

ארכיטקטורת יח' עיבוד מרכזית

361-1-4201

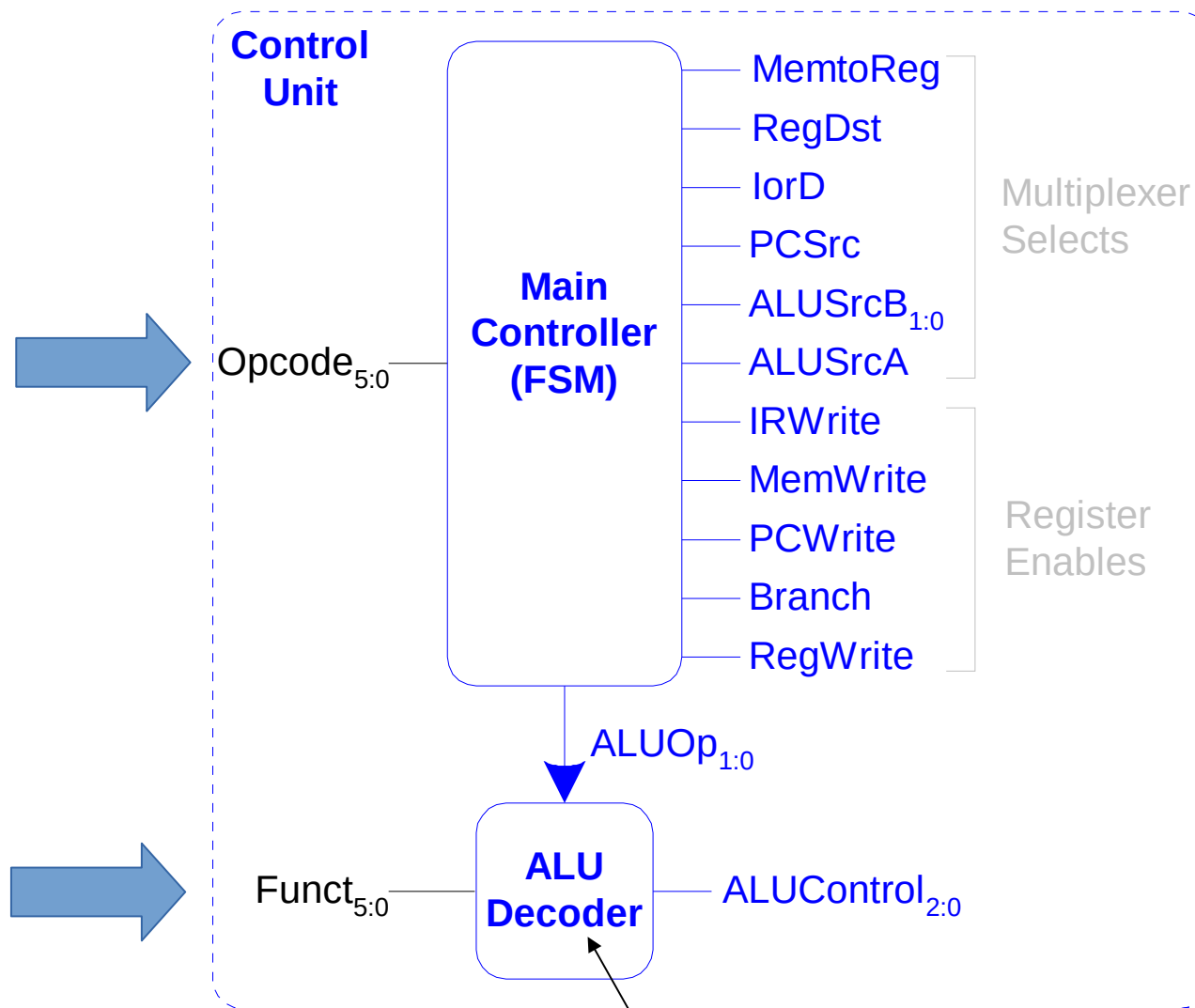
הרצאה מס' 7

**Microprogrammed Microarchitectures
and Pipelining I**

ד"ר גיא תל-צור

Multi-Cycle Control (MIPS!)

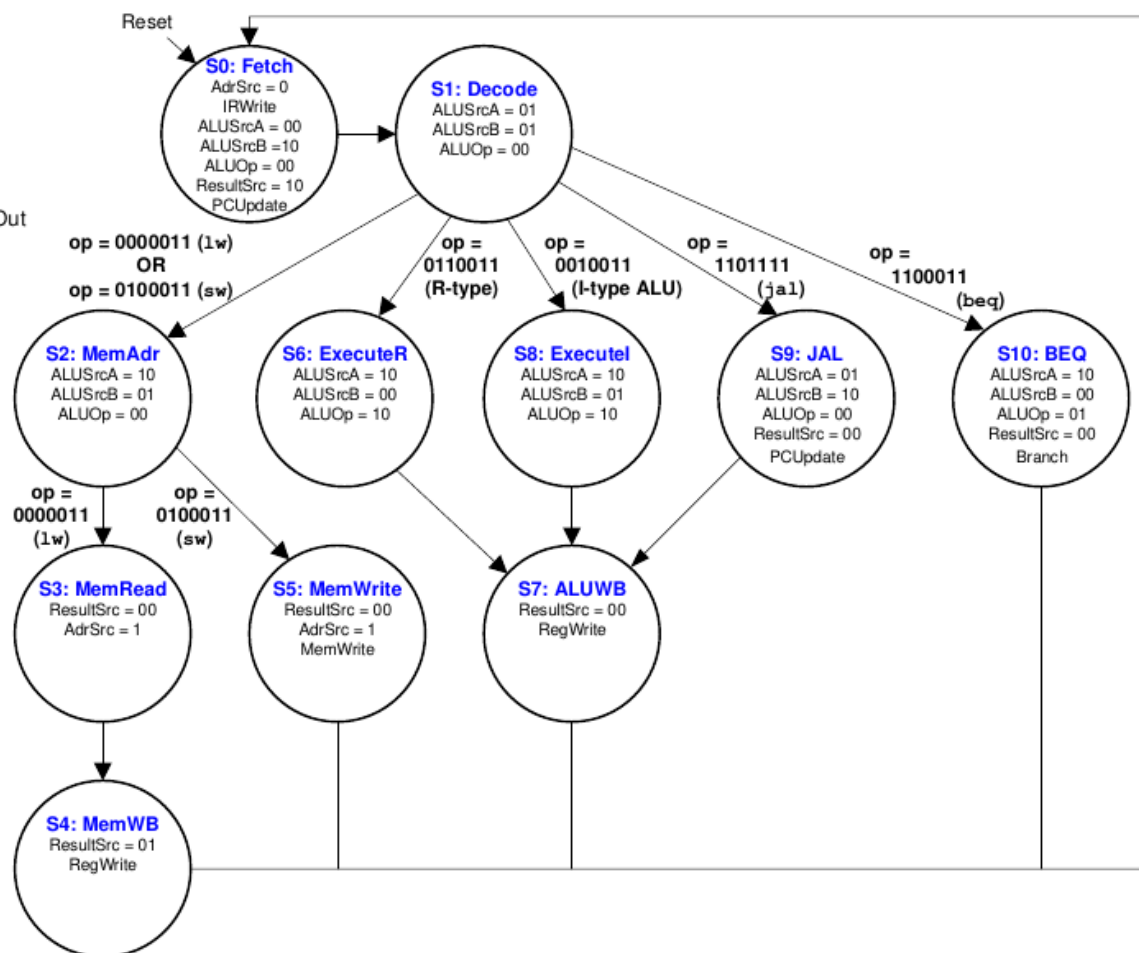
הבקר ב- Multi-cycle



דיאגרמת המצבים במעבד RISC-V

Multicycle Processor Main FSM

State	Datapath μ Op
Fetch	Instr $\leftarrow$ Mem[PC]; PC $\leftarrow$ PC+4
Decode	ALUOut $\leftarrow$ PCTarget
MemAdr	ALUOut $\leftarrow$ rs1 + imm
MemRead	Data $\leftarrow$ Mem[ALUOut]
MemWB	rd $\leftarrow$ Data
MemWrite	Mem[ALUOut] $\leftarrow$ rd
ExecuteR	ALUOut $\leftarrow$ rs1 op rs2
ExecuteI	ALUOut $\leftarrow$ rs1 op imm
ALUWB	rd $\leftarrow$ ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC $\leftarrow$ ALUOut
JAL	PC $\leftarrow$ ALUOut; ALUOut $\leftarrow$ PC+4



