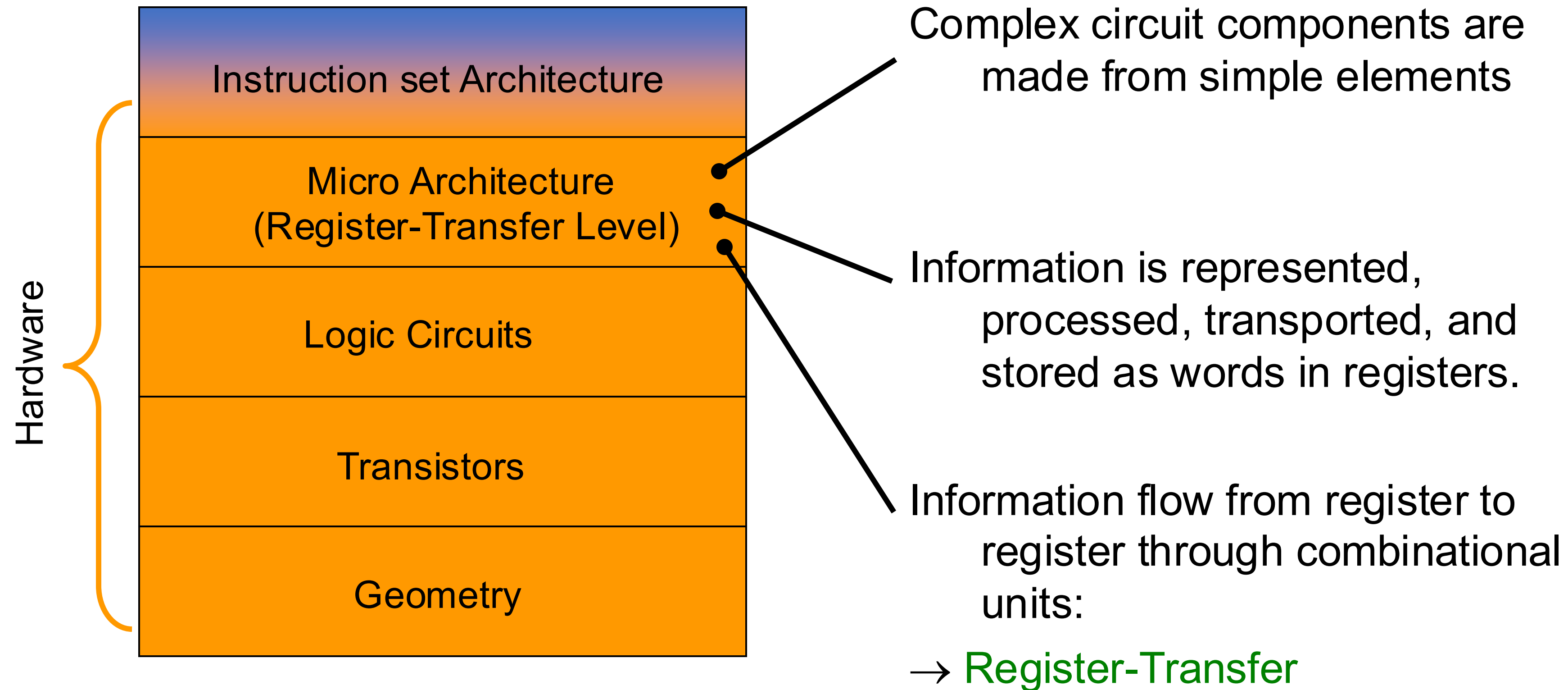
Herbert Wertheim College of Engineering
UNIVERSITY *of* FLORIDA

Digital Logic And Computing Systems

Lecture 06 – Automata – Controlpath - Continued

Dr. Christophe Bobda
EEL3701C Fall 2025

Register-Transfer Level (RTL)

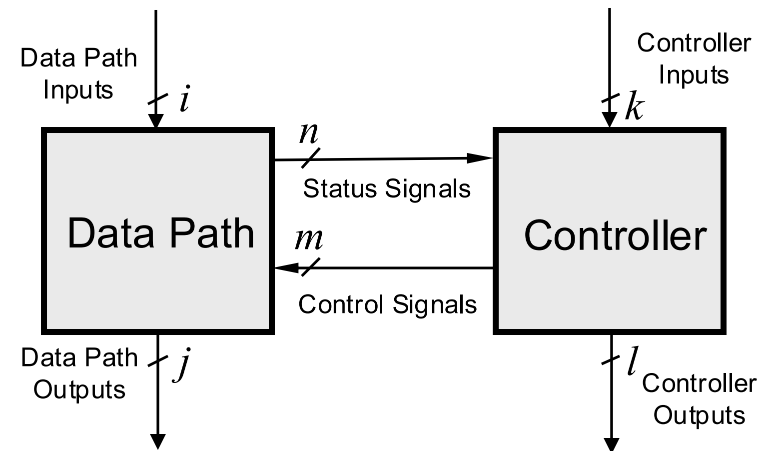


Agenda

- Basic Structure
- Controller Implementation
- Microprograming
- RTL Design Flow
- Design Example

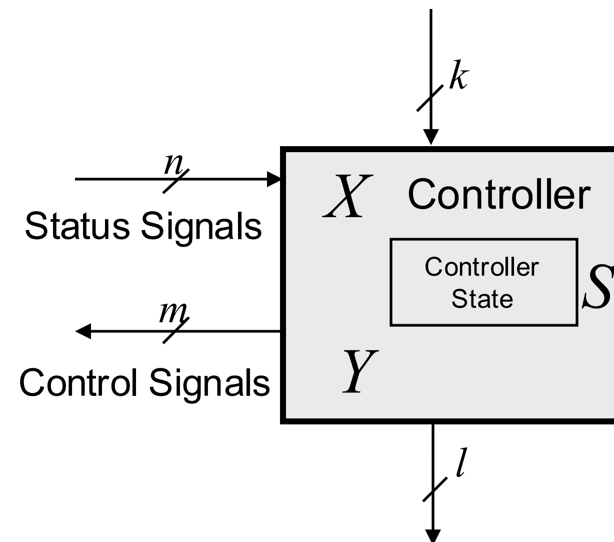
Basic Structure: Data Path+Controller

- Complex digital systems usually consist of two parts
  - A data path (ALU) performs computation on data
  - A control path (control unit, controller) manages the system
    - Data path and control path are connected through status and control signals
    - Data path and control path consists of sequential and combinational components



Controller Implementation

- Controllers are modelled using automata theory
  - Input variables X: k control inputs and n status signals
  - Output variables Y: l control outputs and m control signals
  - State variables S: control states.
  - State transition function $\delta(x,s)$
  - Output function $\mu(s)$ (usually modelled after Moore)



Controller Implementation

- Implementation Options
  - “Hard-wired control“
    - Automata design discussed in chapter 6, i.e. $\delta(x,s)$ and $\mu(s)$ are realized as combinational logic
    - Advantage: Fast
    - Drawback: Not Flexible
  - “Micro-programmable control“
    - $\delta(x,s)$ and $\mu(s)$ are stored in memory
    - Automata behavior can be changed through memory override
    - Advantage: Flexible
    - Drawback: Slow

RTL Design Flow

- Systematic mapping of (sequential) algorithms in a digital circuit

Multiplication Algorithm:

1. Load operand registers X, Y
2. $C \leftarrow 0$, $P \leftarrow 0$
3. repeat n-times:

   if $Y(0)=1$ then

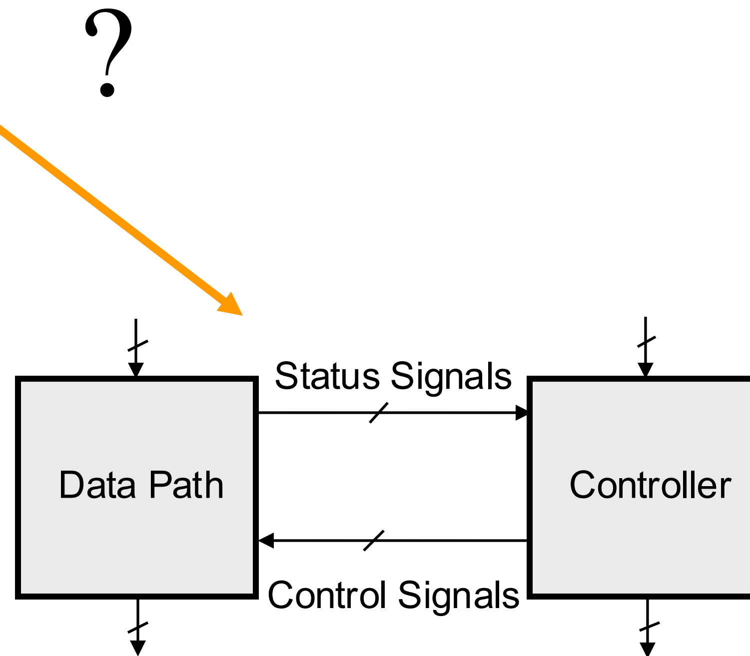
   $P \leftarrow P+X$

   else

   $P \leftarrow P+0$

   shift (C,P,Y) right
4. Result is in (P,Y)

?

