and other matter from flax or hemp*

Ripple (2nd meaning)

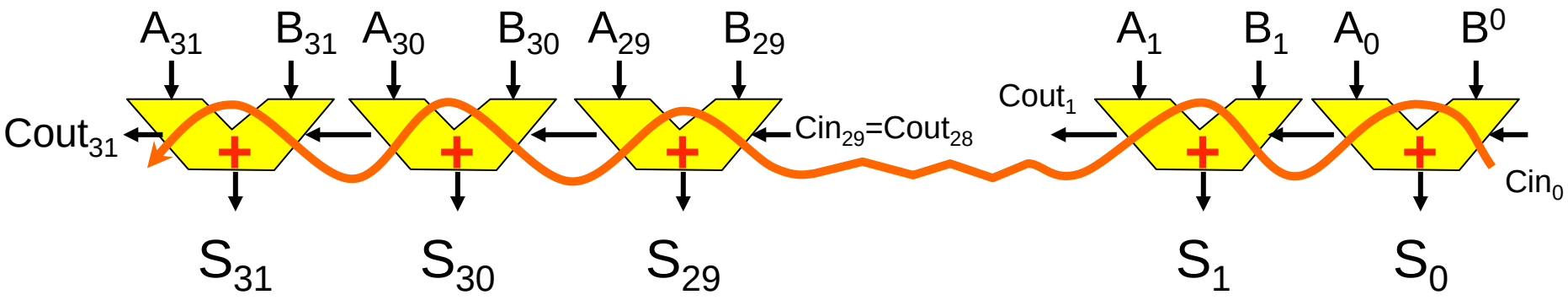
- ***ripple***: *slightly undulating on the surface or appearing in small waves*
- *Ripple Effect*
  in economics:
  the Domino or Avalanche effect
  in physics: Chain Reaction

Adders and multipliers

- The SoP (coverage 1) and PoS (coverage 0) methods express logic functions as expressions with three levels:
  SoP: input inverters, AND gates of implicants, and a merging OR gate.
  PoS: inverters of inputs, OR gates of implicants and merging AND gate.
- Both adders and multipliers are described by Karnaugh maps (KM), which contain many small groups of 0 and 1. Their coverage then has many implicants and their number increases exponentially with the bit length of the inputs.
- We need to describe them with multilevel logic, but there is currently no method that can efficiently decompose a complicated KM into smaller minimizable blocks in acceptable time. We have to invent the decomposition ourselves.
- Division is converted to multiplication in circuits. A general divisor can be used, but its circuit is slow and takes too many resources.

For more details, see the Logic Circuits textbook, Chapter 6.

Decomposition of adder

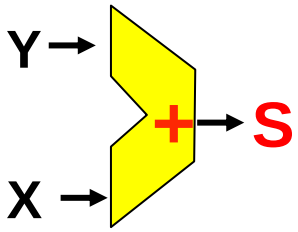


- Simplest N-bit adder
  - we chain 1-bit full adders
  - "Carry" ripple through their chain
  - Minimal number of logical elements
  - Delay is given by the last Cout - 2*(N-1)+ 3 gates of the last adder
    = (2 N+1) times propagation delay of 1 gate

Karnaugh maps for bits of adder

S_0

Y \ X	0	1
0	0	1
1	1	0



Too small groups of neighbour bits! The coverage of adder KMs has exponential complexity in gates and interconnections.

S_1

Y \ X	00	01	11	10
00	0	0	1	1
01	0	1	0	1
11	1	0	1	0
10	1	1	0	0

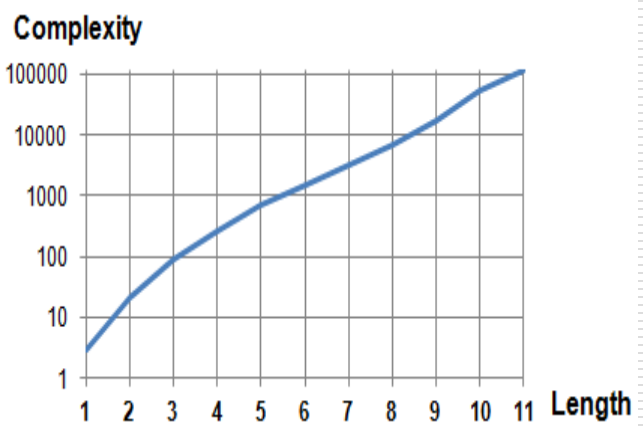
S_2

Y \ X	0 000	1 001	3 011	2 010	6 110	7 111	5 101	4 100
0 000	0	0	0	0	1	1	1	1
1 001	0	0	1	0	1	0	1	1
3 011	0	1	1	1	0	0	0	1
2 010	0	0	1	0	0	0	1	1
6 110	1	0	0	0	1	1	0	0
7 111	1	0	0	0	1	0	1	0
5 101	1	0	0	1	0	1	0	0
4 100	1	0	1	1	0	0	0	0

S=A+B

S bity $[S_{n-1} \ldots S_2\ S_1\ S_0]$

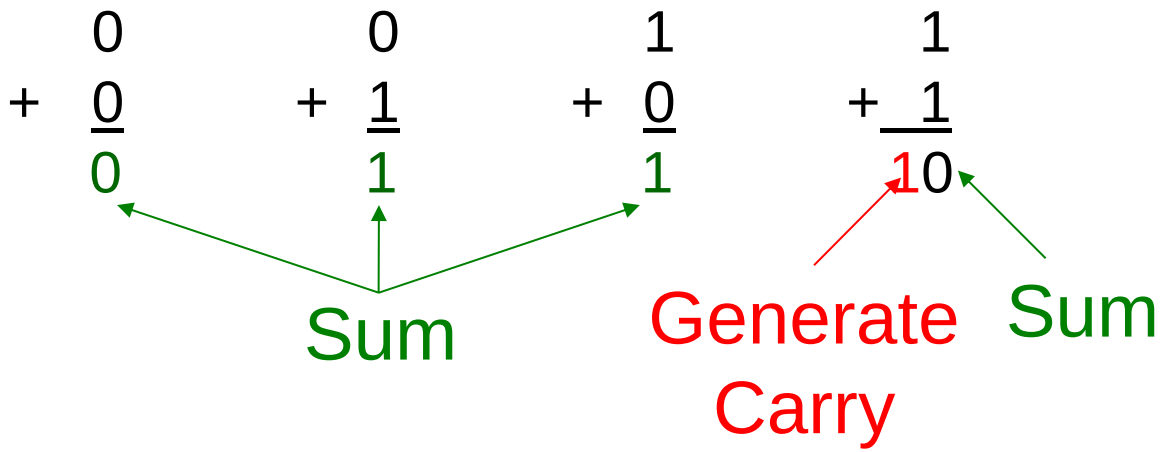
Design of adder directly from KM by Sum-Of-Products