MIPS FSM

בתרשים זה
מתוארים 10
מצבים

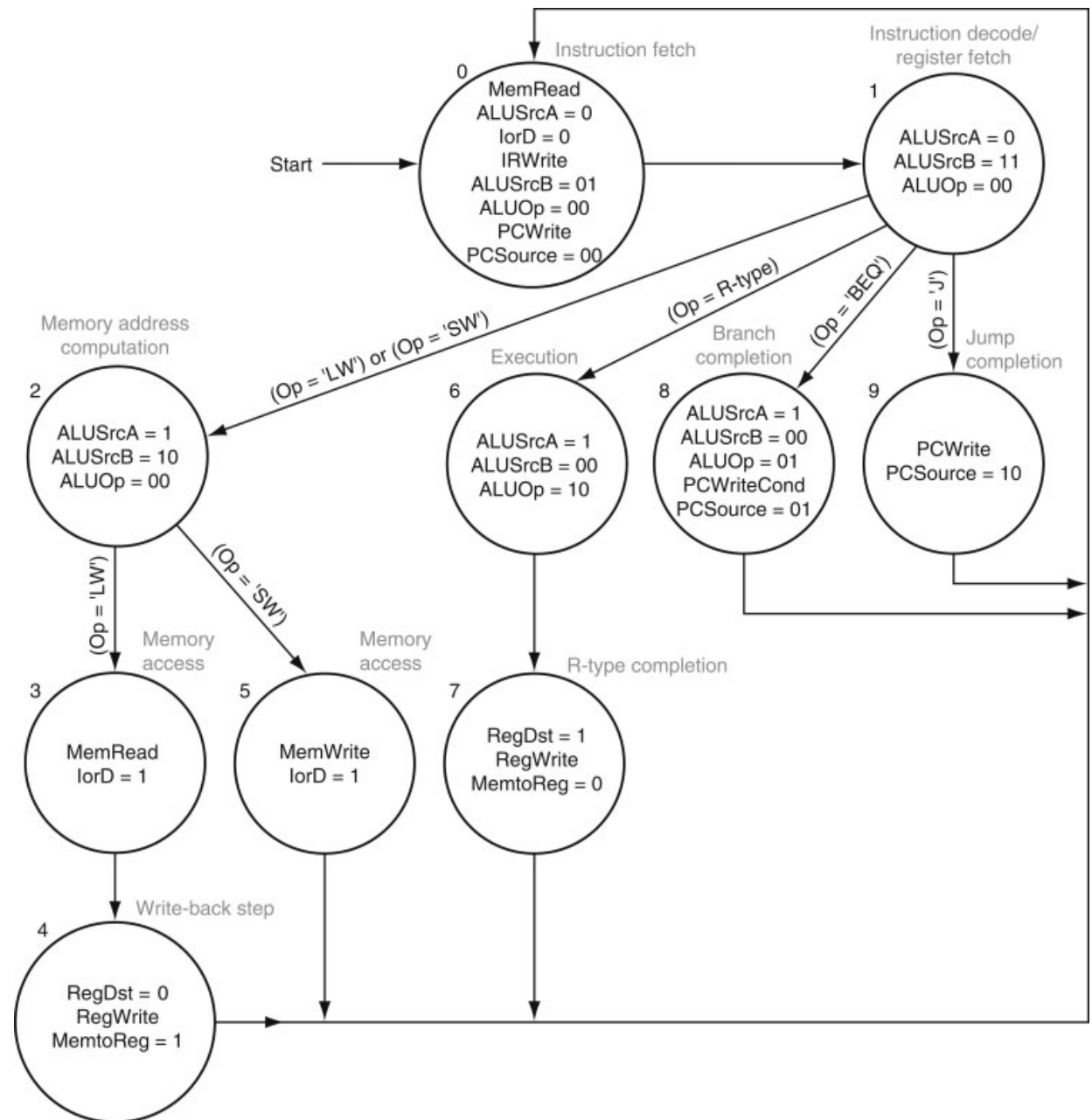
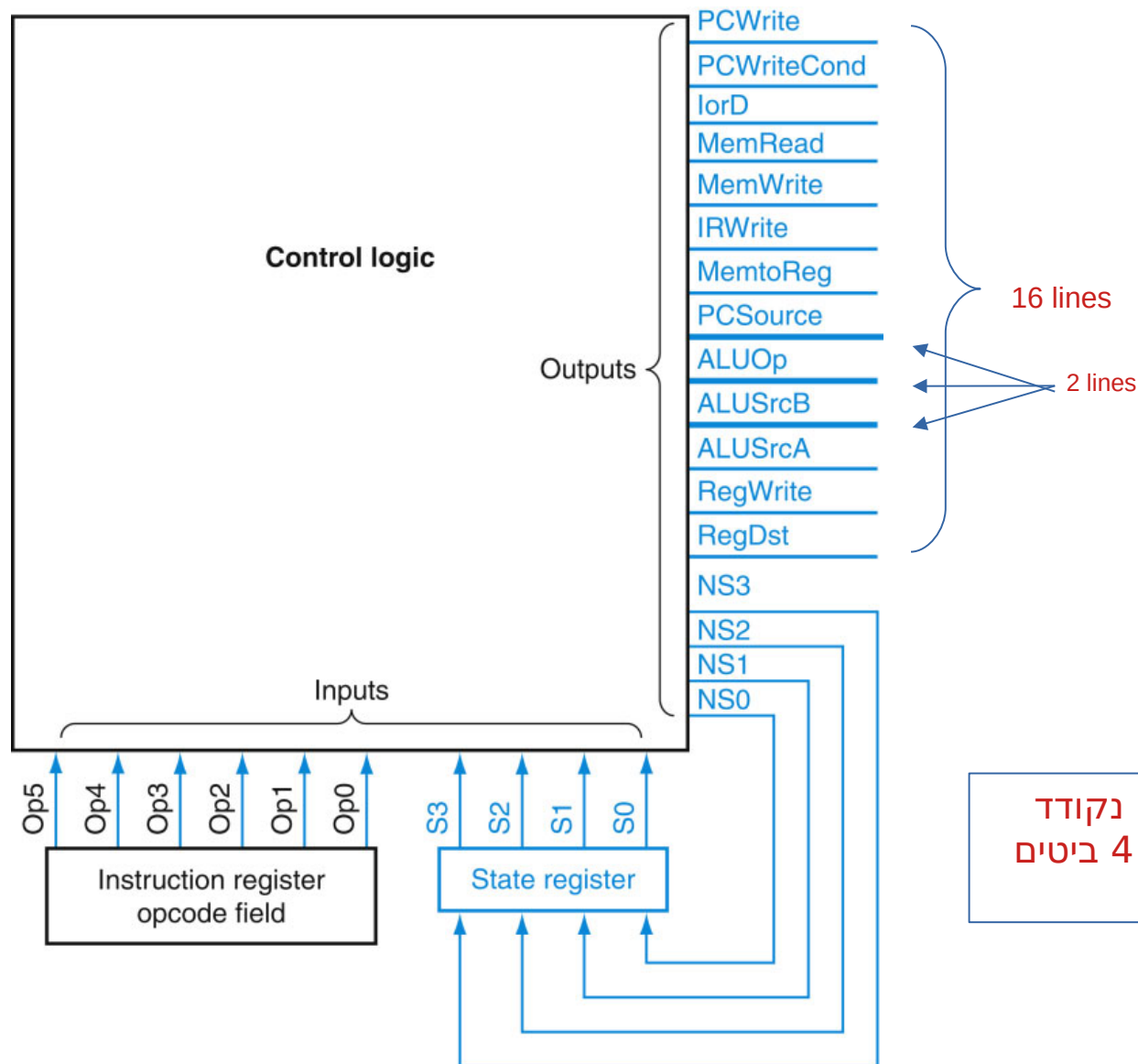


FIGURE D.3.1 The finite-state diagram for multicycle control.

10 כניסות.
 20 יציאות.
 כמה אחסון
 נדרש?
 10^2
 20,480bit=20
 ===20kb
 2,560byte



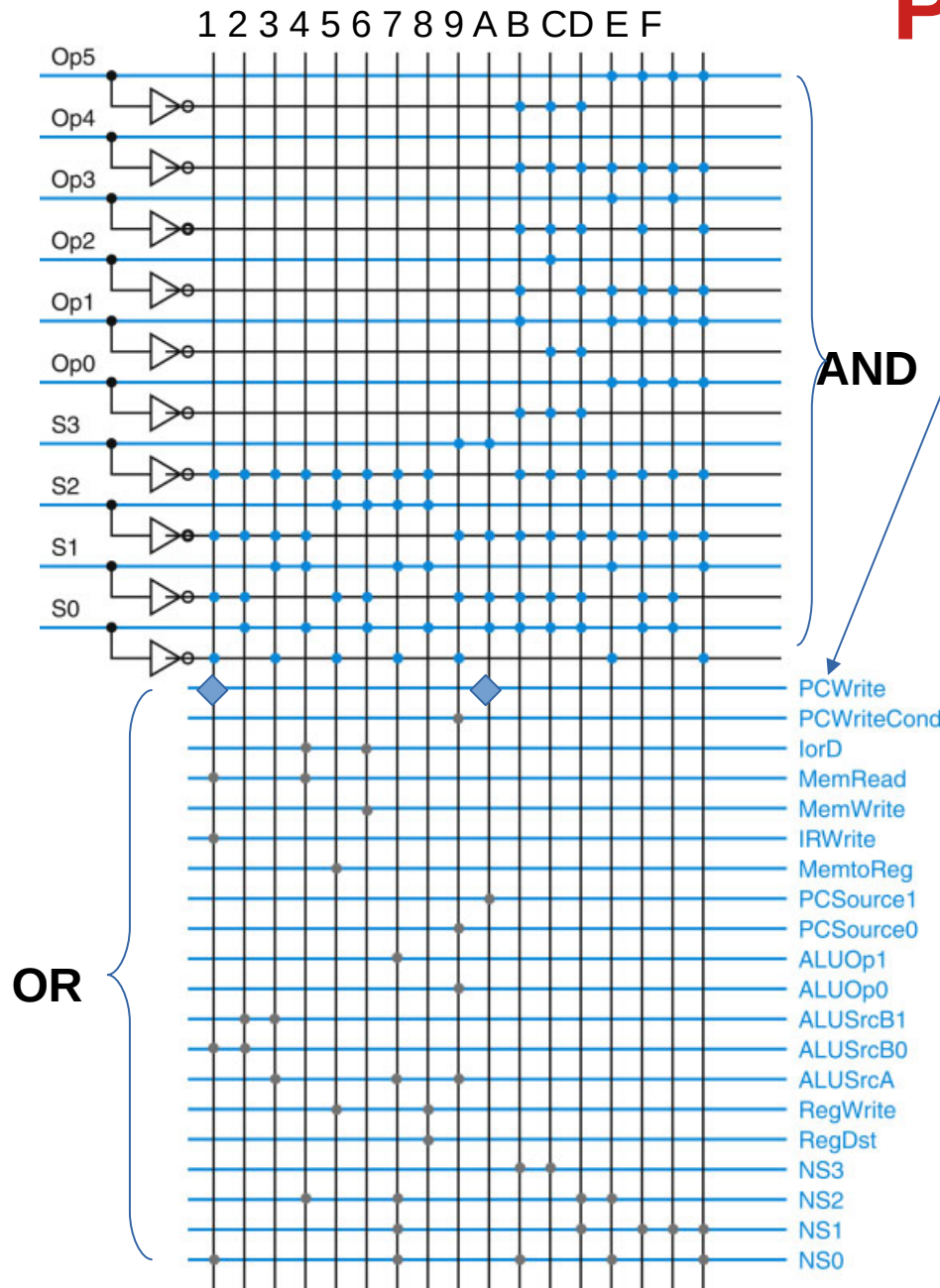
10 מצבים נקודד
 באמצעות 4 ביטים

FIGURE D.3.2 The control unit for MIPS will consist of some control logic and a register to hold the state. The state register is written at the active clock edge and is stable during the clock cycle

PLA

דוגמה

לוגיקה



AND

PCWrite= "1" + "A"

"1" = not(S0)*not(S1)*not(S2)*not(S3)

"A"=S3*not(S2)*not(S1)*S0

תזכור PCWrite הוא פעיל ב-State0-1

State0=0000

State9=1001

ראו טבלת אמת ב- D.3.4

ש: איך מאחסנים
טבלה?
ת: PLA

Multi-Cycle Microprogram Control

Implementing the **Next-State** Function with a Sequencer

Appendix D Part 2 PPTs

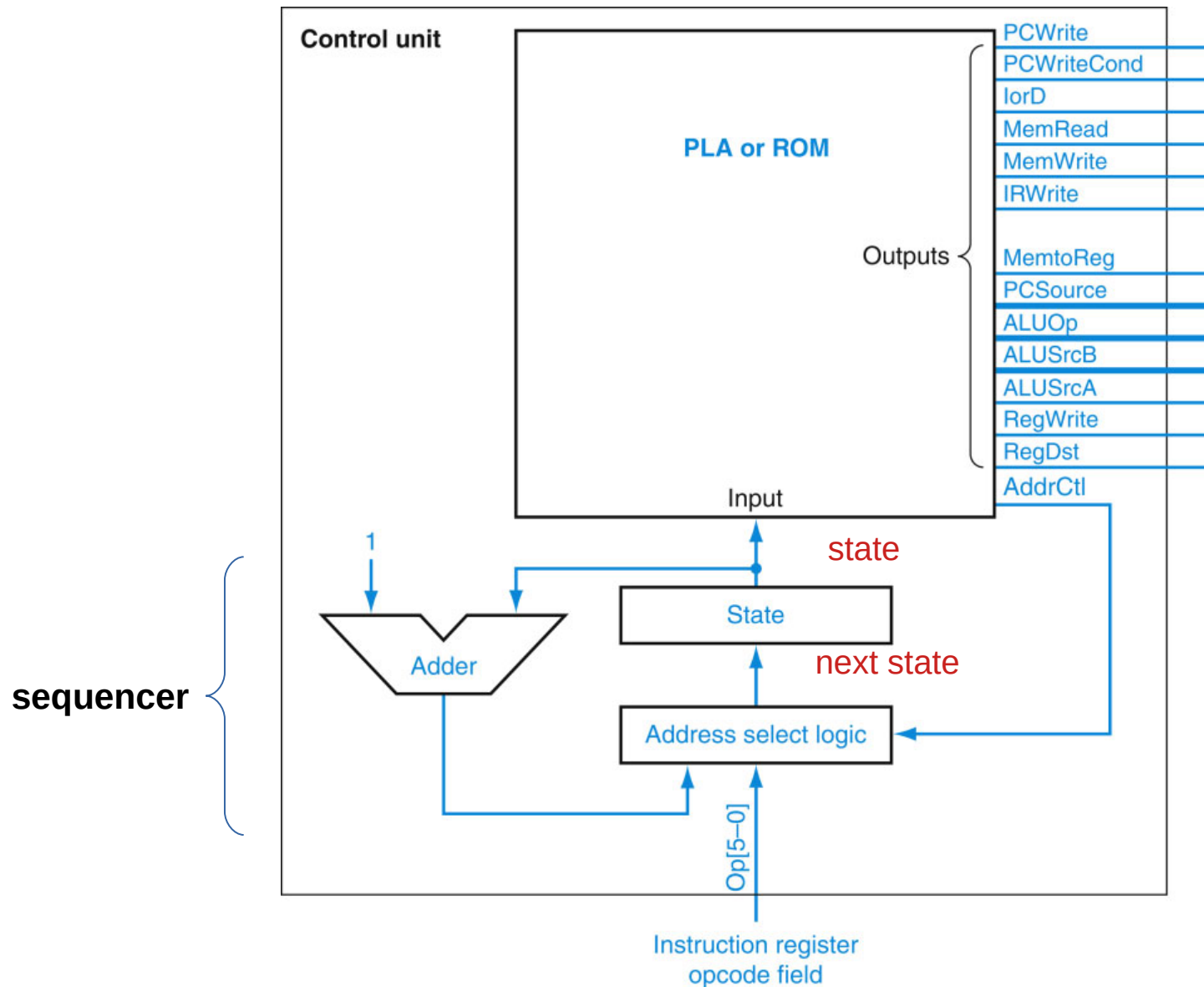


FIGURE D.4.1 The control unit using an explicit counter to compute the next state. In this control unit, the next state is computed using a counter (at least in some states). By comparison, Figure D.3.2 encodes the next state in the control logic for every state. In this control unit, the signals labeled *AddrCtl* control how the next state is determined.