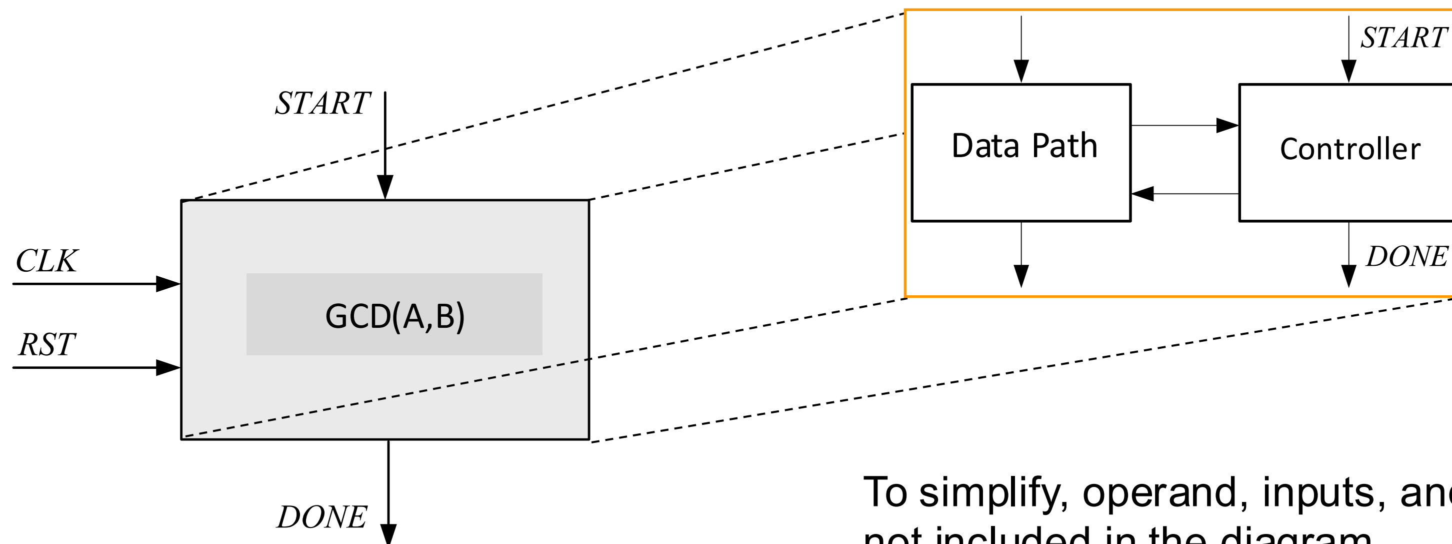


RTL Design Flow

- Possible Approach
  - Data Path
    - Identify the components of the circuits (Objects)
    - Define the connections among the various units
    - Define operations : for each method 1 control signal
    - Define conditions: for each status signal 1 status signal
  - Controller
    - Define 1 state for each sequential step
    - For each state: define next state and control signals
    - Implement controller “hardwired” or microprogrammed
  - In practice, we use computer aided design (CAD) tools (HDL)
    - Modelling in textual and graphical language
    - Model simulation
    - (Semi) automatic design

Design Example I – gcd(a,b)

- Greatest Common Divisor – gcd(a,b)
  - A circuit to compute the gcd of two 8-bit unsigned integer A and B
  - Interface signals: *START, DONE, CLK, RST*
  - Behavior:
    - Processing begins when $START=1$
    - Computation completed when $DONE=1$



To simplify, operand, inputs, and result output not included in the diagram

gcd(a,b) - Algorithm

Idea for the gcd(a,b) - algorithm:

$$gcd(a,b) = a, \text{ if } a = b$$
$$= gcd(a-b, b), \text{ if } a > b$$
$$= gcd(a, b-a), \text{ if } a < b$$

```
done = FALSE;
repeat
  if (a>b)
       a = a-b;
  elseif (b>a)
       b = b-a;
  else done = TRUE;
until done;
```

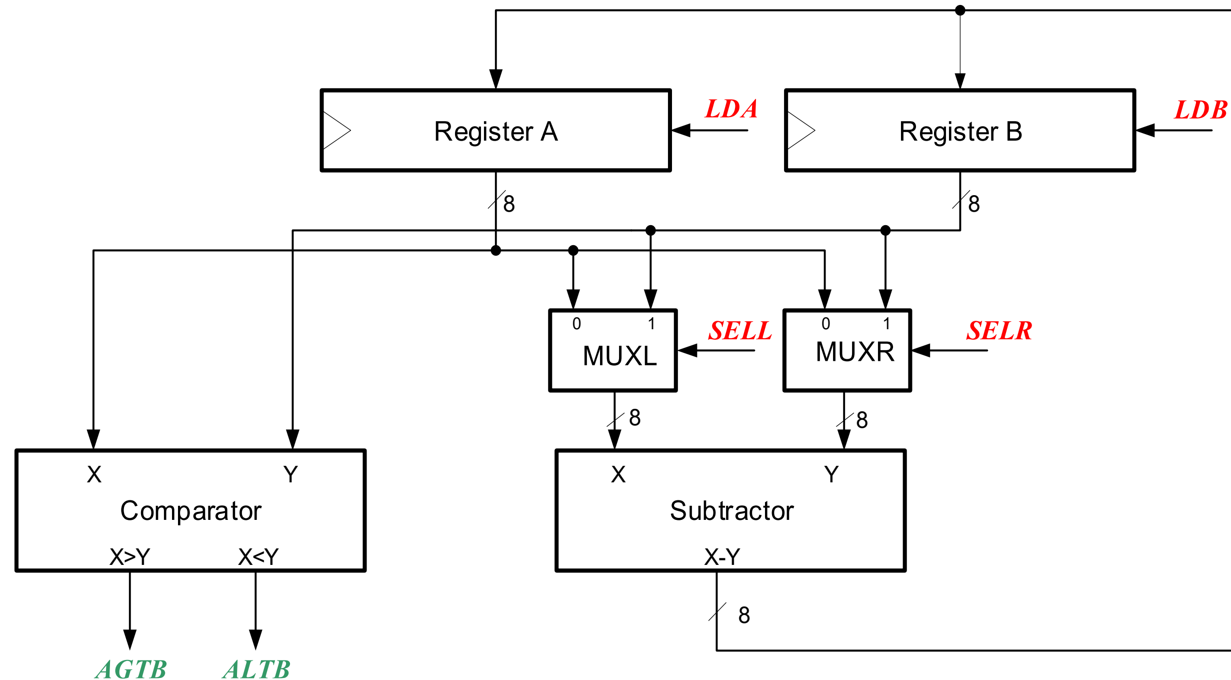
What components are needed for the data path ?

- Register for a and b
- Comparator: to compute a>b, a<b, a=b
- Subtractor
- Multiplexer: to “permutate" a and b, and be able to compute a-b and b-a with only one subtractor

gcd(a,b) – Data Path

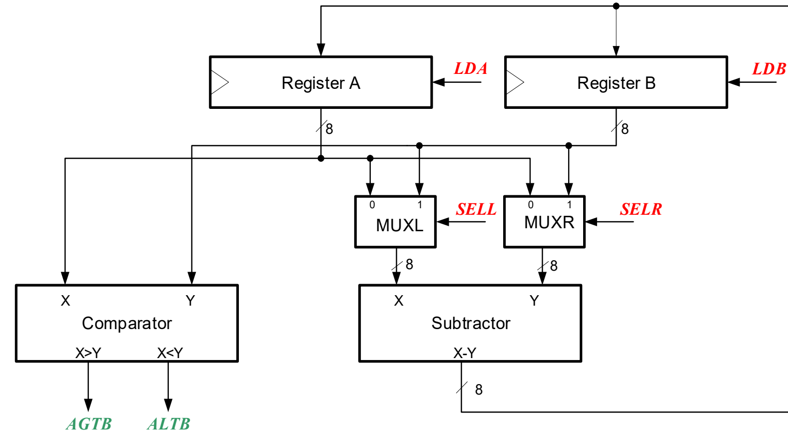
- Control signal (operations): *LDA* (load register A), *LDB* (load register B), *SELL* (select left MUX), *SELR* (select right MUX)
- Status signals (conditions): *AGTB* (A greater than B), *ALTB* (A less than B)

```
done = FALSE;
repeat
  if (a>b)
       a = a-b;
  elseif (b>a)
       b = b-a;
  else done = TRUE;
until done;
```

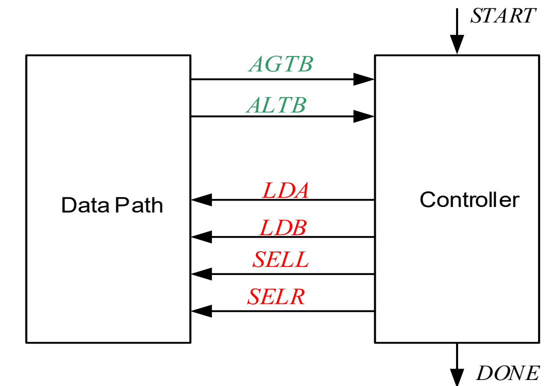


gcd(a,b) – Control Path

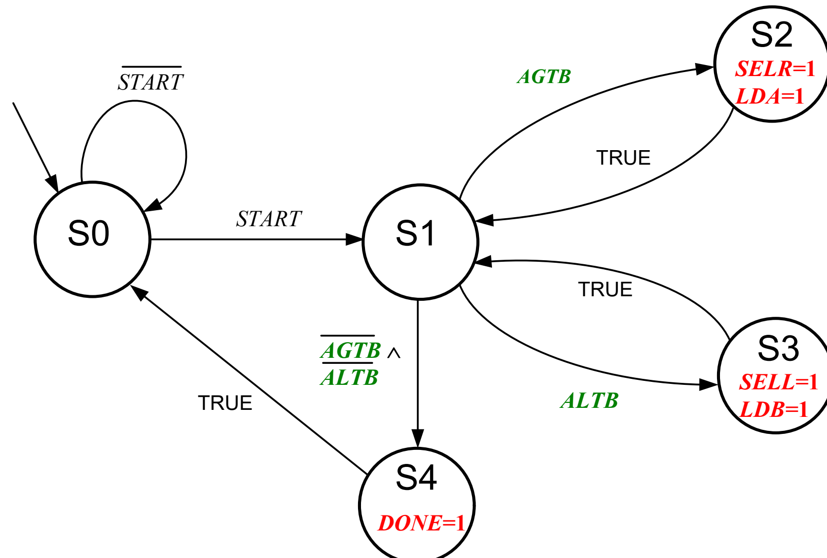
```
done = FALSE;
repeat
  if (a>b)
       a = a-b;
  elseif (b>a)
       b = b-a;
  else done = TRUE;
until done;
```