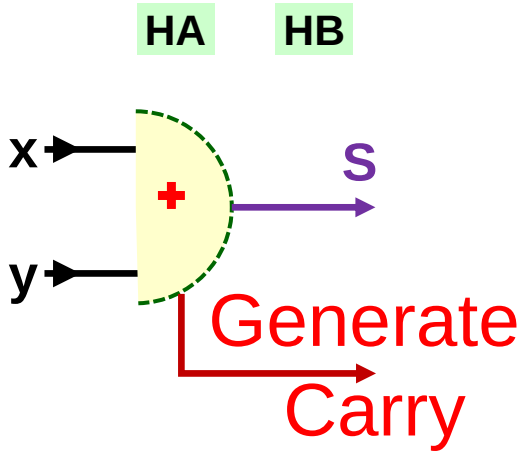


	Bit n of sum		Total operations
bit n of sum	OR operations	AND operations	bit 0 až n
0	1	2	3
1	5	14	22
2	15	52	89
3	35	148	272
4	75	380	727
5	152	688	1567
6	320	1400	3287
7	768	3072	7127
9	2304	8192	17623
10	8194	27648	53465
11	16388	46080	115933

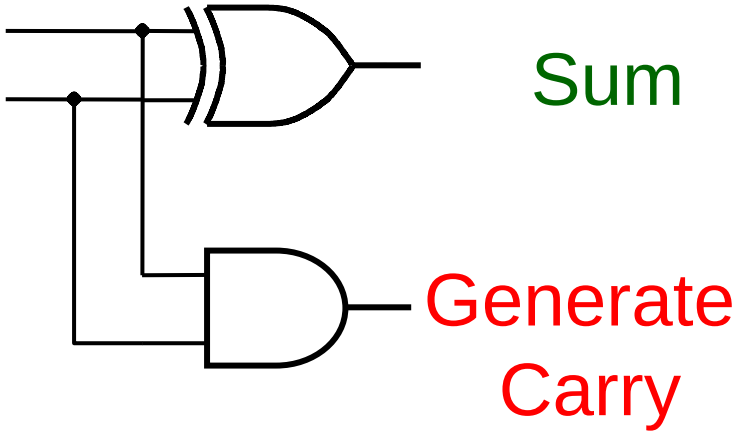
The calculation was performed by logic minimizer BOOM developed by Dept. of Computers of CTU-FEE

Half adder without Carry-In





X	Y	Sum	G
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

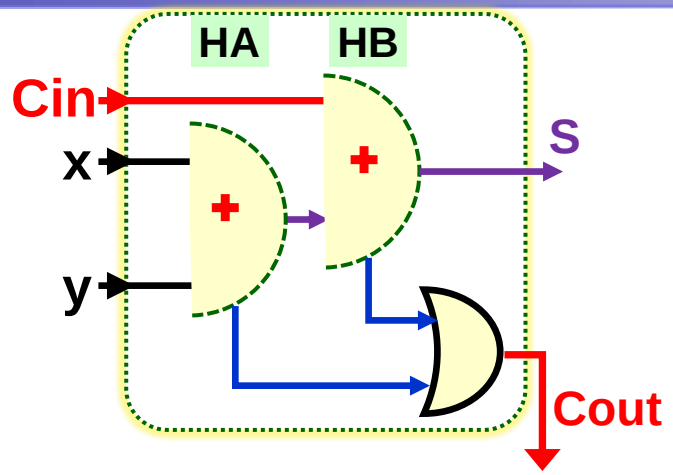


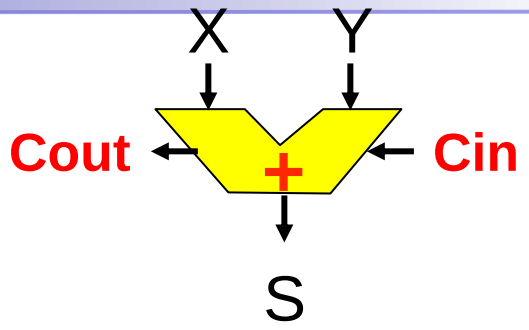
Sum <= X xor Y;