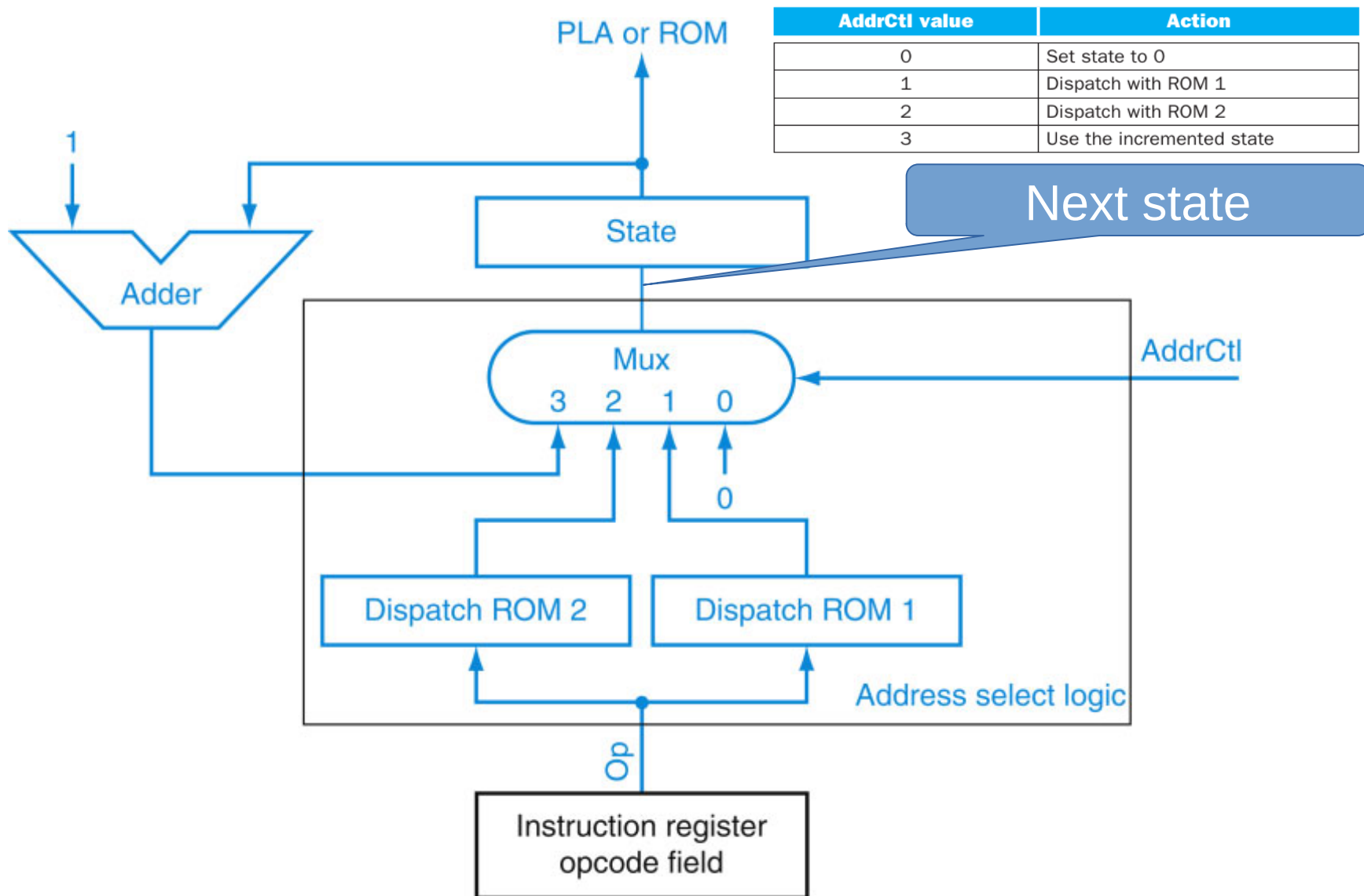


FIGURE D.4.2 This is the address select logic for the control unit of Figure D.4.1.

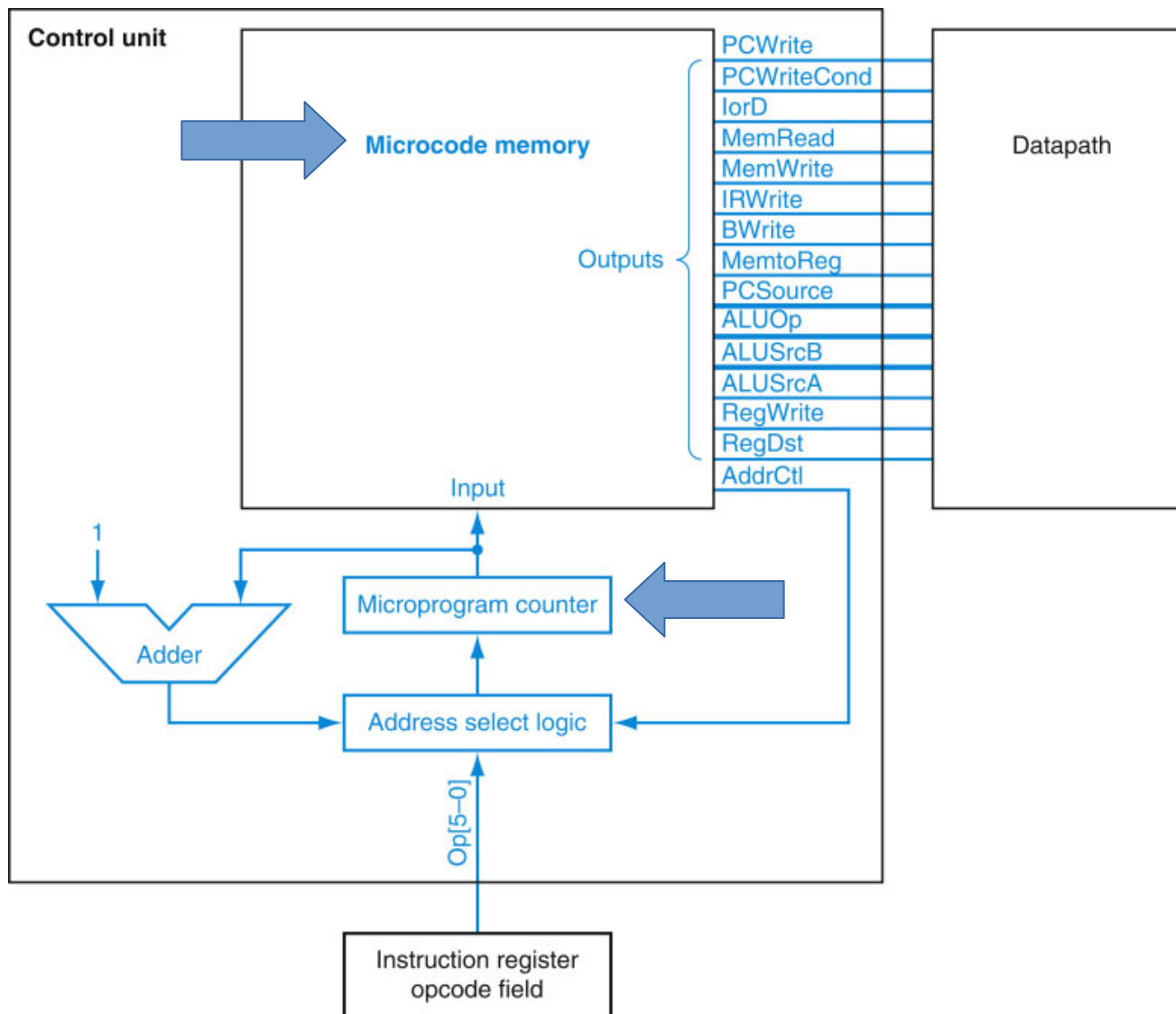


FIGURE D.4.6 The control unit as a microcode. The use of the word “micro” serves to distinguish between the program counter in the datapath and the **microprogram counter**, and between the **microcode memory** and the instruction memory.

Translating a Microprogram to Hardware (Control Signals)

ראינו שאפשר לממש מנגנון של מיקרופרוגרם כמכונת מצבים סופים עם או ללא SEQUENCER.

עכשיו בנקודת הסתכלות מעט שונה:
יש לנו יכולת ליצור מעין שפת תכנות (חדשה) של מיקרו הוראות אשר שולטת על 18 אותות הבקרה ועל 4 מצבי התזמון – ראו השקף הבא

ראו את טבלת
האמת של ה-
ALU decoder
במצגת הקודמת

22 הוראות =
18 קווי בקרה
+
4 מצבי
sequencer

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	MemRead, lorD = 0, IRWrite	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, lorD = 1	Read memory using ALUOut as address; write result into MDR.
	Write ALU	MemWrite, lorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00, PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	Jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

FIGURE D.5.1 Each microcode field translates to a set of control signals to be set. These 22 different values of the fields specify all the required combinations of the 18 control lines. Control lines that are not set, which correspond to actions, are 0 by default. Multiplexor control lines are set to 0 if the output matters. If a multiplexor control line is not explicitly set, its output is a don't care and is not used.

לעמוד

Dispatch ROM 1		
Op	Opcode name	Value
000000	R-format	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

Dispatch ROM 2		
Op	Opcode name	Value
100011	lw	0011
101011	sw	0101

פיצול ממצב 2 - AdrCmp

פיצול ממצב 1 - Decode

כל שאר המצבים יתקבלו מהוספת 1 למצב באמצעות המונה או יובילו שוב להתחלה, מצב 0 - ראו השקף הבא.