Interface Definition



State Transition Diagram

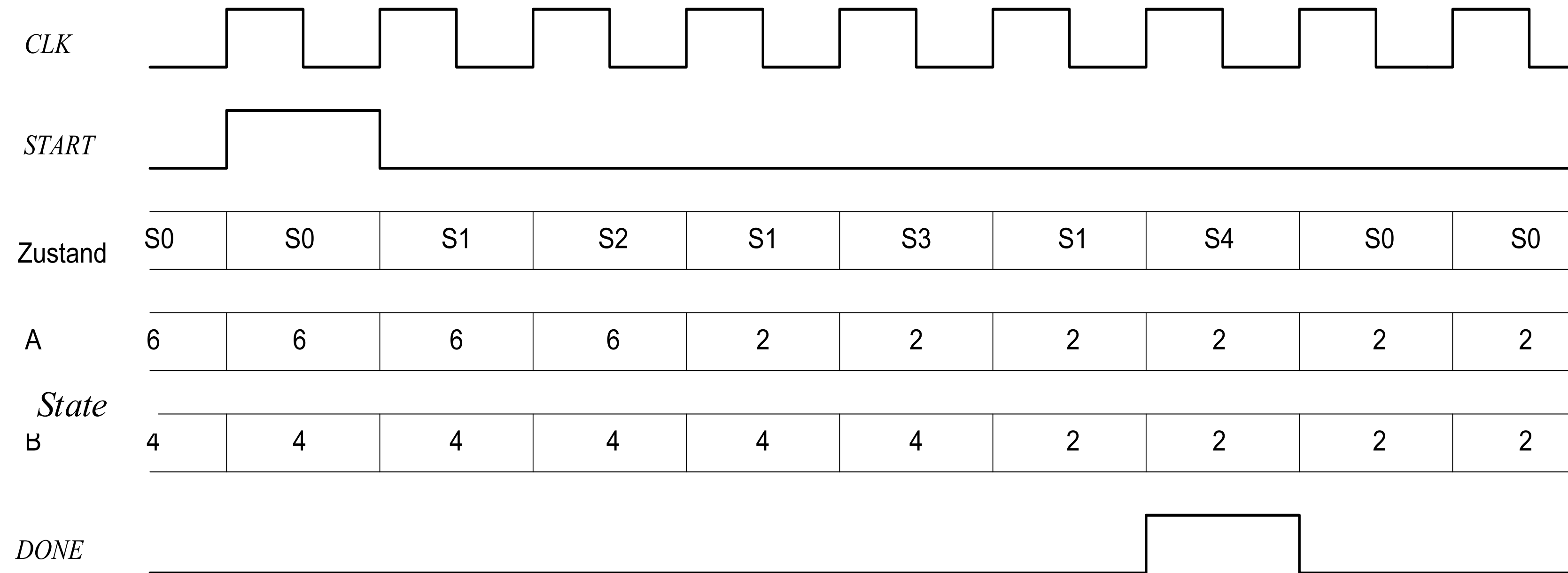


Remarks:

– only outputs that are set to 1 are shown in the diagram

– Optimization:
$SELR = \overline{SELL}$ → save 1 signal

gcd(a,b) - Timing



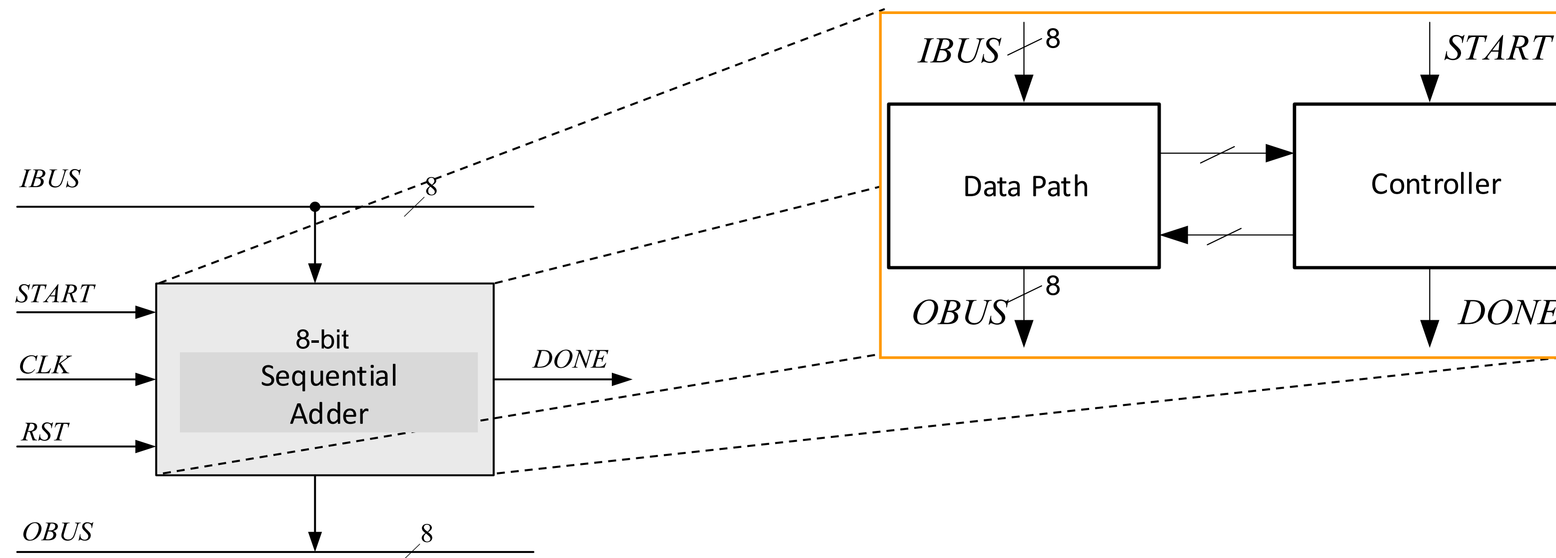
Assumption:

- A=6, B=4
- Use of positive edge-triggered D-FF

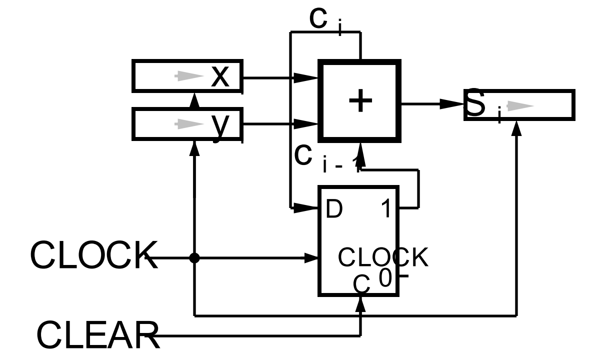
After setting control signals, the controller always waits 1 clock cycle for the correct status signals to be available

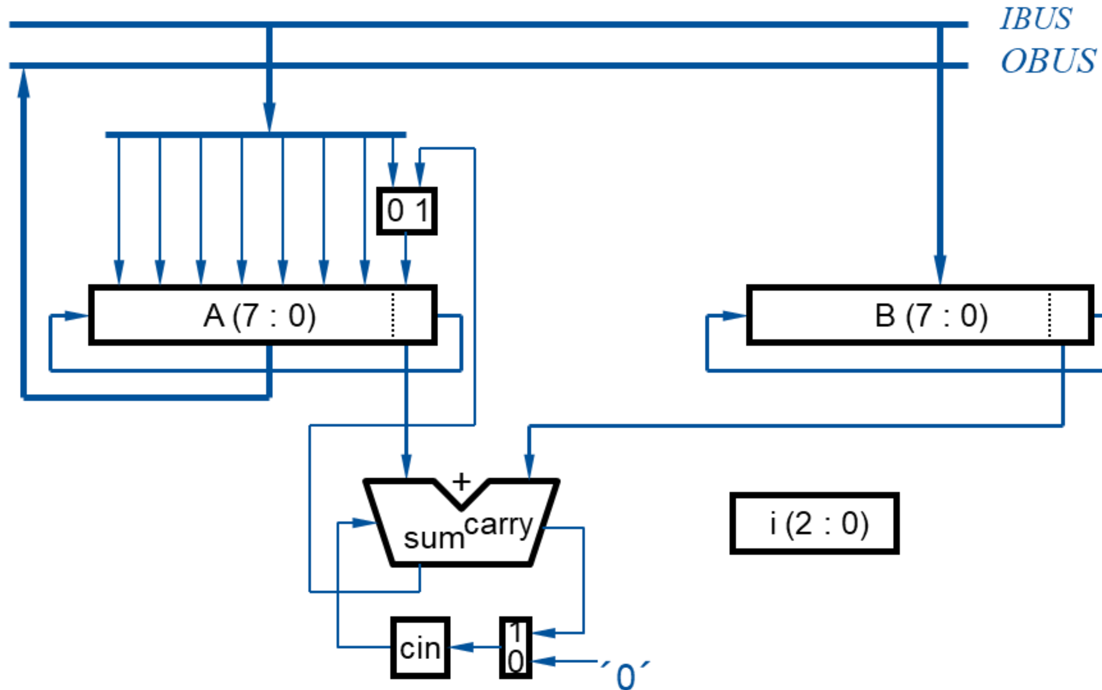
Design Example II - SeqAdd(a,b)

- Design of an 8-bit sequential adder for unsigned integer numbers: $C=A+B$
  - Interface signals: *IBUS, OBUS, START, DONE, CLK, RST*
  - Behavior:
    - $START=1$ start of the addition. A and B are placed on the input bus, *IBUS*, sequentially, after each clock cycle.
    - Operation completed: $DONE=1$, result is placed on the output bus, *OBUS*