G<= X and Y

Full 1bit adder

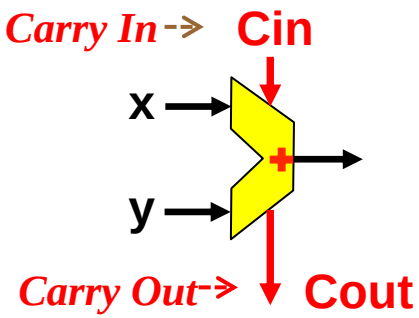
Half Adder A + Half Adder B



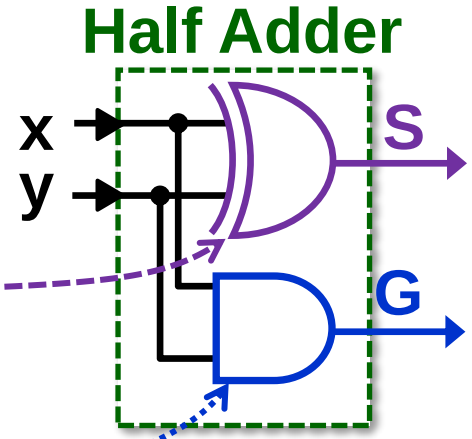


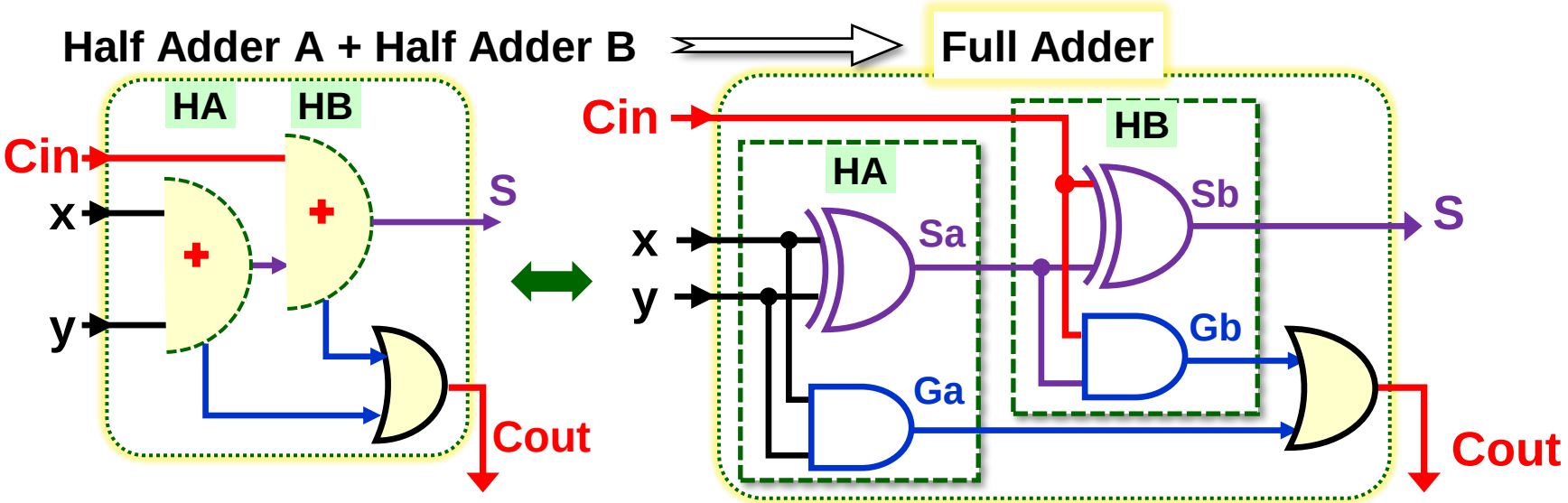
X	0	0	1	1	0	0	1	1
+Y	0	1	0	1	0	1	0	1
Sum	00	01	01	10	00	01	01	10
+ Carry-In	0	0	0	0	1	1	1	1
CarryOut Sum	00	01	01	10	01	10	10	11