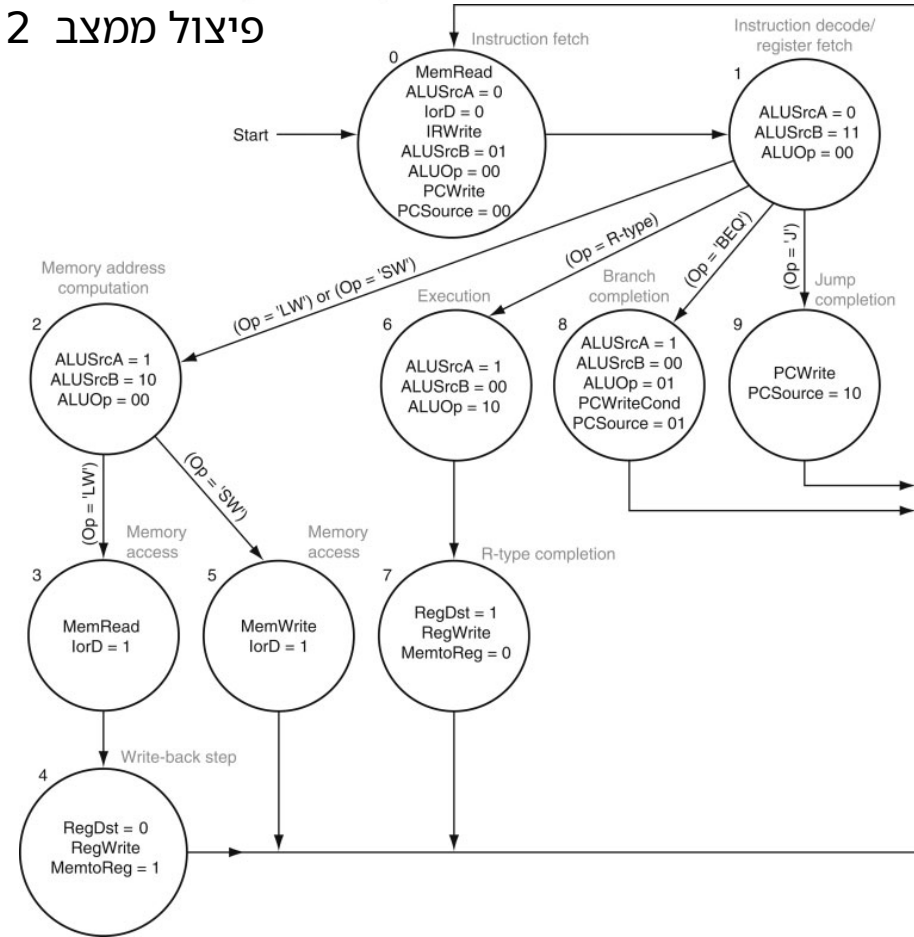


FIGURE D.4.3 The dispatch ROMs each have $2^5 = 32$ entries that are 4 bits wide, since that is the number of bits in the state encoding. This figure only shows the entries in the ROM that are of interest for this subset. The first column in each table indicates the value of Op, which is the address used to access the dispatch ROM. The second column shows the symbolic name of the opcode. The third column indicates the value at that address in the ROM.

The sequencing field can have 4 values: Fetch, Dispatch1, Dispatch2 and Sequential.

dispatch table 1			Microcode dispatch table 2		
Opcode field	Opcode name	Value	Opcode field	Opcode name	Value
000000	R-format	Rformat1	100011	lw	LW2
000010	jmp	JUMP1	101011	sw	SW2
000100	beq	BEQ1			
100011	lw	Mem1			
101011	sw	Mem1			

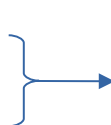

 2 השורות האלה מופיעות פעמיים, לכן נדרשים 2 רומים

FIGURE D.5.2 The two microcode dispatch ROMs showing the contents in symbolic form and using the labels in the microprogram.

Summary

Independent of whether **the control is represented as a finite-state diagram** or as a **microprogram**, translation to a hardware control implementation is similar. Each state or microinstruction asserts a set of control outputs and specifies how to choose the next state.

The **next-state function** may be implemented by either **encoding it in a finite-state machine** or using an explicit **sequencer**. The **explicit sequencer is more efficient** if the number of states is large and there are many sequences of consecutive states without branching.

The control signals decoder

We just implement the table of slide 54:

Let's look at ALUSrcA: it is "0" in states 0 and 1 and it is "1" in states 2, 6 and 8. In all other states we don't care.

let's look at PCWrite: it is "1" in states 0 and 9. In all other states it must be "0".

And so, we'll fill the table below and build the decoder

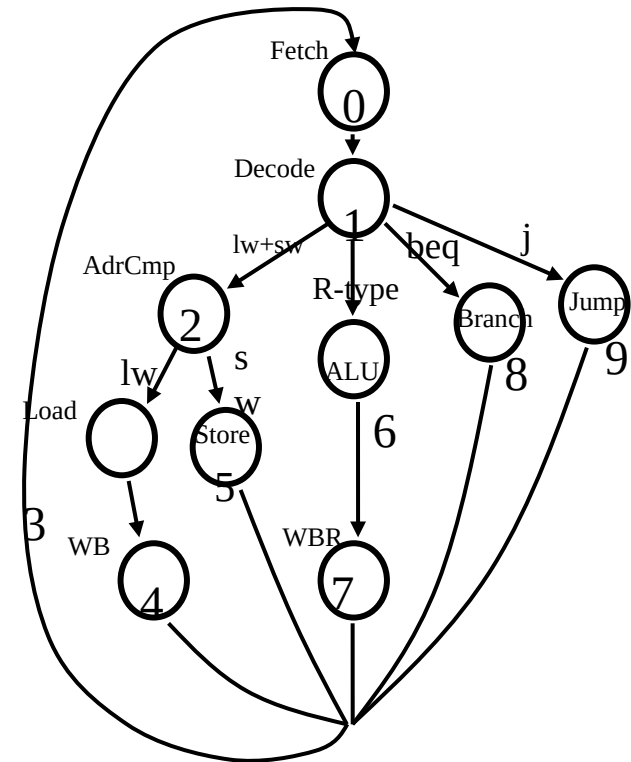
	state				Control signals			
	S3	S2	S1	S0	ALUSrcA	PCWrite	PCWriteCond	
fetch	0	0	0	0	0	1	0	
decode	0	0	0	1	0	0	0	
AdrCmp	0	0	1	0	1	0	0	
load	0	0	1	1	X	0	0	
WB	0	1	0	0	X	0	0	
store	0	1	0	1	X	0	0	
ALU	0	1	1	0	1	0	0	
WBR	0	1	1	1	X	0	0	
branch	1	0	0	0	1	0	1	
jump	1	0	0	1	X	1	0	
	All other				X	0	0	

נפח האחסון:

$2^4 \times 16 = 256\text{bits}$

State Machine “next state calc.” logic

	opcode						current state				next state			
	IR31	IR30	IR29	IR28	IR27	IR26	S3	S2	S1	S0	S3	S2	S1	S0
	X	X	X	X	X	X	0	0	0	0	0	0	0	1
R-type	0	0	0	0	0	0	0	0	0	1	0	1	1	0
lw+sw	1	X	X	X	X	X	0	0	0	1	0	0	1	0
lw	X	X	0	X	X	X	0	0	1	0	0	0	1	1
sw	X	X	1	X	X	X	0	0	1	0	0	1	0	1



R-type=000000, lw=100011, sw=101011, beq=000100, bne=000101, lui=001111,
j=0000010, jal=000011, addi=001000

נפח האחסון: $2^{10} \times 4 = 4096 \text{ bits}$

עתה, תוכן ה-ROM "הגדול" במימוש עם מונה: עשר מילים (שורות) על $2+16$ עמודות (ביטים), סה"כ 180 ביטים.

בנוסף, 2 ה-dispatchers, כ"א ברוחב (=מוצא) של 4 ביט על 6 מצבים בכניסה (64 ביט):
 $2^6 \text{ combinations} \times 4 \text{ bit value} = 256 \times 4 = 1024$
 dispatchers: $2 \times 256 = 512$
 Total: $180 + 512 = 692 \text{ bit}$
 (64x4) כפול 2 = 512 ביט, ובסה"כ 692 ביט (לעומת 4.3Kb גרסת ה-ROM הראשונה). אפשר אפילו לצמצם יותר מאחר ורוב ה-dispatchers הם sparse.

10X18=180bit
 10 שורות
 ברוחב
 $18=16+2$

State number	Control word bits 17-2	Control word bits 1-0
0	1001010000001000	11
1	0000000000011000	01
2	0000000000010100	10
3	0011000000000000	11
4	0000001000000010	00
5	0010100000000000	00
6	0000000001000100	11
7	0000000000000011	00
8	0100000010100100	00
9	1000000100000000	00

See table D.3.3

FIGURE D.4.5 The contents of the control memory for an implementation using an explicit counter. The first column shows the state, while the second shows the datapath control bits, and the last column shows the address-control bits in each control word. Bits 17-2 are identical to those in Figure D.3.7.

From T1 & T2 : על-פי המודל המקורי (ללא SQUENCER):

המצבים - 4.3Kb, total: $2^{10} \times 4 = 4096$, T2: $2^4 \times 16 = 256$, אותות הבקרה, T1:

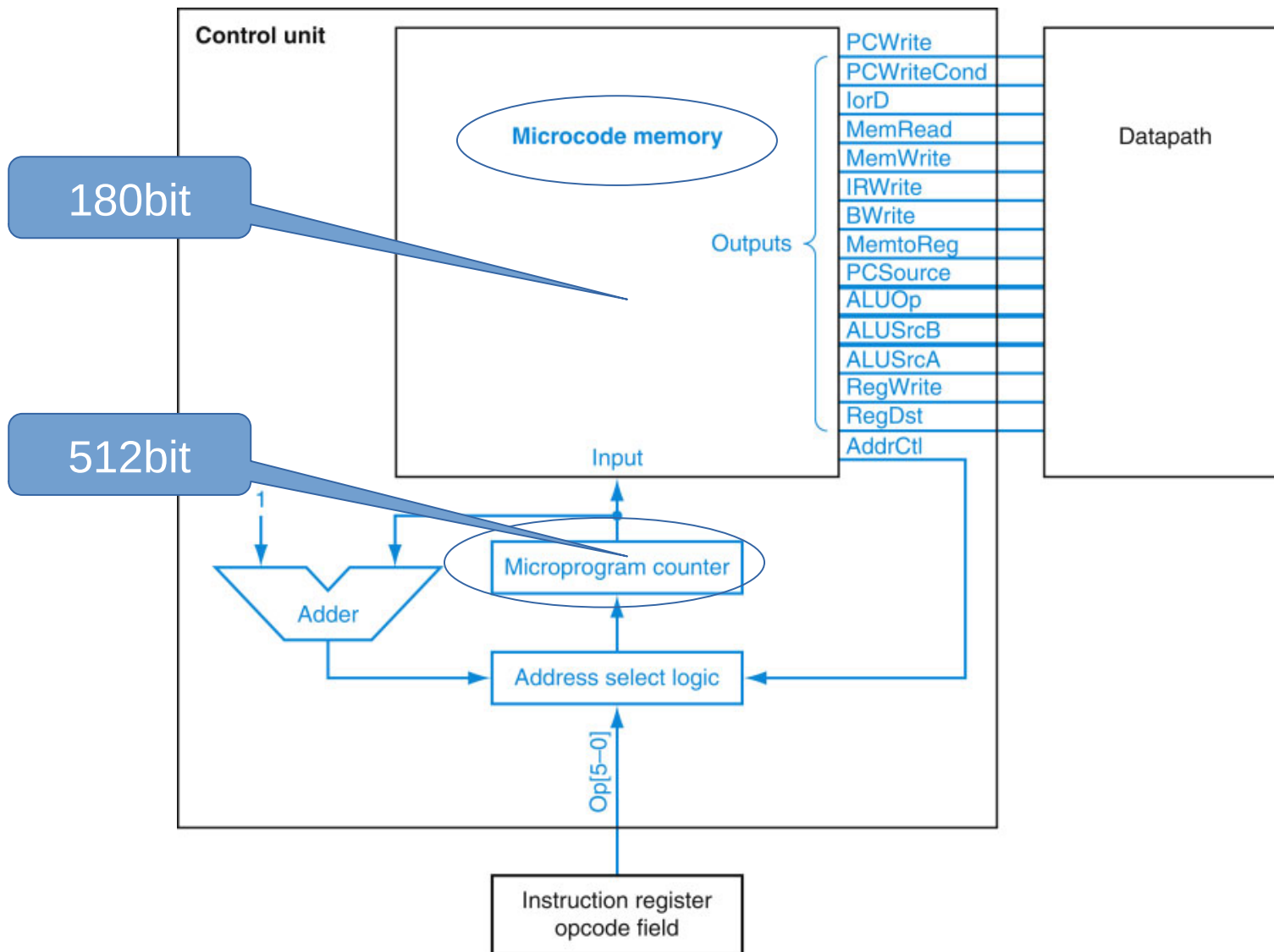
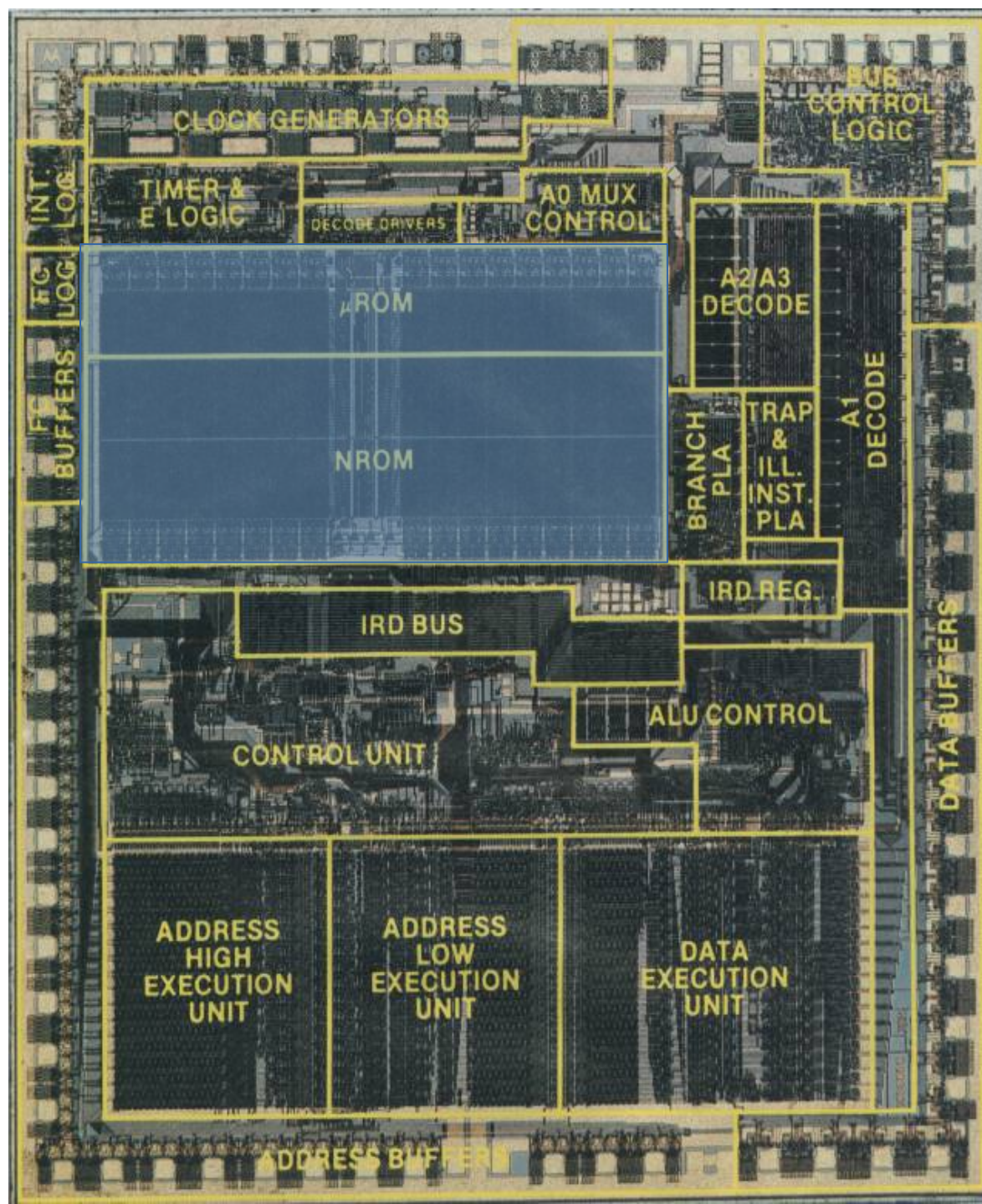


FIGURE D.4.6 The control unit as a microcode. The use of the word “micro” serves to distinguish between the program counter in the datapath and the microprogram counter, and between the microcode memory and the instruction memory.

Motorola 68000

http://www.easy68k.com/paulrsm/doc/dp_bm68k2.htm



Bus Interface Unit (BIU)



<https://www.righto.com/2023/04/8086-microcode-string-operations.html>

Execution Unit (EU)

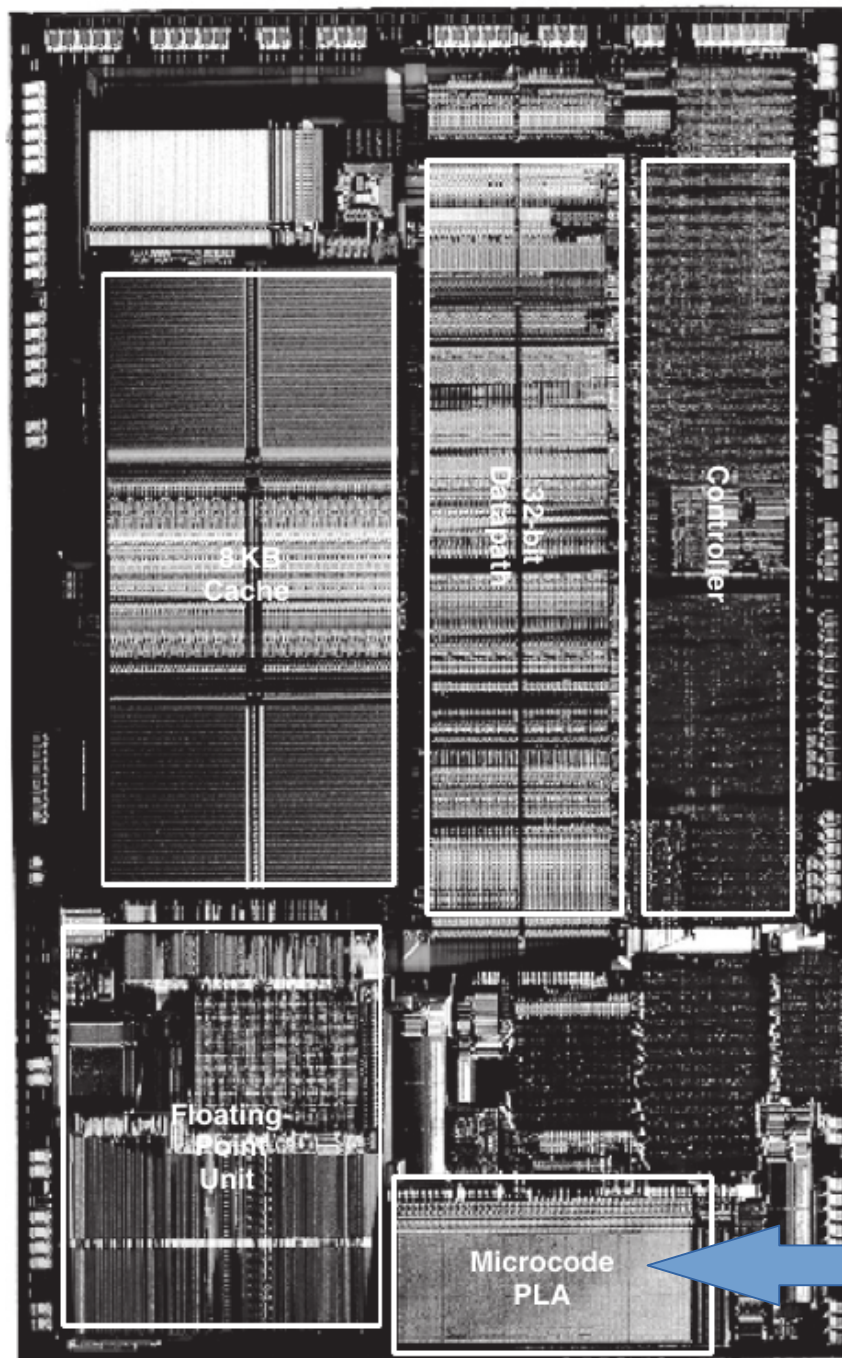


Figure 7.75 80486
microprocessor chip

Credit: Harris & Harris
DD&CA

End of Appendix D slides (MIPS)
Appendix C (RISC-V Edition)

Thoughts on Control & Microprogramming

The Control Store: Some Questions

- What control signals can be stored in the control store?

אלה שאינם תלויים בנתונים Those independent on data

vs.

- What control signals have to be generated in hardwired logic?
 - i.e., what signal cannot be available without processing in the datapath?

אלה שתלויים בנתונים Those dependent on data

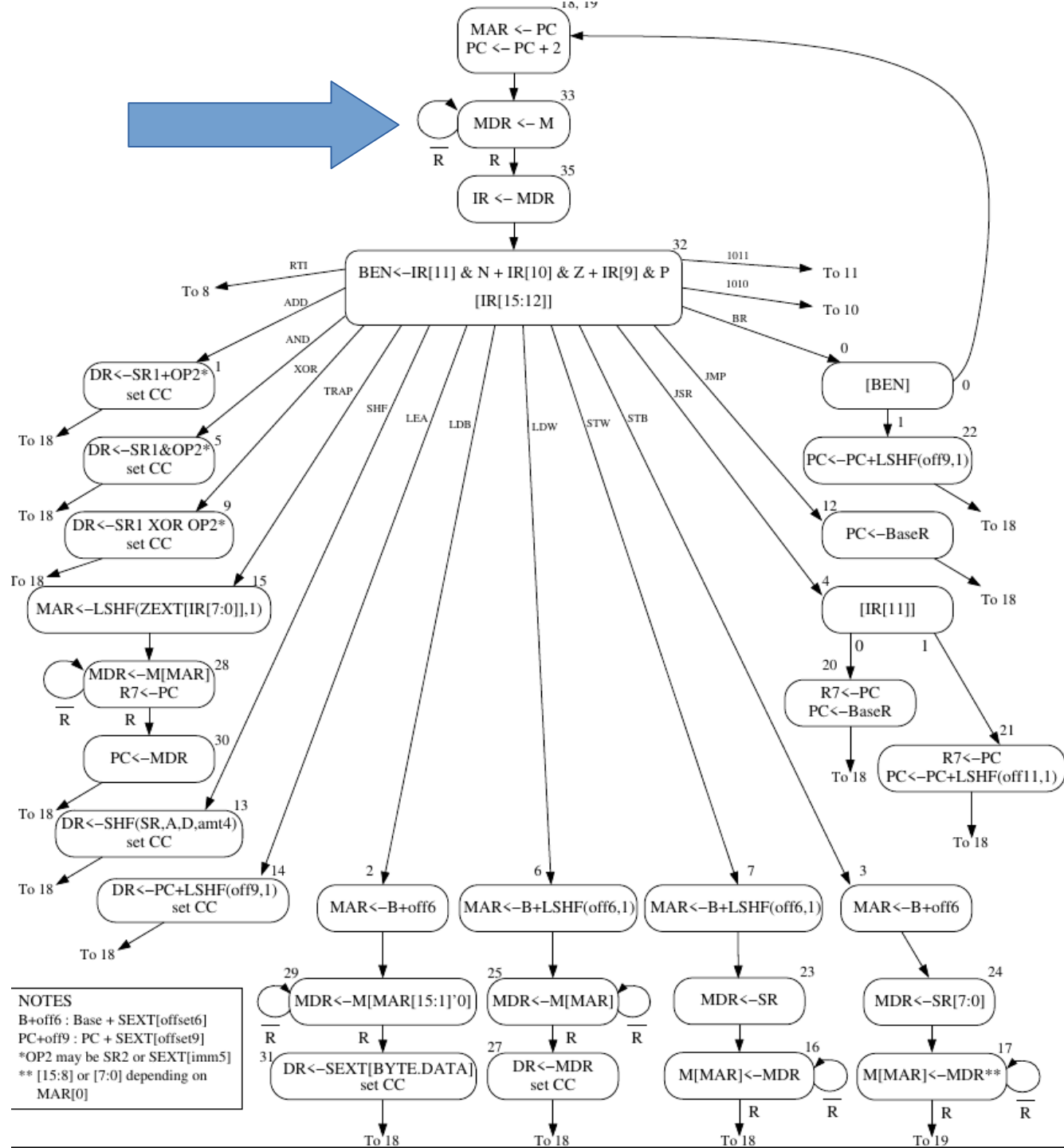
- Remember the datapath
 - One **PCSrc signal depends on processing** that happens in the datapath (**bcond** logic)

Variable-Latency Memory

- The **ready signal** (R) enables memory read/write to execute correctly
 - Example: transition from one state to another state is controlled by the R bit asserted by memory when memory data is available
- Could we have done this in a single-cycle microarchitecture?