SeqAdd(a,b) – Algorithm, Data Path





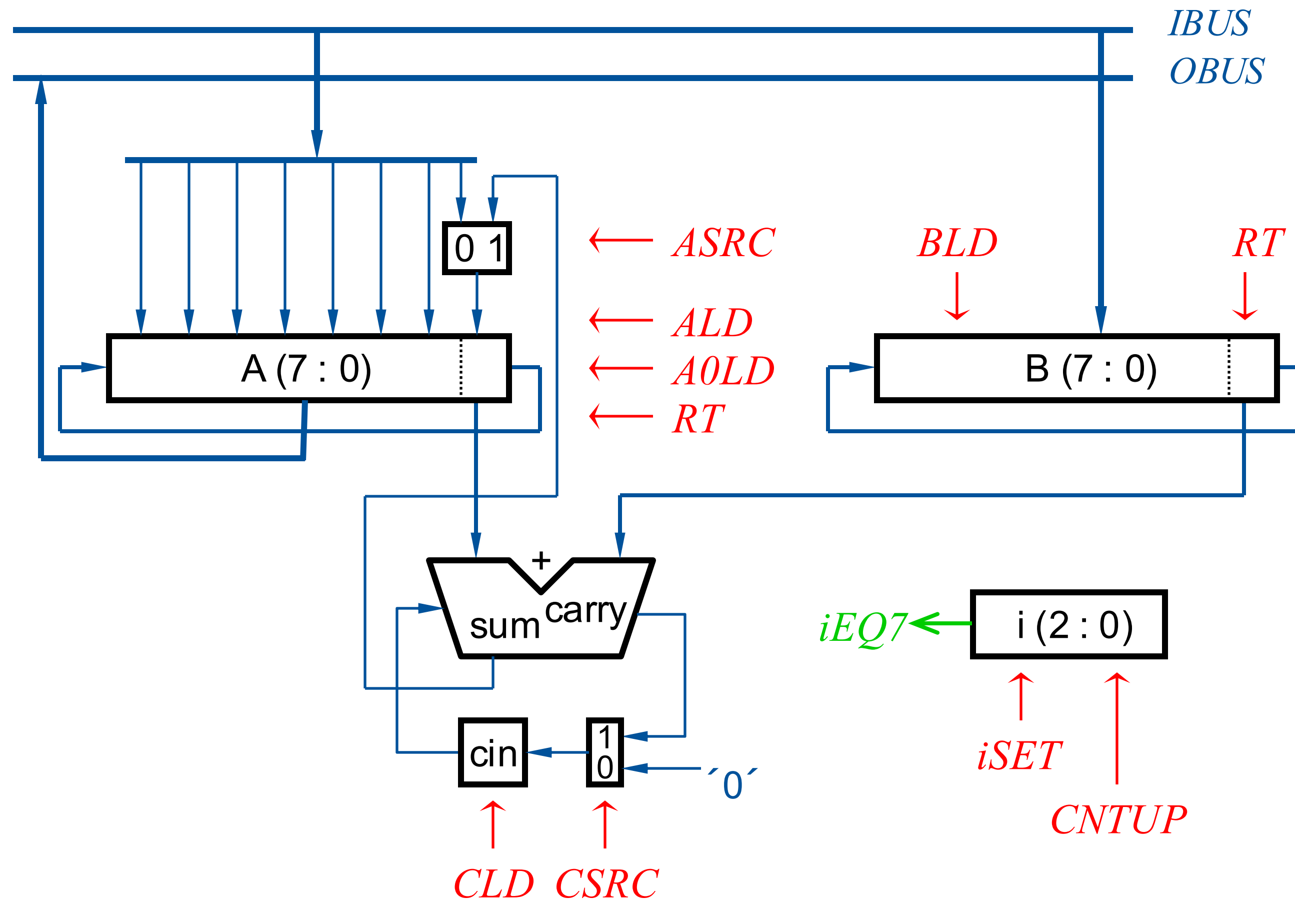
Algorithms

1. Wait until $START=1$
2. $A \leftarrow IBUS,\ Cin \leftarrow 0$
3. $B \leftarrow IBUS$
4. for i=0..7:
5. $A(0) \leftarrow$ sum($A(0),B(0),Cin$), $Cin \leftarrow$ carry($A(0),B(0),Cin$)
6. rotate A and B left
7. Result is in A and on $OBUS$

SeqAdd(a,b) – Data Path-Components

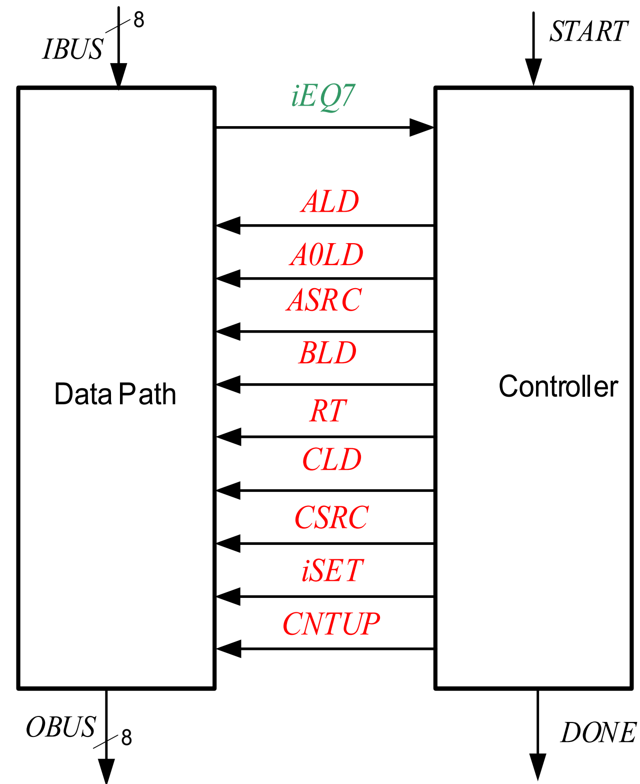
Components		Control Signals/Status Signals
Register A:	8-bit Register	
	parallel load A(7:0)	*ALD*
	load A(0)	*A0LD*
	rotate left A(7:0)	*RT*
MUX:	2-times, 1-bit Multiplexer	*ASRC*
Register B:	8-bit Register	
	parallel load B(7:0)	*BLD*
	rotate right B(7:0)	*RT*
+:	Full Adder	
Cin:	1-bit Register	
	load Cin	*CLD*
MUX:	2-times, 1-bit Multiplexer	*CSRC*
i:	Loop counter (3-bit Register, mod-8 counter)	
	set i(2:0) to 0	*iSET*
	increment i(2:0)	*CNTUP*
	loop counter = 7	*iEQ7*

SeqAdd(a,b) – Control/Status Signals

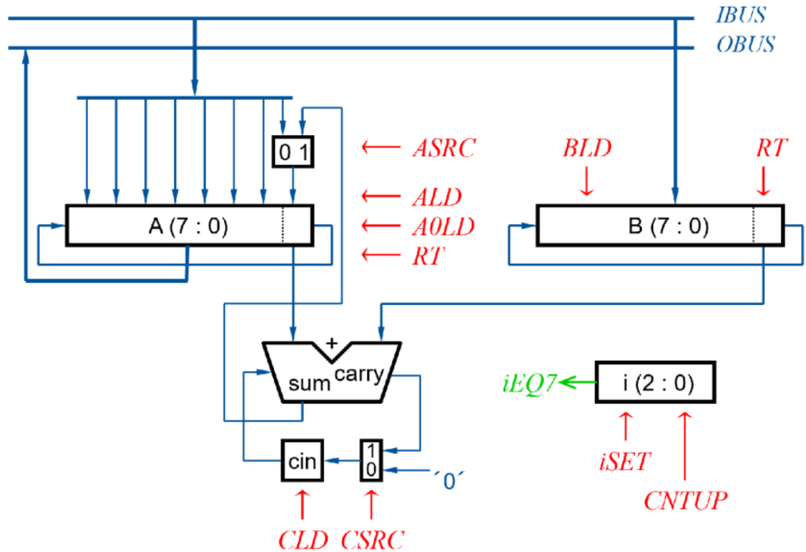


SeqAdd(a,b) - Controller

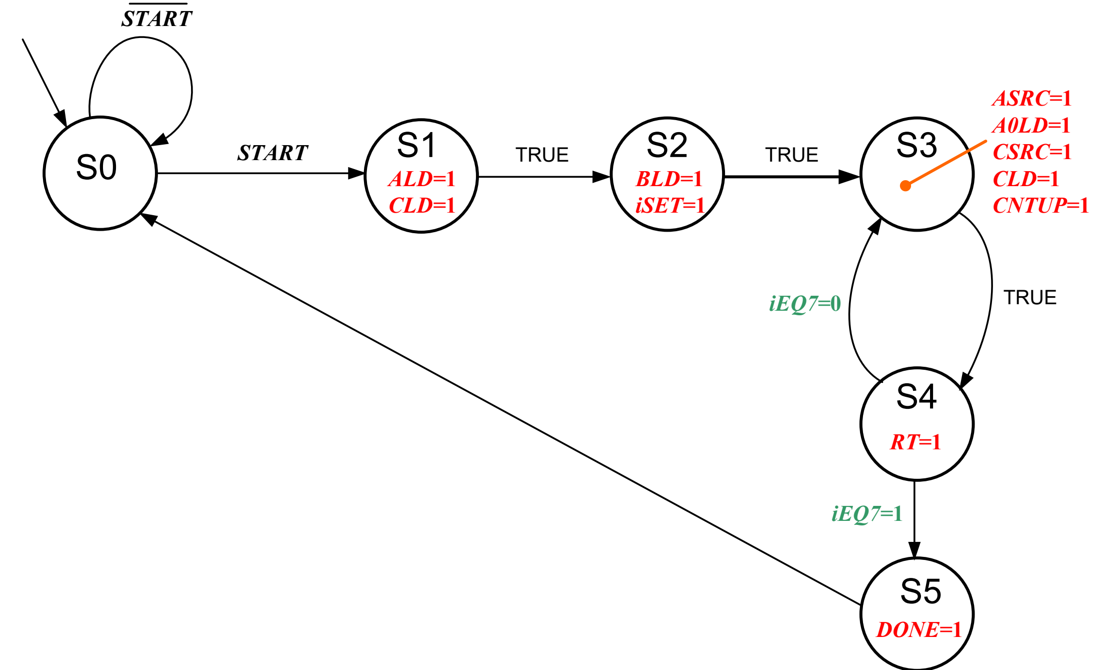
- Interface Definition:



SeqAdd(a,b) - Controller



State Transition:



<u>Remarks</u>

– only outputs that are set to 1 are shown in the diagram

– Optimization:

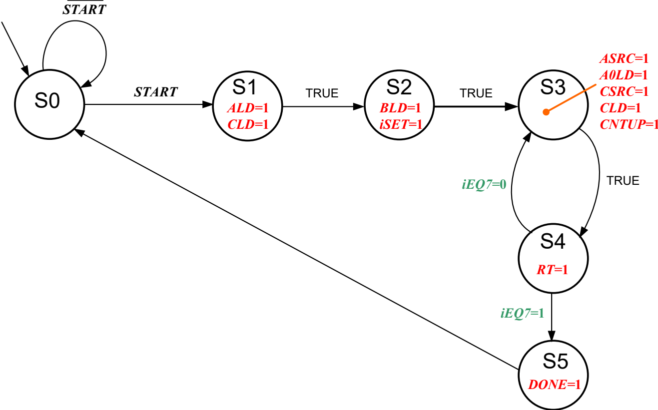
$A0LD=CSRC=CNTUP=ASRC$

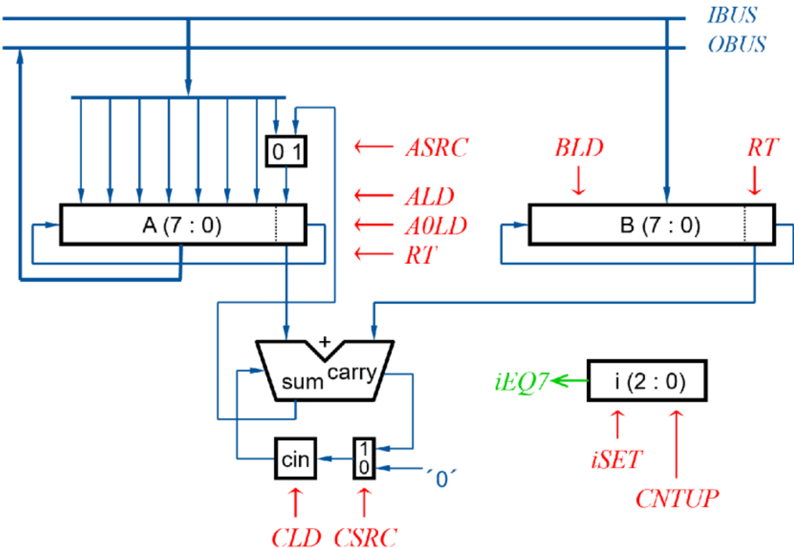
$iSET=BLD$

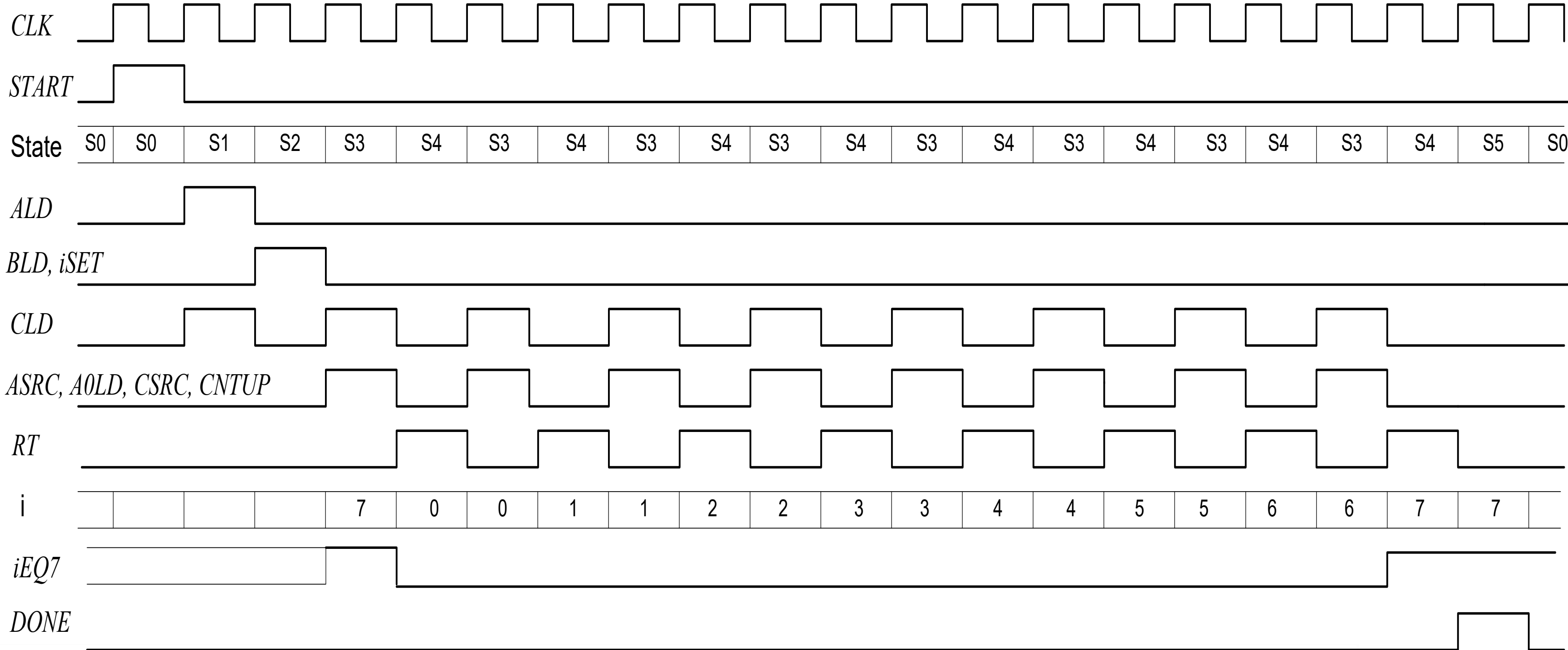
$CLD=ASRC+ALD$

$\rightarrow$ The number of control signals can be reduced to 4

SeqAdd(a,b) - Timing



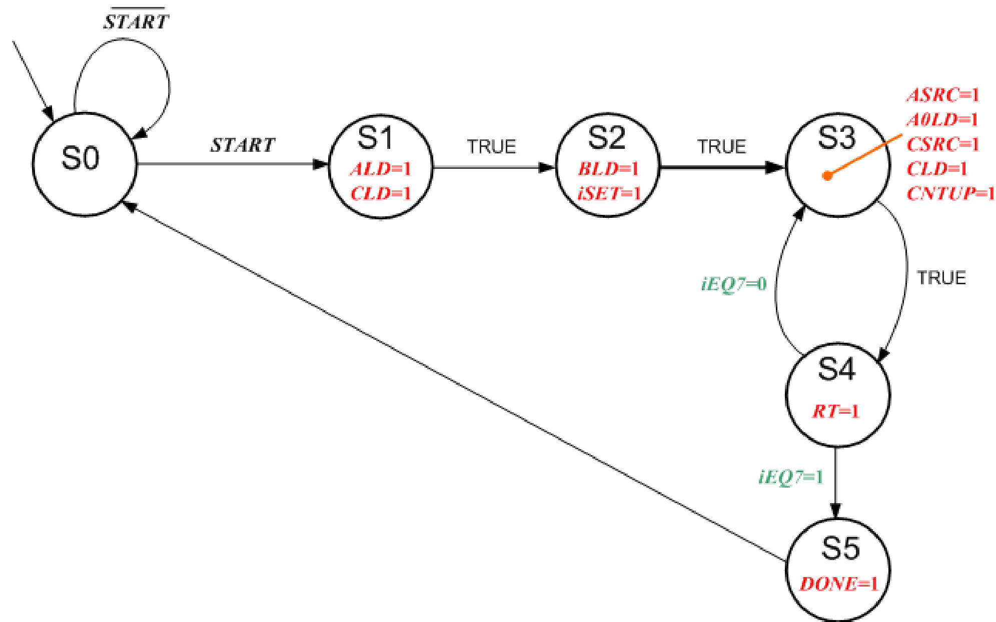




SeqAdd(a,b) - Controller

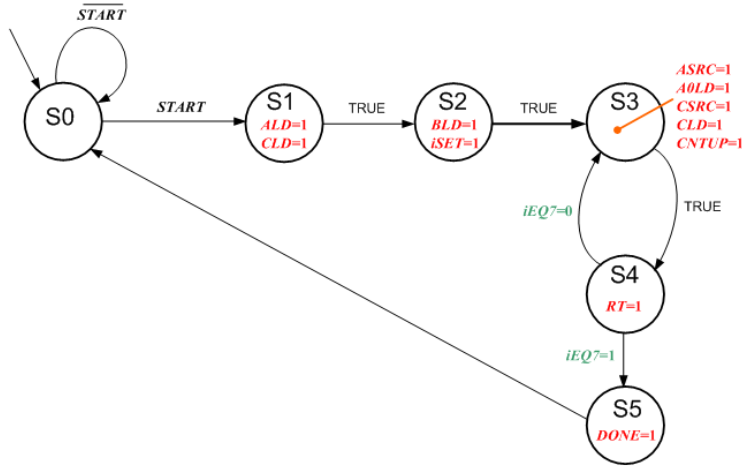
Transition Table

S0 = "000", S1 = "001", S2 = "010", S3 = "011", S4 = "100", S5 = "101"



S	ALD	ASRC	BLD	RT	DONE	(START, iEQ7) 0,0	0,1	1,0	1,1
000	0	0	0	0	0	000	000	001	001
001	1	0	0	0	0	010	010	010	010
010	0	0	1	0	0	011	011	011	011
011	0	1	0	0	0	100	100	100	100
100	0	0	0	1	0	011	101	011	101
101	0	0	0	0	1	000	000	000	000