1 bit adder

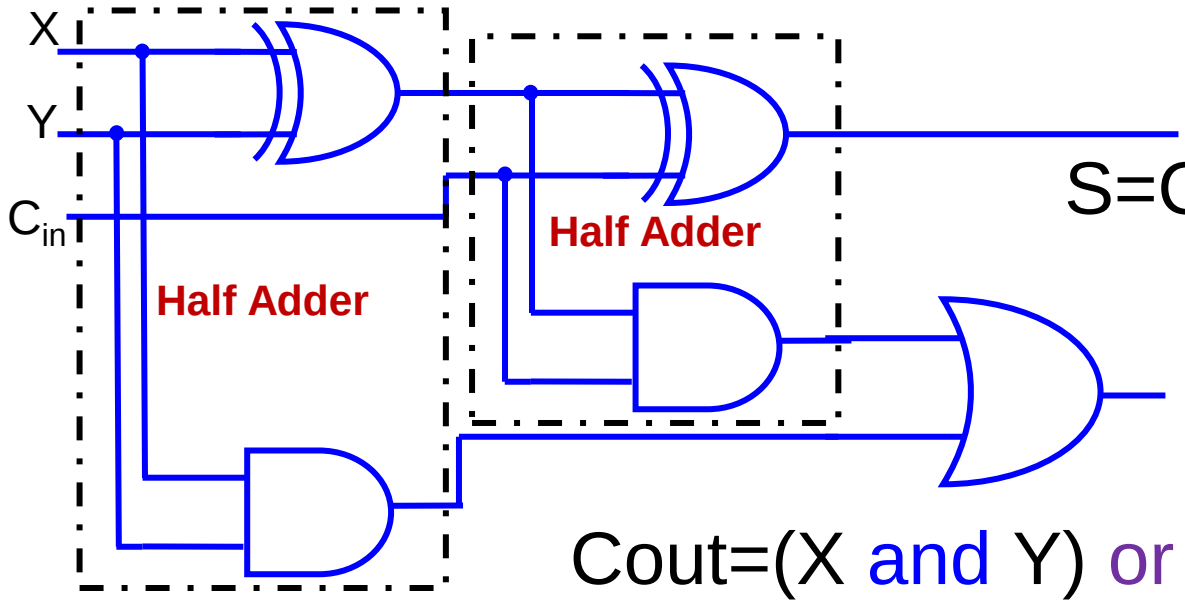


$(x+y)_{dec}$	x	y	$(x+y)_{bin}$ G	S
0	0	0	0	0
1	0	1	0	1
1	1	0	0	1
2	1	1	1	0





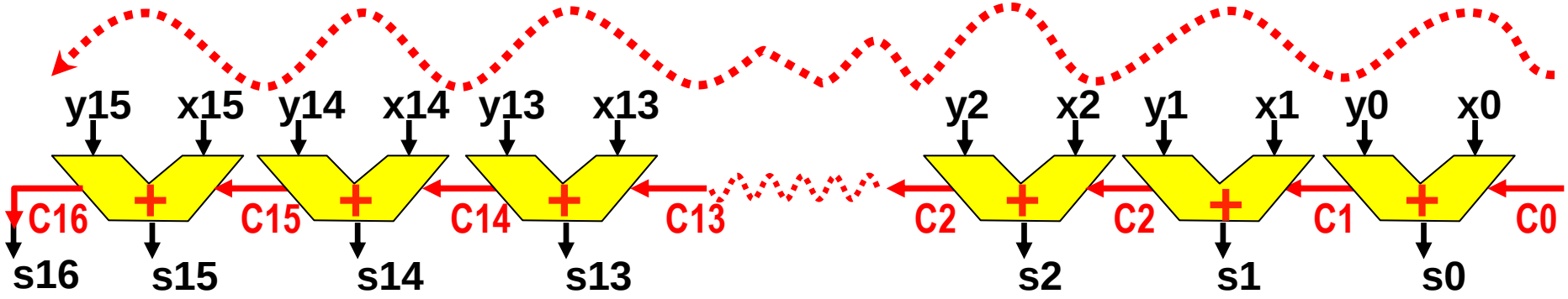
Full Adder Equations



S=Cin **xor** X **xor** Y

Cout=(X and Y) or (Cin and (X **xor** Y))

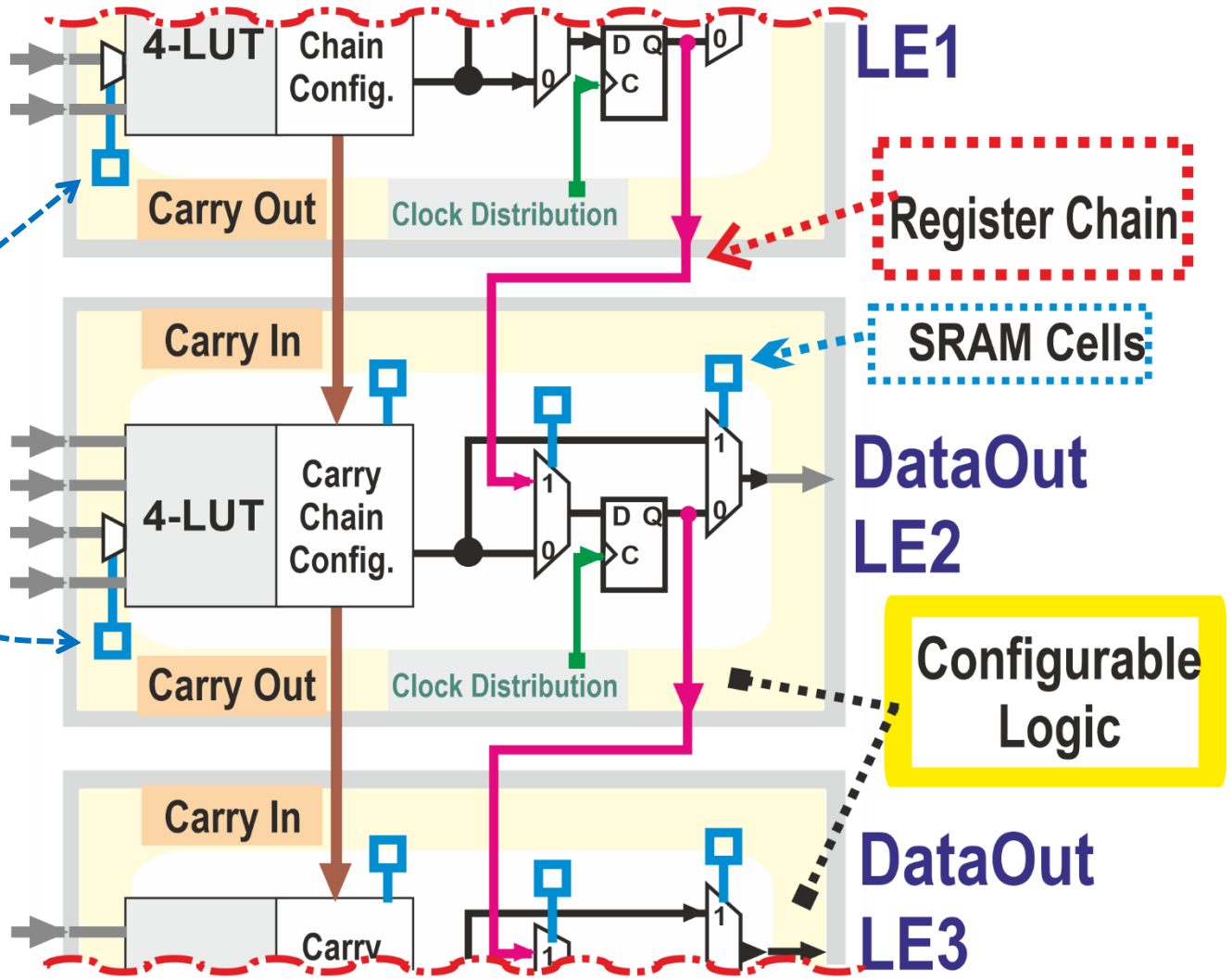
Decomposition into a chain of 1-bit adders



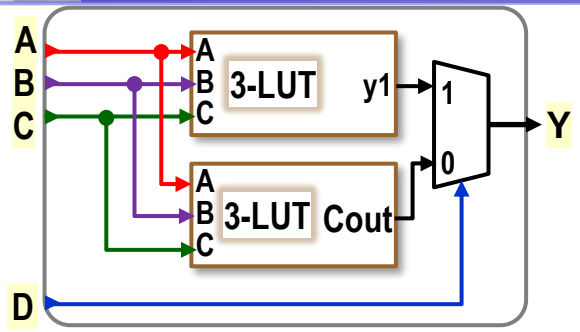
For **unsigned** addition,
C16 announces that the result overflowed,
i.e., it is out of range.
C16 has no such meaning for the addition
of **signed** numbers.
See Binary Prerequisite, page 11 and 19.