להראות דוגמה מ-LC3b - בשקף הבא ונשים לב למעבר
משלב 33 לשלב 35.
מיקום הקובץ:

file:///home/telzur/science/Teaching/CPU/Materials/LC3b/state_machine.pdf



Aside: Memory Mapped I/O

- Address control logic determines whether the specified address of LDx and STx are to memory or I/O devices
- Correspondingly enables memory or I/O devices and sets up muxes
- This is another instance where the final control signals (e.g., MEM.EN or INMUX/2) cannot be stored in the control store
 - These signals are dependent on address

The Microsequencer: Advanced Questions

- What happens if the machine is interrupted?
- What if an instruction generates an exception?
- How can you implement a complex instruction using this control structure?

- Think REP MOVS

(Guy: REP MOVS means: repeat moving data string into another data string in memory. REP is an instruction prefix)

הערת אגב: ...עוד אודות REP MOVSB...

OllDbg - Sample_5_2_e-String_instr.exe - [CPU - main thread, module Sample_5_2_e-String_instr.exe]

File View Debug Trace Plugins Options Windows Help

00401003 PUSH ECX
00401004 PUSH EBX
00401005 PUSH ESI
00401006 PUSH EDI
00401007 MOV DWORD PTR SS:[EBP-4], 0
0040100E NOP
0040100F XOR ESI, ESI
00401011 XOR EDI, EDI
00401013 CLD
00401014 LEA ESI, [411C7C]
0040101A LEA EDI, [414290]
00401020 MOVSB
00401021 MOVSW
00401023 MOVSD
00401024 NOP
00401025 XOR ESI, ESI
00401027 XOR EDI, EDI
[00411C7F]=53495F53
[00414293]=0

Registers (3DNow!)

EAX 00413DC0 sample_5_2_e-Str
ECX 00000000
EDX BFF74FF8
EBX 00309000
ESP 0019FF18
EBP 0019FF28
ESI 00411C7F ASCII "S_IS_MOVS
EDI 00414293 sample_5_2_e-Str
EIP 00401023 sample_5_2_e-Str

C 0 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 1 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 30C000(FFF
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr 00000000 ERROR_S

Address Hex dump ASCII

00411C7F 53 5F 49 53 5F 4D 4F 56 53 5F 49 4E 53 54 52 55 S_IS_MOVS_IN
00411C8F 43 54 49 4F 4E 00 00 00 00 54 48 49 53 5F 49 53 CTION THI
00411C9F 5F 53 54 4F 53 5F 49 4E 53 54 52 55 43 54 49 4F _STOS_INSTRU
00411CAF 4E 00 00 00 00 41 42 43 44 45 46 47 48 49 4A 4B N ABCDEFG
00411CBF 4C 4D 4E 00 00 00 00 00 00 00 00 00 00 00 00 LMN Å
00411CCF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00411CDF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00411CEF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00411CFF 00 00 00 00 00 04 30 41 00 00 1E 41 00 02 00 00 Ì Ñ@ Ì OA A
00411D0F 00 08 D1 40 00 00 00 00 00 00 00 00 00 00 00 00

0019FF18 006D90D0 D m
0019FF1C 006D6DE8 èmm
0019FF20 00309000 0
0019FF24 00000000
0019FF28 0019FF70 pÿ
0019FF2C 0040136B k @ RETURN from S
0019FF30 00000001
0019FF34 006D6DE8 èmm
0019FF38 006D90D0 D m
0019FF3C 7DF58F4B K ö
0019FF40 004013F3 ó @ sample_5_2_e-
0019FF44 004013F3 é @ sample_5_2_e-

Breakpoint at Sample_5_2_e-String_instr.00401023

Windows taskbar: Type here to search, 99+, 30°C, 10:20 AM 8/7/2023, 24

Some good examples for Microprogramming

- Implement REP MOVS in a microarchitecture using microprogramming
- Guidelines: What changes, if any, do you make to the
 - state machine?
 - datapath?
 - control store?
 - microsequencer?
- Another good example: Implement unaligned word memory access using microprogramming

Table 1: REP MOVSB Instruction Variants

Instruction	Operand Size	Description
MOVSB	8-bit (Byte)	Moves one byte from DS:(SI/ESI/RSI) to ES:(DI/EDI/RDI). Index registers updated by 1.
MOVSW	16-bit (Word)	Moves one word from DS:(SI/ESI/RSI) to ES:(DI/EDI/RDI). Index registers updated by 2.
MOVSD	32-bit (Doubleword)	Moves one doubleword from DS:(SI/ESI/RSI) to ES:(DI/EDI/RDI). Index registers updated by 4.
MOVSSQ	64-bit (Quadword)	Moves one quadword from DS:(SI/ESI/RSI) to ES:(DI/EDI/RDI). Index registers updated by 8 (64-bit mode only).

Table 2: Performance Evolution of REP MOVSB (Selected Microarchitectures)

Microarchitecture	Key Performance Characteristics/Optimizations related to REP MOVSB
Pentium (P5)	Became relatively slow compared to other instructions.
Pentium Pro (P6)	Introduced "fast strings" feature, significantly improving performance, especially for larger transfers.
Ivy Bridge	Introduced ERMSB (Enhanced REP MOVSB), further optimizing byte string operations for large, aligned blocks, making it competitive with other methods in certain scenarios.
Zen 3 (AMD)	Advertises "Fast Short Rep Mov" but has shown poor performance with REP MOVSB on unaligned data, leading to potential issues with <u>memcpy</u> implementations that rely on this flag.
Modern Processors	Continue to be optimized, often involving larger internal data transfers (e.g., 16 or 64 bytes), sophisticated cache management (prefetching, non-temporal stores), and dynamic adjustments based on alignment and size. Startup costs can still be a factor for very small transfers.



Software Optimization Guide for AMD Family 16h Processors

Table 1. Typical Instruction Mappings

Instruction	Macro-ops	Micro-ops	Comments
MOV reg, [mem]	1	1: load	Fastpath single
MOV [mem], reg	1	1: store	Fastpath single
MOV [mem], imm	1	2: move-imm, store	Fastpath single
REP MOVSB [mem], [mem]	Many	Many	Microcode
ADD reg, reg	1	1: add	Fastpath single
ADD reg, [mem]	1	2: load, add	Fastpath single
ADD [mem], reg	1	2: load/store, add	Fastpath single
MOVAPD [mem], xmm	1	2: store, FP-store-data	Fastpath single
VMOVAPD [mem], ymm	2	4: 2 · {store, FP-store-data}	256b AVX Fastpath double
ADDPD xmm, xmm	1	1: addpd	Fastpath single
ADDPD xmm, [mem]	1	2: load, addpd	Fastpath single
VADDPD ymm, ymm	2	2: 2 · {addpd}	256b AVX Fastpath double
VADDPD ymm, [mem]	2	4: 2 · {load, addpd}	256b AVX Fastpath double

The Power of Abstraction

- The concept of a control store of microinstructions enables the hardware designer with a new abstraction: **microprogramming**
- The designer can translate any desired operation to a sequence of microinstructions
- All the designer needs to provide is
 - The sequence of microinstructions needed to implement the desired operation
 - The ability for the control logic to correctly sequence through the microinstructions
 - Any additional datapath control signals needed (no need if the operation can be “translated” into existing control signals)

Advantages of Microprogrammed Control

- Allows a very simple design to do powerful computation by controlling the datapath (using a sequencer)
 - High-level ISA translated into microcode (sequence of microinstructions)
 - Microcode enables a minimal datapath to emulate an ISA
 - Microinstructions can be thought of as a **user-invisible ISA (micro ISA)**
- Enables easy extensibility of the ISA
 - Can support a new instruction by changing the microcode
 - Can support complex instructions as a sequence of simple microinstructions
- If I can sequence an arbitrary instruction then I can sequence an arbitrary “program” as a microprogram sequence
 - will need some new state (e.g. loop counters) in the microcode for sequencing more elaborate programs

Update of Machine Behavior

- The ability to update/patch microcode in the field (after a processor is shipped) enables
 - Ability to add new instructions without changing the processor!
 - Ability to “fix” buggy hardware implementations
- Examples
 - **IBM 370** Model 145: microcode stored in main memory, can be updated after a reboot
 - **IBM System z**: Similar to 370/145.
 - Heller and Farrell, “[Millicode in an IBM zSeries processor](#),” IBM JR&D, May/Jul 2004.
 - **B1700** microcode can be updated while the processor is running
 - User-microprogrammable machine!

Microprogramming in IBM 360

Model	M30	M40	M50	M65
Datapath width	8 bits	16 bits	32 bits	64 bits
Microcode size	4k x 50	4k x 52	2.75k x 85	2.75k x 87
Clock cycle time (ROM)	750 ns	625 ns	500 ns	200 ns
Main memory cycle time	1500 ns	2500 ns	2000 ns	750 ns
Price (1964 \$)	\$192,000	\$216,000	\$460,000	\$1,080,000
Price (2018 \$)	\$1,560,000	\$1,760,000	\$3,720,000	\$8,720,000



Stanford

Credit: Stanford Seminar - New Golden Age for Computer Architecture - John Hennessy