SeqAdd(a,b) - Controller



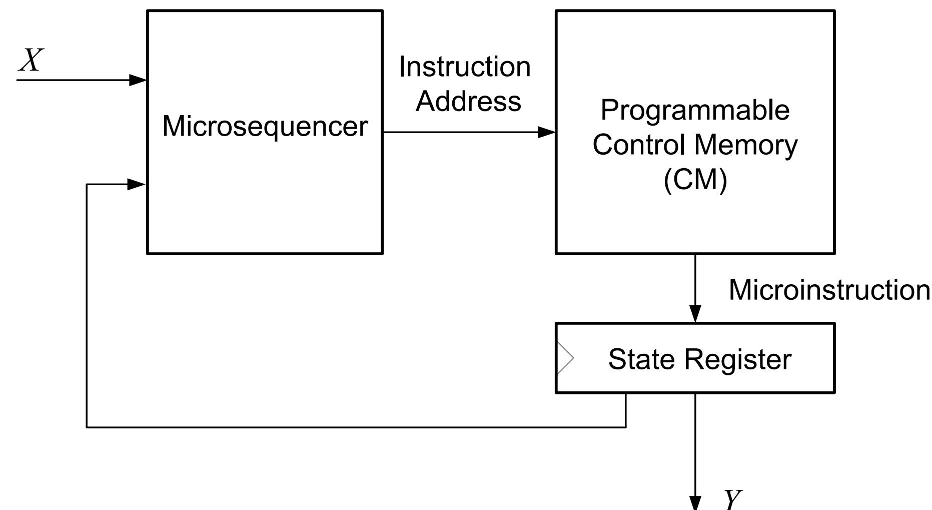
Transition Table

S0 = "000", S1 = "001", S2 = "010", S3 = "011", S4 = "100", S5 = "101"

S	ALD	ASRC	BLD	RT	DONE	(START, iEQ7) 0,0	0,1	1,0	1,1
000	0	0	0	0	0	000	000	001	001
001	1	0	0	0	0	010	010	010	010
010	0	0	1	0	0	011	011	011	011
011	0	1	0	0	0	100	100	100	100
100	0	0	0	1	0	011	101	011	101
101	0	0	0	0	1	000	000	000	000

Microprogramming

- Basic Idea
  - Current state and outputs are defined using microinstructions
  - The microinstructions are stored in a microprogram memory (Control Memory, CM)
  - A microsequencer computes the address of the next microinstruction using the inputs and the current state

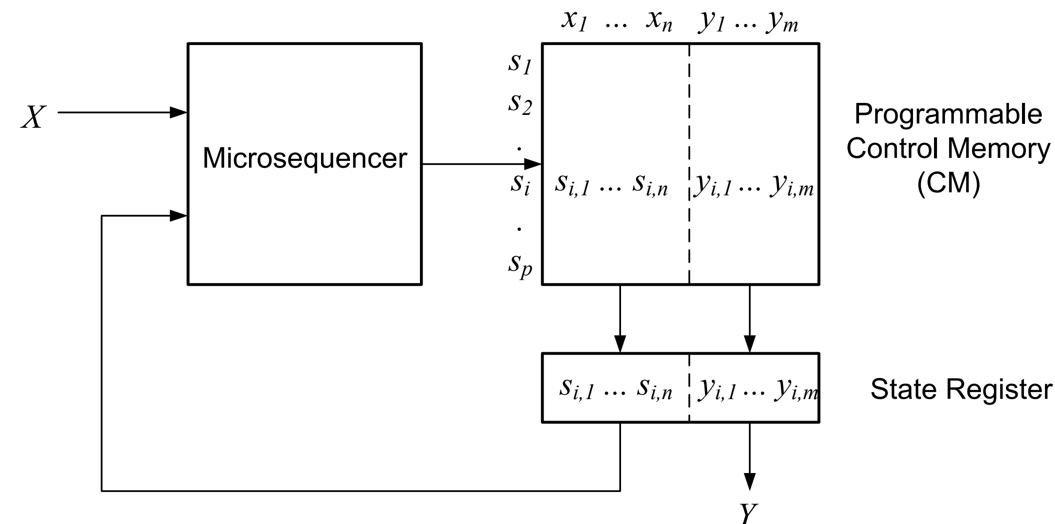


Microprogramming

- Important Questions
  - How do we compute the next address?
  - Which values do we store in microprogram memory?
  - How do we generate output signals?

Microprogramming

- Option 1: CM stores the complete state transition table
  - A row in the table corresponds to a microinstruction (i.e. next state and outputs)
  - The microsequencer selects the next state based on the input one of the next states as address for the CM



  - Advantage: most flexible, $\delta(x,s)$ and $\mu(s)$ (re)programmable
  - Drawback: CM can be very large and very slow

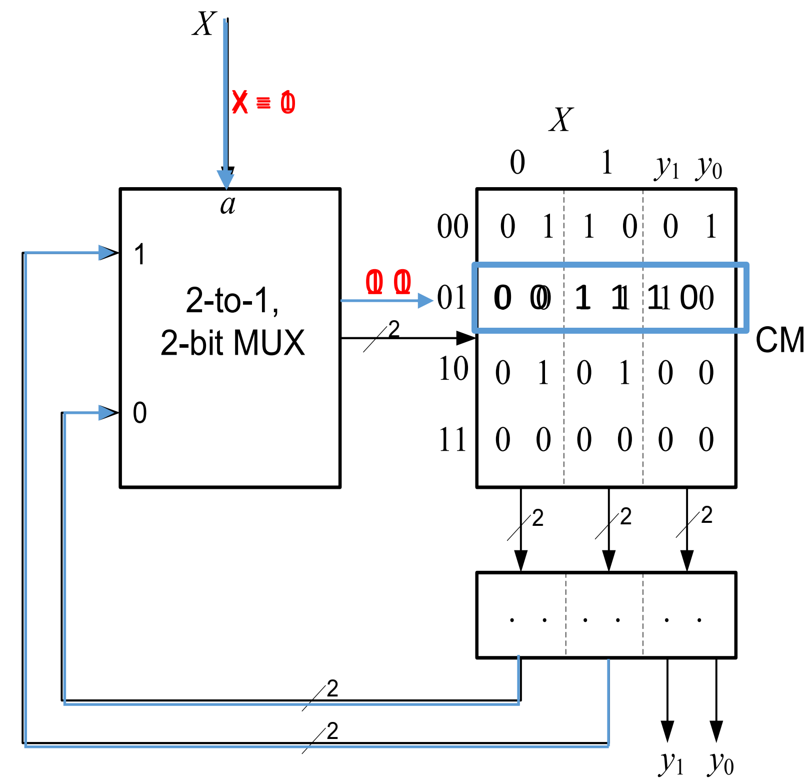
Microprogramming

- Example

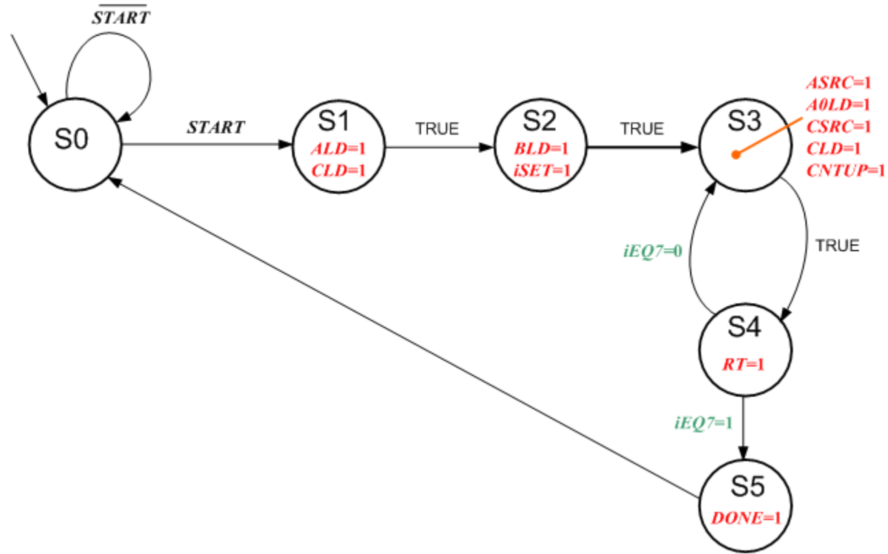
State Transition Table

	Y	X	
	y_1y_0	0	1
S_0	00	01	S_1	S_2
S_1	01	10	S_0	S_3
S_2	10	00	S_1	S_1
S_3	11	00	S_0	S_0

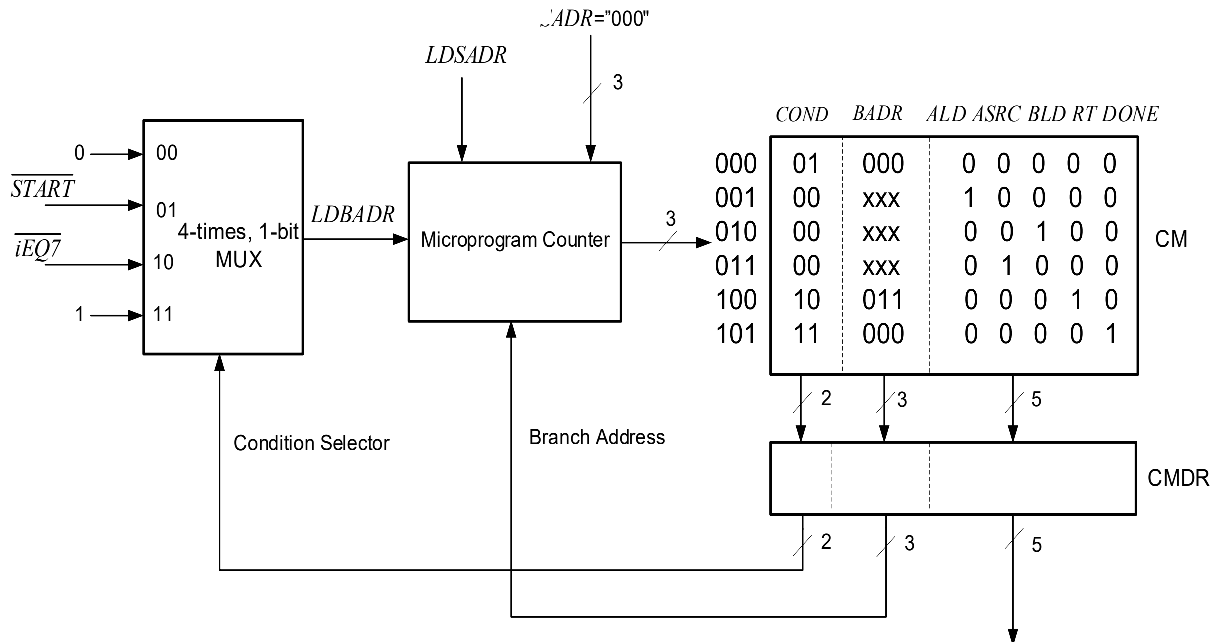
Microprogrammed Controller



SeqAdd(a,b) – Microprogrammed Control



S	ALD	ASRC	BLD	RT	DONE	(START, iEQ7) 0,0	0,1	1,0	1,1
000	0	0	0	0	0	000	000	001	001
001	1	0	0	0	0	010	010	010	010
010	0	0	1	0	0	011	011	011	011
011	0	1	0	0	0	100	100	100	100
100	0	0	0	1	0	011	101	011	101
101	0	0	0	0	1	000	000	000	000