Ripple support in Cyclone II and IV Logical Elements (LEs)

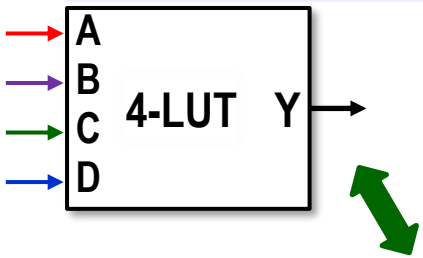
For configurations
Carry Chain
nsmission propagation)
will disconnect
C 4-LUT input
and it's being run
Carry In.



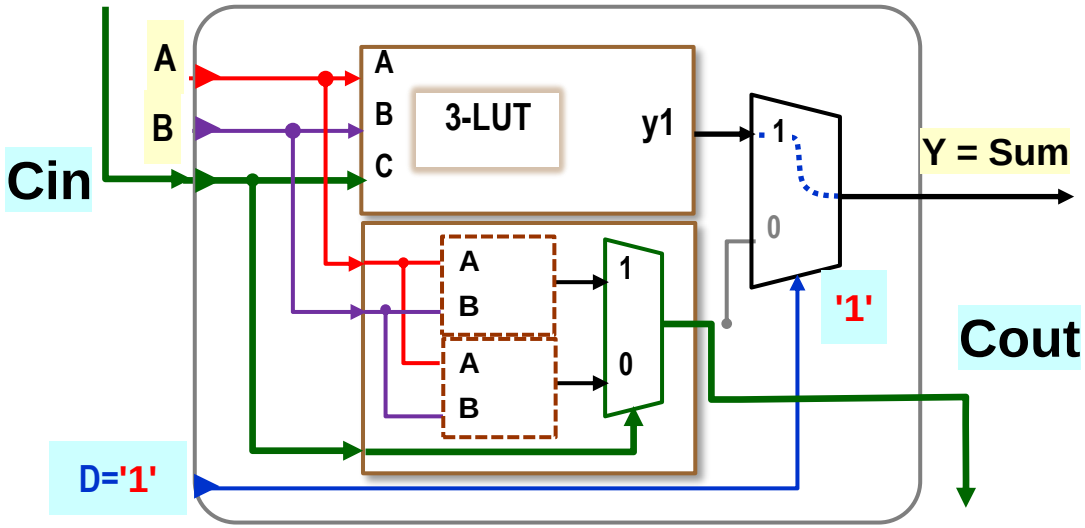
FPGA Carry Chain for "Ripple"