<https://www.youtube.com/watch?v=bfPV4x-HrUI&t=3343s>

IBM 370/145



IBM 370 Model 145

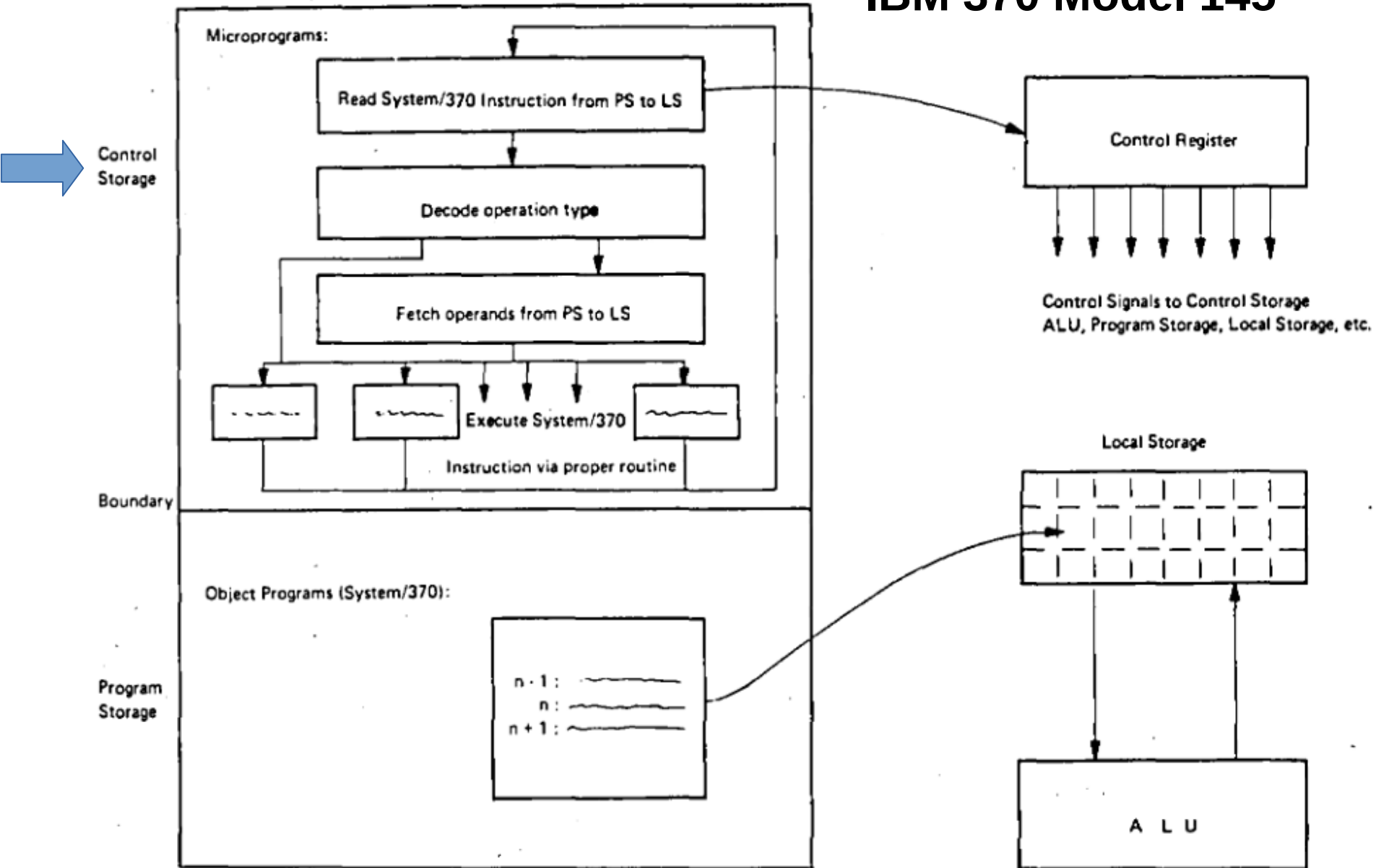


Figure 1. General microprogram flow

IS PROUD TO ANNOUNCE
THE RELEASE IN ISRAEL
OF THE BURROUGHS FOURTH GENERATION
B-1700 COMPUTER, THE YOUNGEST
MEMBER OF THE 700 SERIES.

THE B-1700 COMPUTER
sets a new record in Burroughs
advanced technology, for the
first time the cost (from
\$1,600 a month) justifies
the operation of a
"fully-fledged" computer by
medium size businesses.

THE B-1700 COMPUTER
brings you micrologic,
multiprogramming, virtual memory,
sequential and random access,
data-base facilities, all adapted
and controlled - without any
operator intervention - by the
"Master Control Program",
Burroughs' special gift to the user.

THE B-1700 COMPUTER
is oriented to run programs in B.F.O.,
Cobol, Fortran, Basic, including programs
written for other computers.

THE B-1700 COMPUTER
is fully compatible with the larger
700 series computers.

THE B-1700 COMPUTER
provides you with the unique opportunity of developing
earlier and advancing faster in infusing your Data-
processing requirements.
Please call us at No. 228114.



גאה להודיע על פתיחת המכירה
בארץ של המחשב Buroughs - 1700
(דור רביעי) האח הצעיר במשפחת
המחשבים מסדרת 700

המחשב B-1700
מהווה נקודת שיתוף חדשה בטכנולוגיה
המתקדמת של Burroughs וזו הפעם
הראשונה הצדקה העולה (הכול
מ-1600 לחודש) הפעלת מחשב
יעוצאי גם ק"י בעלים
במחיר נמוך.

המחשב B-1700
מביא לך מיקרולוגיקה, עבוד
במקביל, זיכרון נטול, גישה
שיושלים המקורות והמטרות
תקשורת, כל זה מתאם המכל
- ללא צורך בהתערבות מפעיל -
על ידי "מגבית המסך"
המרכזית, מתנת
של Burroughs למשתמש.

המחשב B-1700
מומאם לקבלת תוכנית בשפות
Basic, Fortran, Cobol, B.F.O.,
ואחרות, אפילו נכתבו
למחשבים אחרים.

המחשב B-1700
מקיים קומפטיביליות מלאה עם
מחשבי סדרה היסטורית המורשים ממנו.

המחשב B-1700
שמן לך את ההזדמנות הגדולה להקדים
את התפתחותן העצמאית במחשבי עבוד המונים.
עלול אליו לוי חסי הטלפון 228114.

לט

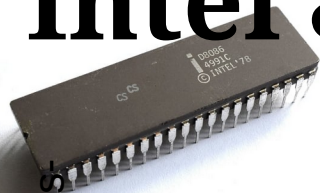
הספרייה
הלאומית
1975

במחשב הזה, בשונה
מהמקובל, ניתן היה
לראות את ה u-ISA
ואף ניתן היה לשנות
את המיקרוקוד בצורה
דינמית בעת
שהמערכת רצה.

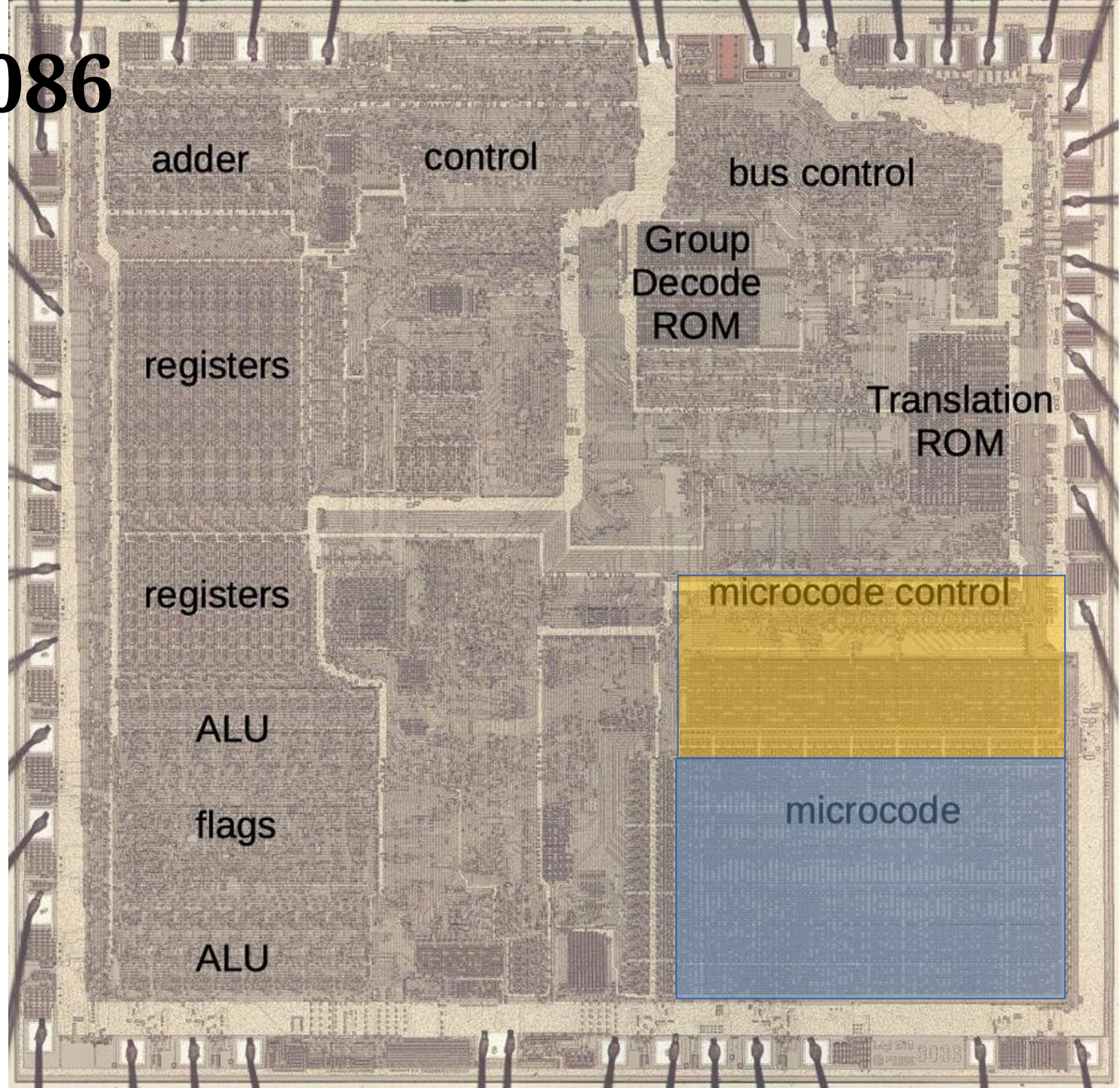
אין שקיפות כזו כיום.
זו גם אפשרות טובה
אבל יש גם צד שלילי
והוא סיכוני אבטחת
מידע.

המחשב המהפכני של Burroughs
B1700
המחשב הראשון מדור רביעי המשווק בארץ

Intel 8086



credit: <https://www.righto.com/2022/11/how-8086-processors-microcode-engine.html>



Microcode update for Intel X86-64 (in my laptop)



Type	Upgrade	Name	Old Version
Shield	✕	bluez Bluetooth tools and daemons	5.53-0ubuntu3.5
Shield	✕	php7.4 Server-side, HTML-embedded scripting language (metapackage)	7.4.3-4ubuntu2.10
Lightning bolt	✕	Linux kernel 5.13.0.51.58~20.04.31 The Linux kernel	5.13.0.48.54~20.04.30
Up arrow	✕	apt Commandline package manager	2.0.8
Up arrow	✕	intel-microcode Processor microcode firmware for Intel CPUs	3.20210608.0ubuntu0.20.04.1

Description Packages Changelog

- CVE-2020-24513, INTEL-TA-00465

-- Alex Murray <alex.murray@canonical.com> Wed, 26 May 2021 13:44:00 +0930

intel-microcode (3.20210216.0ubuntu0.20.04.1) focal-security; urgency=medium

* SECURITY UPDATE: New upstream microcode datafile 2021-02-16 (LP: #1927911)

+ Updated Microcodes:

- sig 0x00050654, pf_mask 0xb7, 2020-12-31, rev 0x2006a0a, size 36864
- sig 0x00050656, pf_mask 0xbf, 2020-12-31, rev 0x4003006, size 53248
- sig 0x00050657, pf_mask 0xbf, 2020-12-31, rev 0x5003006, size 53248
- sig 0x000706a1, pf_mask 0x01, 2020-06-09, rev 0x0034, size 74752

- CVE-2020-8695 RAPL, INTEL-TA-00389

- CVE-2020-8696 Vector Register Leakage-Active, INTEL-TA-00381

- CVE-2020-8698 Fast forward store predictor, INTEL-TA-00381

-- Alex Murray <alex.murray@canonical.com> Mon, 10 May 2021 16:42:34 +0930

intel-microcode (3.20201110.0ubuntu0.20.04.2) focal-security; urgency=medium

* SECURITY REGRESSION: Some CPUs in the Tiger Lake family sig=0x806c1 fail to boot (LP: #1903883)

- remove 06-8c-01/0x000806c1 microcode

-- Alex Murray <alex.murray@canonical.com> Thu, 12 Nov 2020 09:54:34 +1030

Next slide

Type	Upgrade	Name	Old Version
Shield	✕	bluez Bluetooth tools and daemons	5.53-0ubuntu3.5
Shield	✕	php7.4 Server-side, HTML-embedded scripting language (metapackage)	7.4.3-4ubuntu2.10
Lightning bolt	✕	Linux kernel 5.13.0.51.58~20.04.31 The Linux kernel	5.13.0.48.54~20.04.30
Up arrow	✕	apt Commandline package manager	2.0.8
Up arrow	✕	intel-microcode Processor microcode firmware for Intel CPUs	3.20210608.0ubuntu0.20.04.1

Description Packages Changelog

This package contains updated system processor microcode for Intel i686 and Intel X86-64 processors. Intel releases microcode updates to correct processor behavior as documented in the respective processor specification updates.