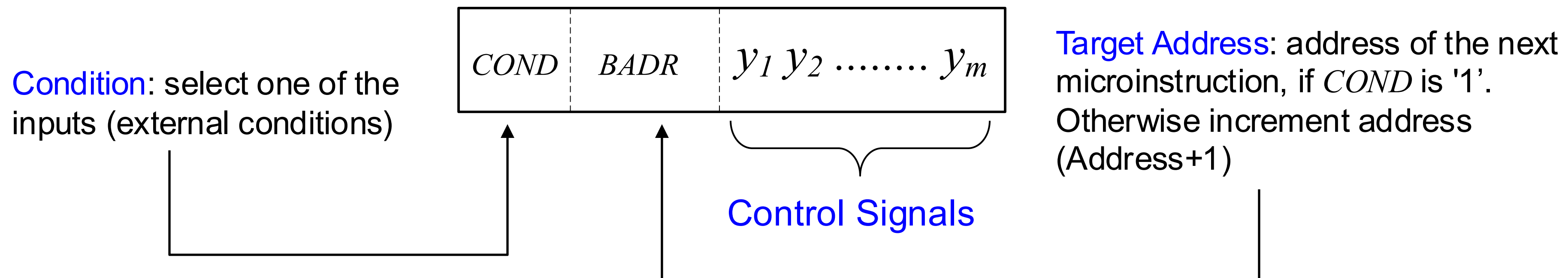


Computing the Next Address

- In general, the number of possible next states is very small
  - In particular, processor instructions are executed sequentially

    → the next instruction address is the current address + 1

  - Possible structure of the microinstructions



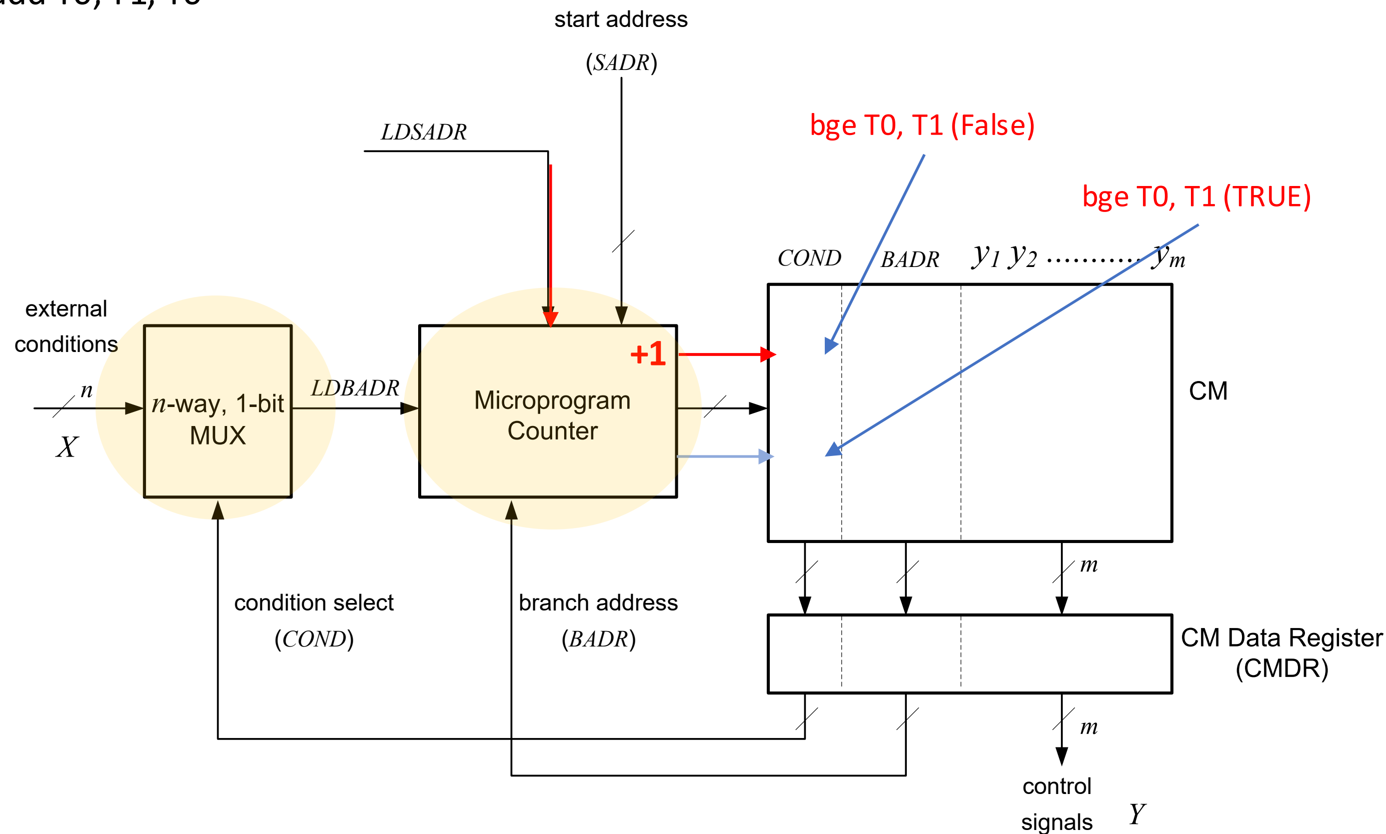
Computing the Next Address

- Option 2: CM store *COND*, *BADR* and output signals
  - The microsequencer consists of 2 components
    1. Microprogram Counter: computes addresses for the CM
       - The microprogram counter is incremented with positive clock edge
       - *LDBADR* (load branch address) = 1: the address of the next microinstruction is loaded in the microprogram counter
       - *LDSADR* (load start address) = 1: the start address *SADR* is loaded in the microprogram counter
         - Microprogram memory in processors store microprograms
    2. MUX, uses *COND* as selector for the inputs

Microprogramming

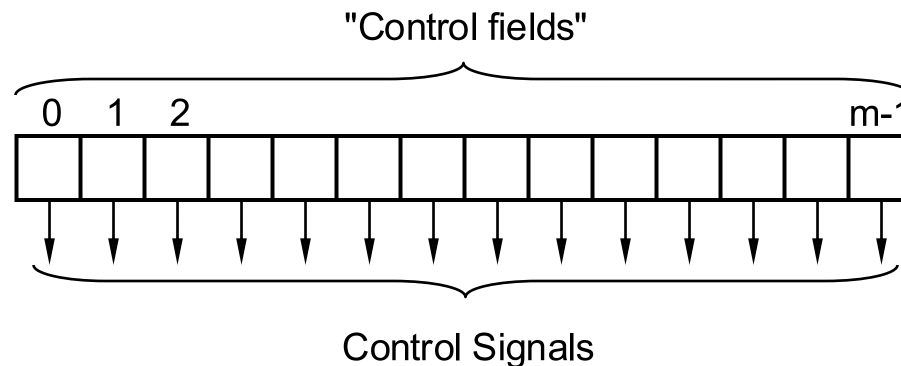
add T0, 0, 0
addi T1, 0, 10
bge T0, T1, BCH # IF (T0 >= T1) GOTO BCH
addi T0, T0, 1 #i++
BCH: add T0, T1, T0

```
        add T0, 0, 0
        addi T1, 0, 10
        bge T0, T1, BCH # IF (T0 >= T1) GOTO BCH
        addi T0, T0, 1 #i++
BCH:    add T0, T1, T0
```



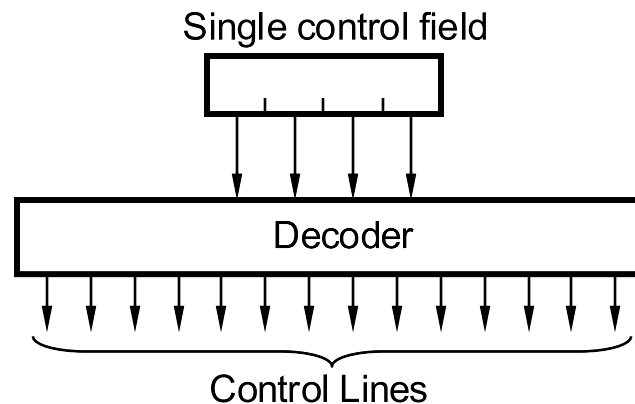
Generating Output Signals

- Horizontal microprogramming
  - 1 bit is used for each output signal (a control field) in CMDR → "horizontal" microprogramming
  - Advantage: maximal parallelism in data path
    - All control signals can be set independently
  - Drawback: wide microinstruction → large control memory (CM)
    - Combination of output signals are sometime possible