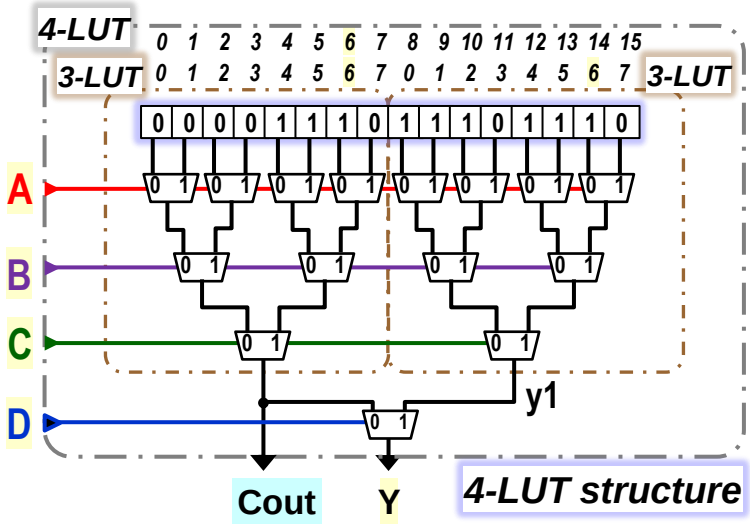


Primary Configuration 1 × 4-LUT

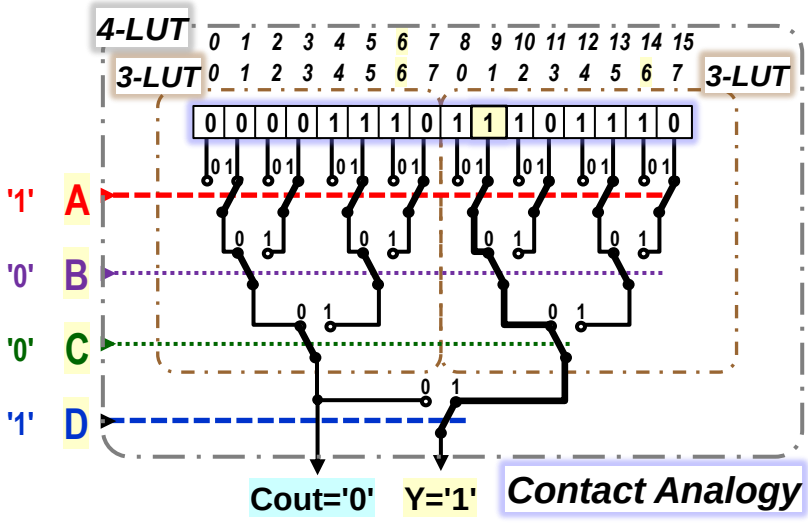


Carry Chain Configuration 2 × 3-LUT



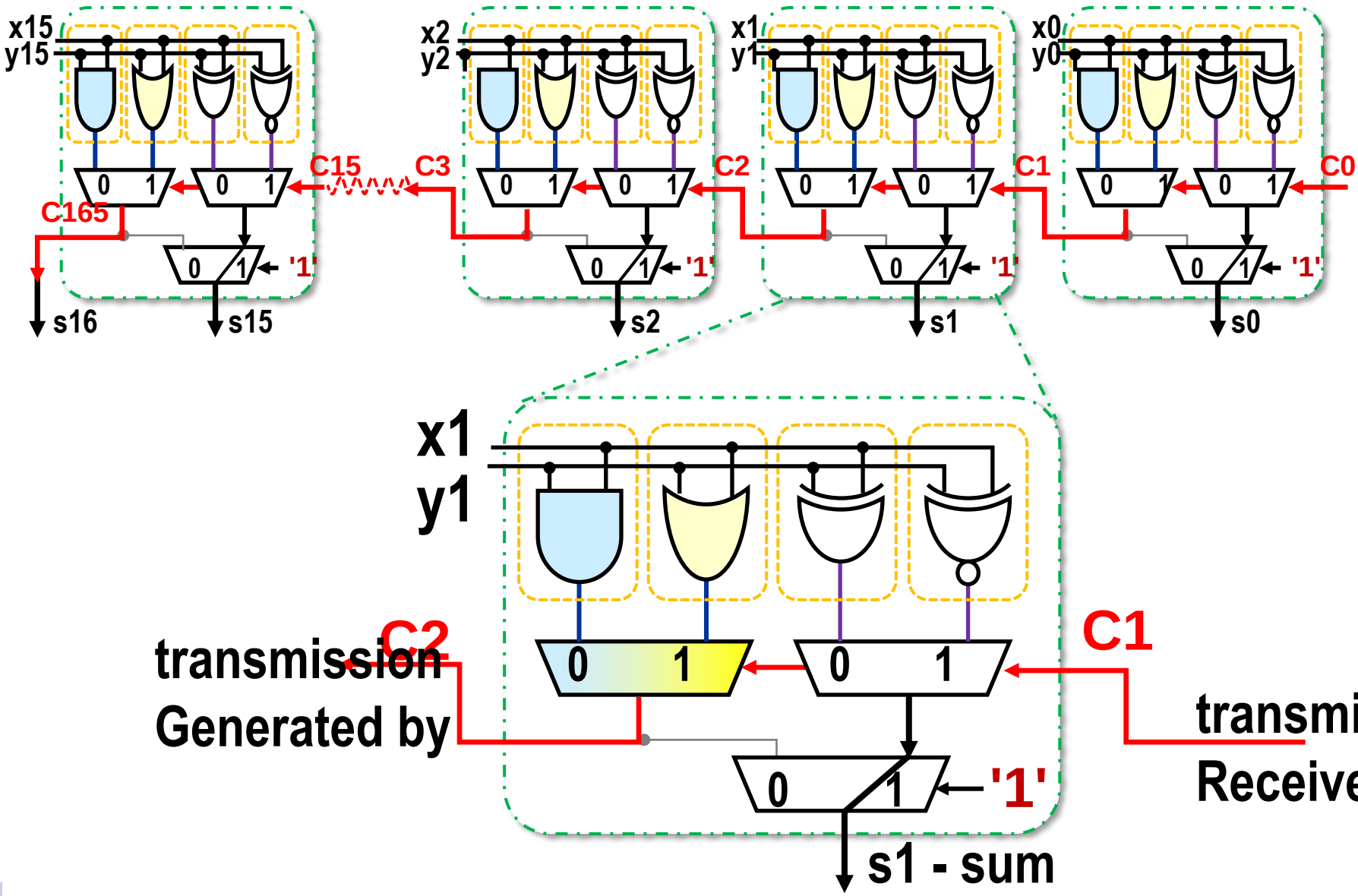


4-LUT structure

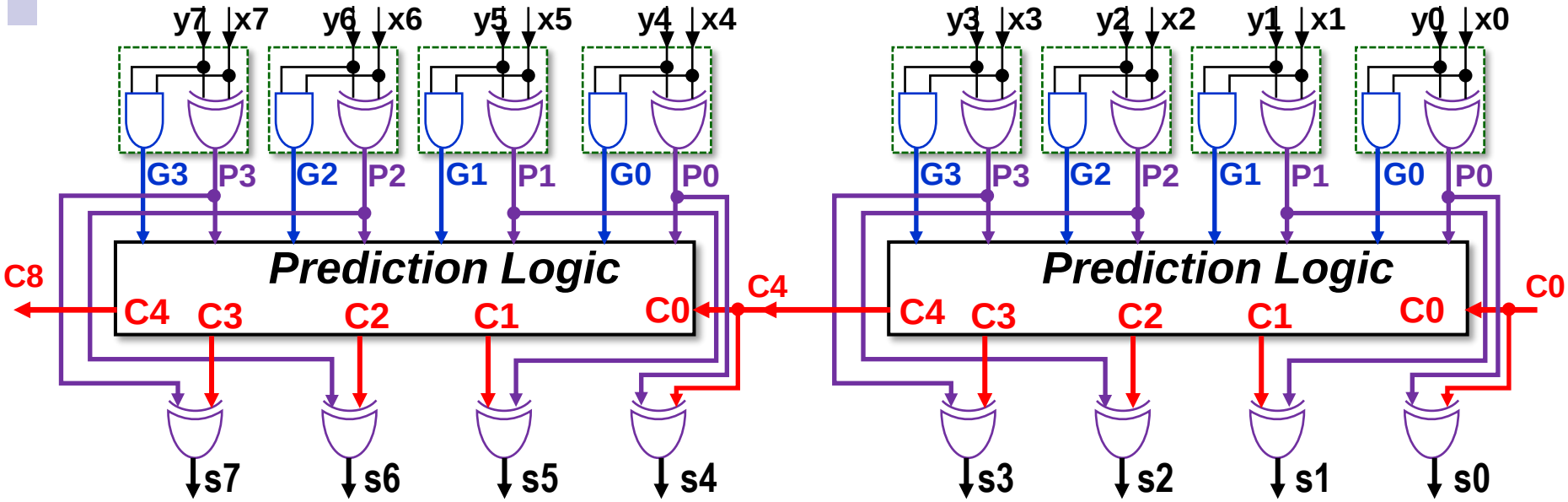


Contact Analogy

Ripple Adder in FPGA



CLA (Carry-lookahead Adder)