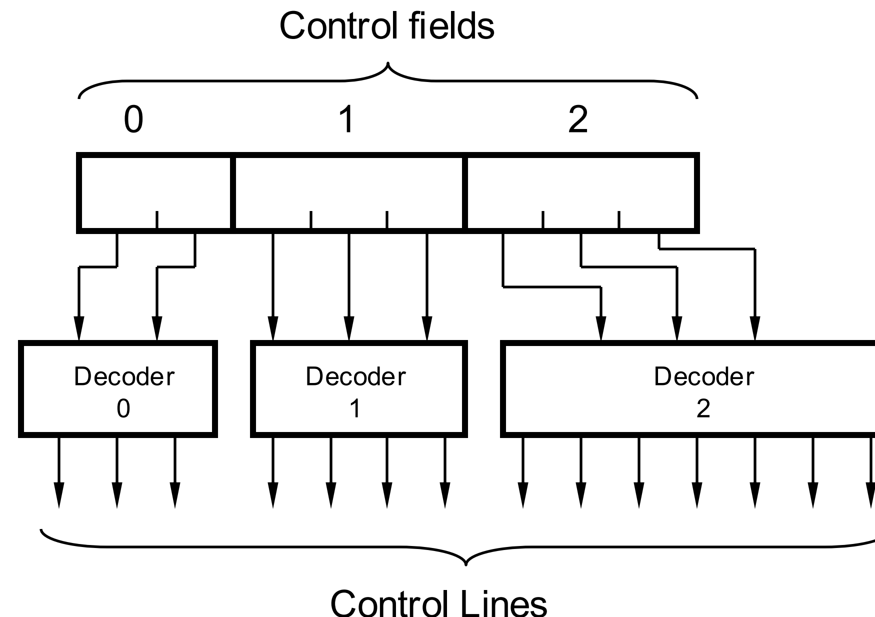
Generating Output Signals

- Vertical microprogramming
  - Only one control field (with $\lceil ld(m) \rceil$ bit) is stored
    - The value in the field corresponds to the binary coding of the signal that must be set $\rightarrow$ "vertical" microprogramming
  - Advantage: small control memory (CM)
  - Drawback: only one signal can be set at a time; Slow; No parallelism



Generating Output Signals

- Diagonal microprogramming
  - Control signals that can/must not be simultaneously active are grouped in a control field and binary coded
    → "diagonal" or "zoned" microprogramming
  - The number of fields defines the level of parallelism

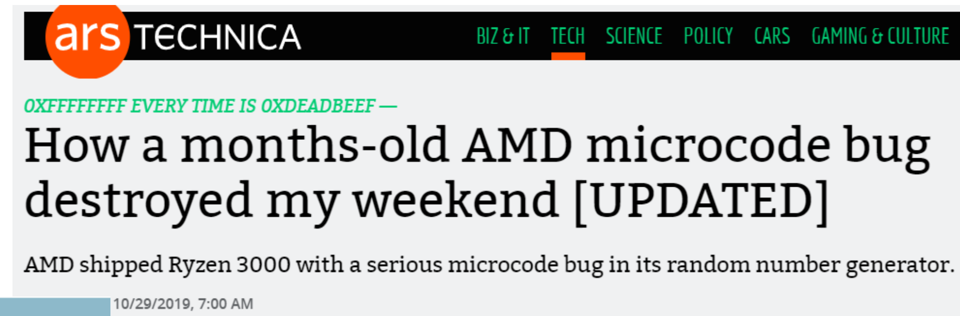


Microprogramming

Home / Component / CPU / AMD Ryzen CPUs are set for a microcode update in November

AMD Ryzen CPUs are set for a microcode update in November

James Dawson · 3 weeks ago · CPU, Featured Tech News, Motherboard


ars TECHNICA — BIZ & IT · TECH · SCIENCE · POLICY · CARS · GAMING & CULTURE

OXFFFFFFFF EVERY TIME IS OXDEADBEEF —

How a months-old AMD microcode bug destroyed my weekend [UPDATED]

AMD shipped Ryzen 3000 with a serious microcode bug in its random number generator.

10/29/2019, 7:00 AM

AMD takes security vulnerabilities very seriously and seeks to respond quickly and appropriately.

We regularly issue security bulletins to our partners. Such communications can include a description of the vulnerabilitie
our CVSS scores and attributions to reporters of those vulnerabilities

AMD recommends following the security best practices of keeping your operating system up-to-date, operating at the la
BMC/TSM, FW, etc)., utilizing safe computer practices and running antivirus software

The Pentium FDIV Bug

computer science · 30. October 2015 · 0 · Harald Sack

Summary of Intel microcode updates

Applies to: Windows Server 2019, all versions, Windows 10, version 1809, Windows 10, version 1803, More

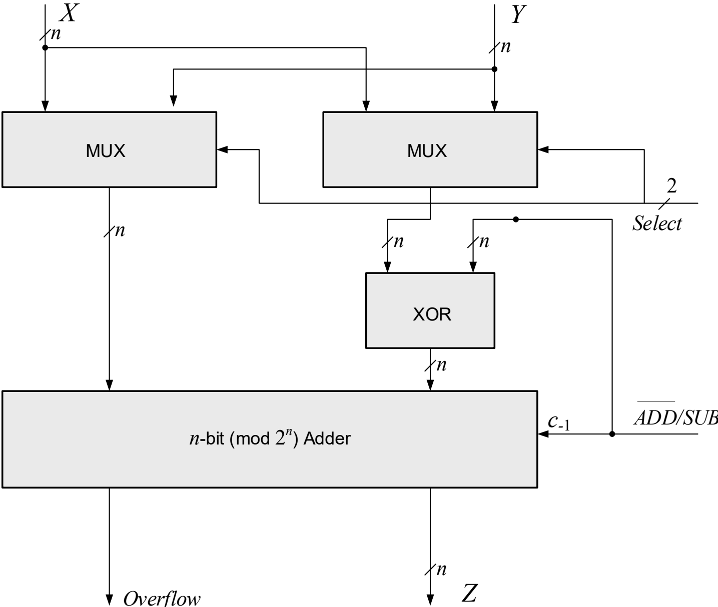
Intel microcode updates

Microsoft is making available Intel-validated microcode updates that are related to Spectre Variant 2 (CVE 2017-5715 ["Branch Target Injection"]).

The following table lists specific Microsoft Knowledge Base articles by Windows version. The article contains links to the available Intel microcode updates by CPU:

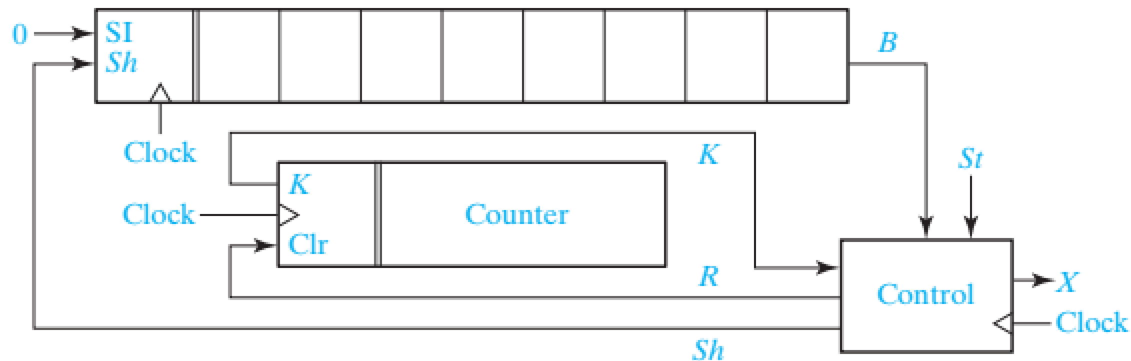
KB Number and Description	Windows Version	Source
KB4100347 Intel Microcode Updates	Windows 10, version 1803, and Windows Server, version 1803	Windows Update, Windows Server Update Service, and Microsoft Update Catalog

Controller for the Add/Sub

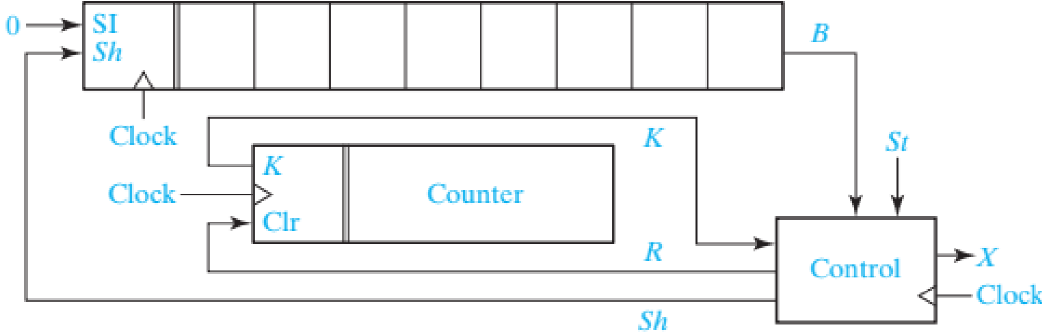


Odd-Parity Generator

- Design a controller for an odd-parity generator.
  - The circuit should transmit 7 bits from a shift register onto the output X.
  - Then, on the next clock cycle, the eighth value of X should be chosen to make the number of 1's be odd.
    - In other words, the last value of X should be 1 if there was an even number of 1's in the shift register, so that the 8-bit output word will have odd parity.



Odd-Parity Generator



- An 8-bit word is placed at the input of the parallel-to-serial register and a *start* signal activated on the controller
- Upon receiving the *start* signal, the controller initiates the loading of the word in the register, using the signal *load*. The controller also clears the counter using the *clear* input of the counter to restart the counting process
- The shift register then places the LSB bit on the transmission line every clock cycle. **No control is needed for this. The shift is directly encoded in the shift register**
- When all bits have been transmitted, the controller sets it's *done* output to 1
- Explain how the controllers knows the end of the 8-bit transmission
- Draw the block diagram (no implementation) of the system (controller+datapath)
- Draw the **Moore state transition diagram** of the controller
- Devise the **Moore state transition table** of the controller

LEADING THE CHARGE, CHARGING AHEAD