Transfer equations:

C1 = G0+P0.C0;

C2 = G1+P1.(G0+P0.C0) =G1+P1. G0+P1.P0.C0;

C3 = G2+P2.(G1+P1.(G0+P0.C0))=G2+P2.G1+P2.P1. G0+P2.P1.P0.C0;

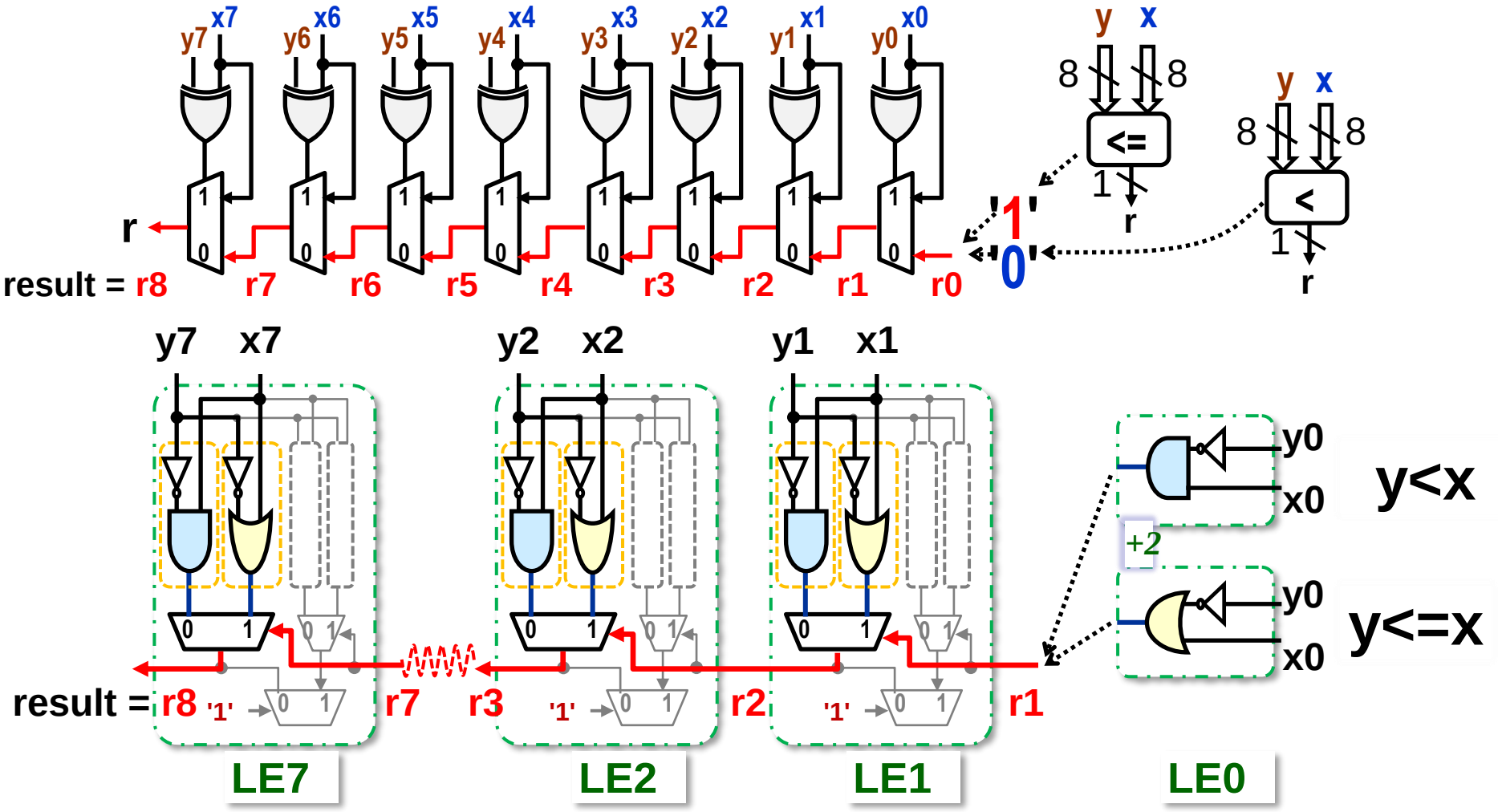
C4 = G3+P3.(G2+P2.(G1+P1.(G0+P0.C0))

= G3+P3.G2 + P3.P2.G1 + P3.P2.P1.G0 + P3.P2.P1.P0.C0;

etc.

They can be very quickly computed even over 128 bits on a tree structure, as is done by Kogge-Stone Adder. It requires about 50% more area than RCA.

Ripple structure has also comparator x>=y

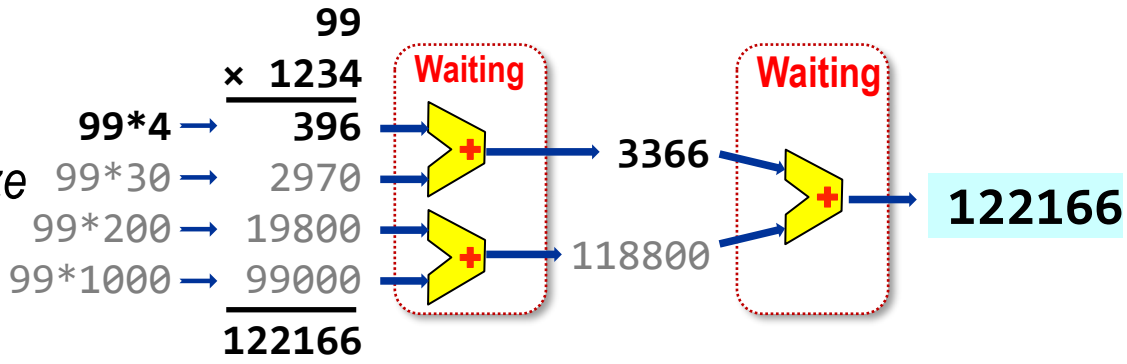


RCA adders are preferred in FPGAs,
as their transfer chain can be modified to other operations.

Wallace Tree Multiplier

The count is slow

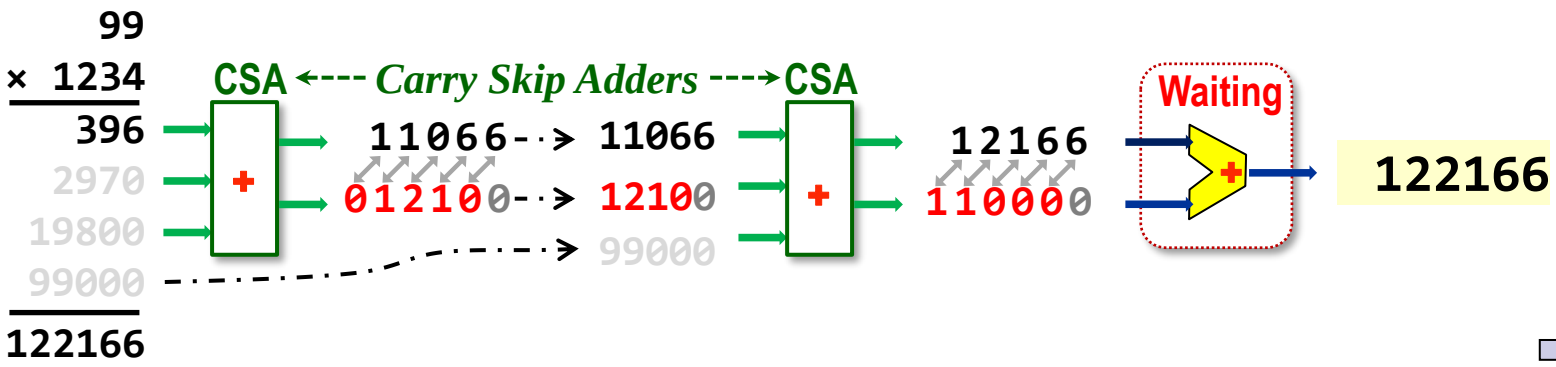
waiting for transmissions to stabilize all intermediate results



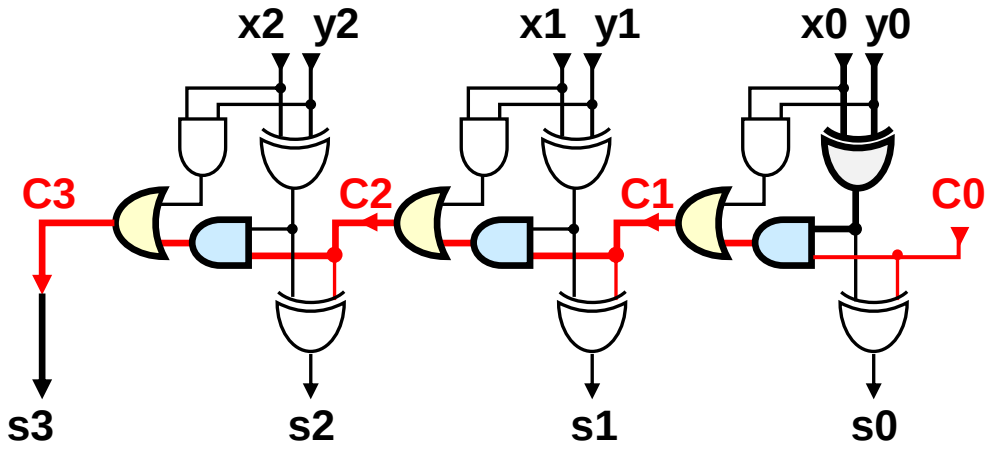
CSA (Carry Skip Adder): adds in orders of magnitude - reduces 3 input addends to 2 output values

```
  396 =                    300 + 90 + 6
 2970 =           2000 + 900 + 70 + 0          11066
+19800 = 10000 + 9000 + 800                   +12100
-------  ------  ------  ---  ---  ---       -------
 23166   10000  +11000  +200 +160 +06   →     23166
```

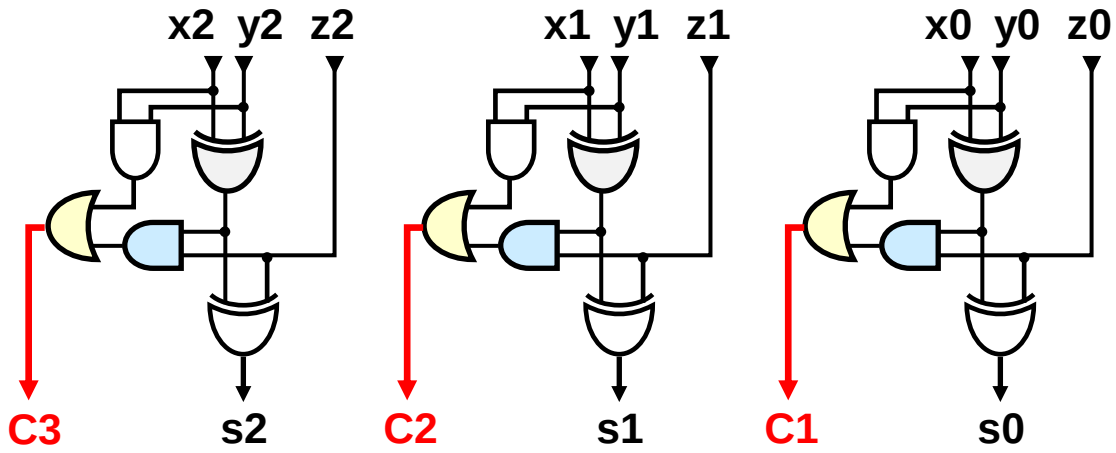
Fast multiplier reduces subtotals. One CSA only adds them at the end.



CSA is simple



RCA - Ripple Carry Adder



CSA - Carry-skip Adder