For AMD processors, please refer to the amd64-microcode package.

5 updates selected (12 MB)

5 updates selected (12 MB)

Microcode patch, an example

CVE-2020-8696

Improper removal of sensitive information before storage or transfer in some Intel(R) Processors may allow an authenticated user to potentially enable information disclosure via local access.

References:

<https://lists.debian.org/debian-lts-announce/2021/02/msg00007.html>

<https://nvd.nist.gov/vuln/detail/CVE-2020-8698>

<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8698>

Review: The Power of Abstraction

- The concept of a control store of microinstructions enables the hardware designer with a new abstraction:
microprogramming

הקונספט של microprogramming משכלל עוד יותר את מושג האבסטרקציה

- The designer can translate any desired operation to a sequence of microinstructions
- All the designer needs to provide is
 - The sequence of microinstructions needed to implement the desired operation
 - The ability for the control logic to correctly sequence through the microinstructions (micro-sequencer)
 - Any additional datapath elements and control signals needed (no need if the operation can be “translated” into existing control signals)

ניתן להמיר כל הוראה לסדרה של מיקרו-הוראות. כל מה שנותר הוא לקבוע את הסדר של המיקרו-הוראות. אם ההוראה ניתנת לתרגום של סיגנלים קיימים אין צורך בשינוי בדאטה-פס

Microcoded Multi-Cycle MIPS Design

גיא - הבהרות:

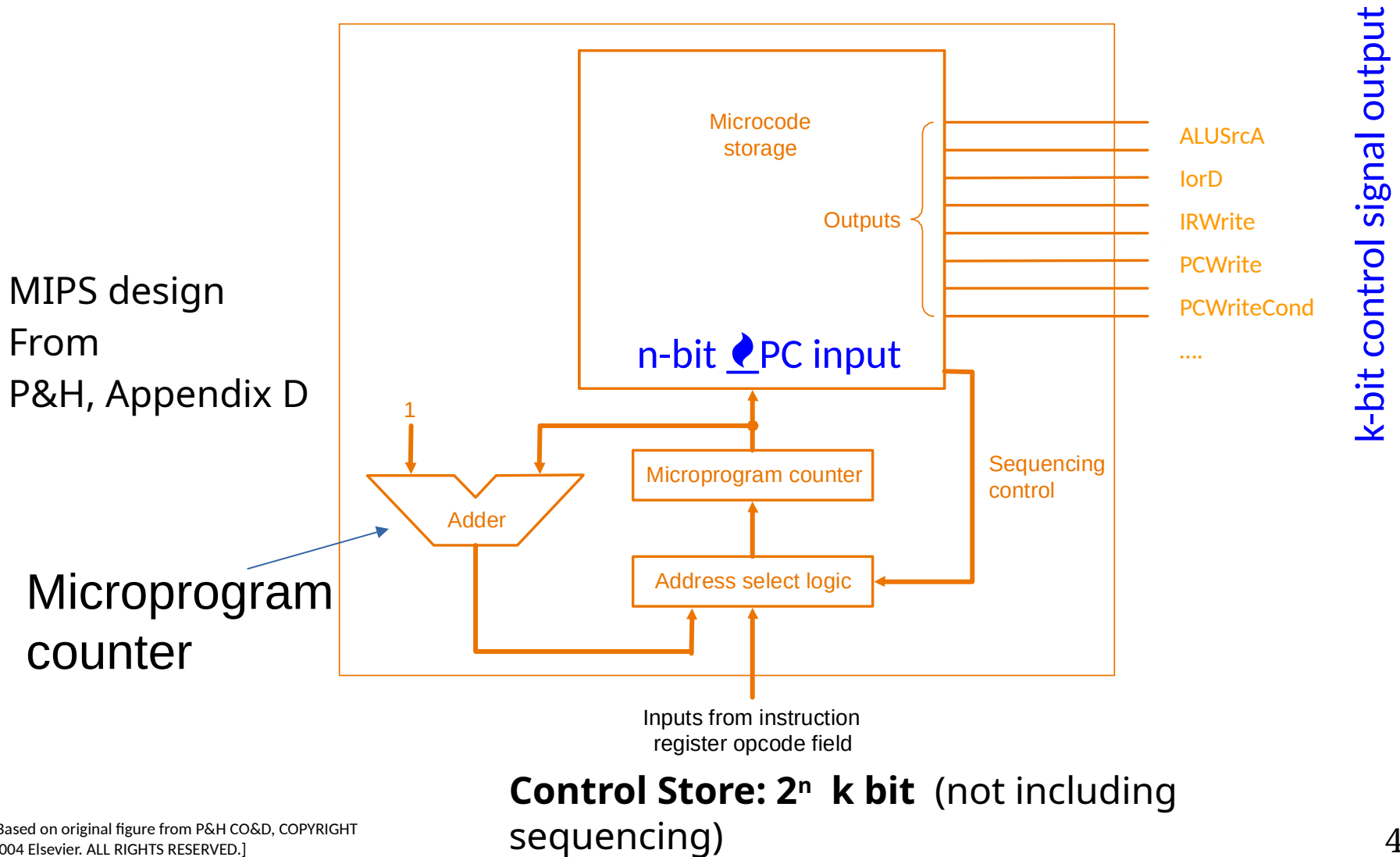
הכותרת הזו בעייתית משני טעמים:

1. אין מעבד מולטי סייקל שהוא מעבד מסחרי
2. מעבדי MIPS לא עשו שימוש במיקרו-קוד

More Micro-Programming terminology

Horizontal Microcode

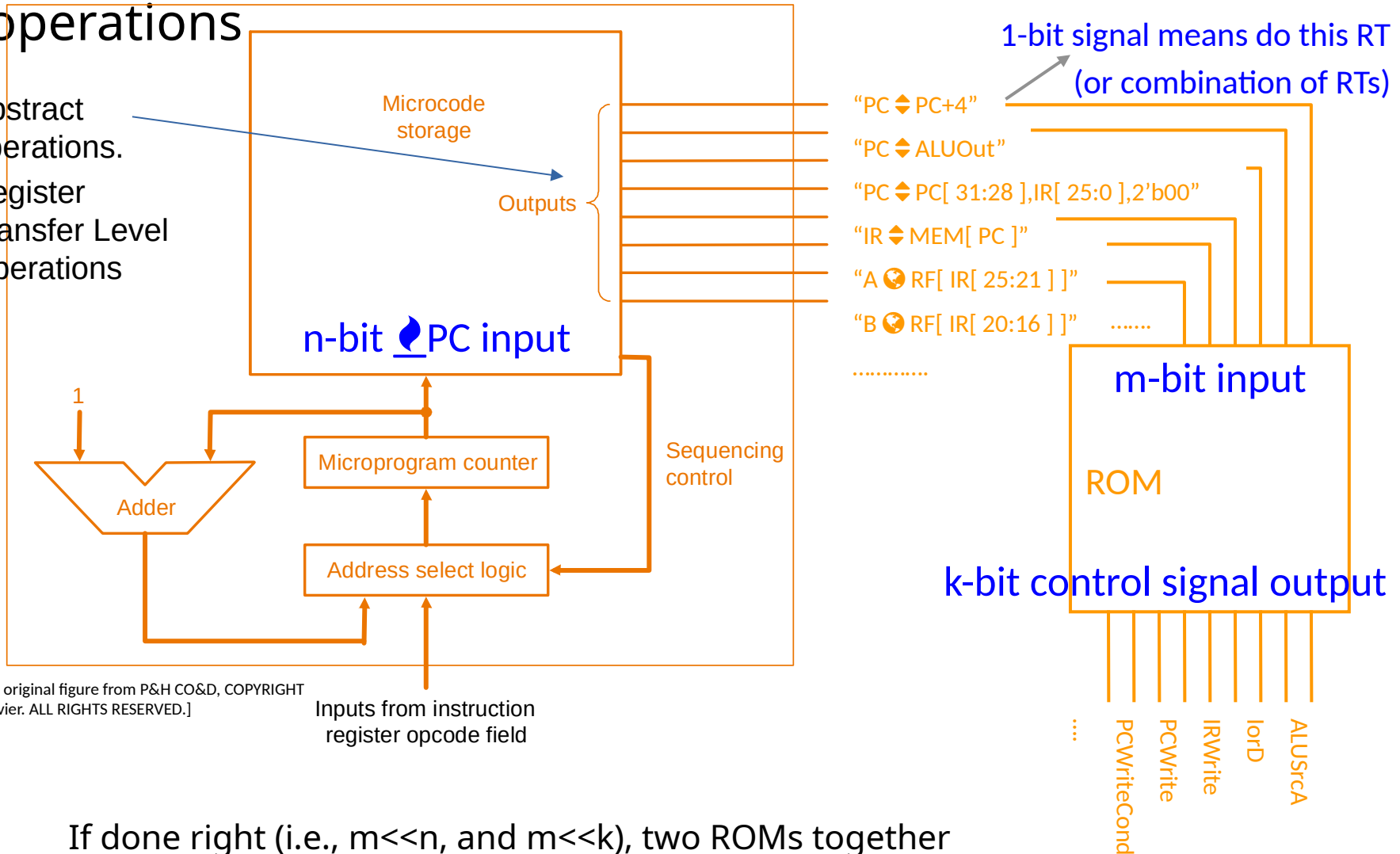
- A single control store provides the control signals



Vertical Microcode

- **Two-level control store:** the first specifies abstract operations

Abstract operations.
Register
Transfer Level
Operations



[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Inputs from instruction register opcode field

If done right (i.e., $m \ll n$, and $m \ll k$), two ROMs together ($2^n m + 2^m k$ bit) should be smaller than horizontal microcode ROM ($2^n k$ bit)

Nanocode and Millicode

- **Nanocode**: a level below traditional microcode
 - microprogrammed control for **sub-systems** (e.g., a complicated floating-point module) that acts as a slave in a microcontrolled datapath
- **Millicode**: a level above traditional microcode
 - ISA-level subroutines that can be called by the microcontroller to handle complicated operations and system functions
 - E.g., Heller and Farrell, “**Millicode in an IBM zSeries processor**,” IBM JR&D, May/Jul 2004.

מוטיבציה: ראו גם השקף הבא. זהו מאמר רשות.

- In both cases, we avoid complicating the main u-controller
- You can think of these as “**microcode**” at different levels of abstraction

Millicode in an IBM zSeries processor

Millicode in an IBM zSeries processor

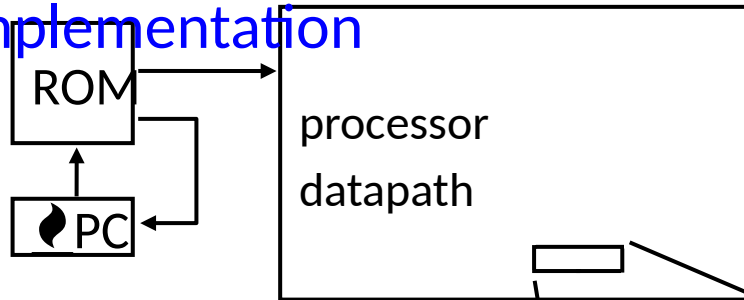
L. C. Heller
M. S. Farrell

Because of the complex architecture of the zSeries[®] processors, an internal code, called millicode, is used to implement many of the functions provided by these systems. While the hardware can execute many of the logically less complex and high-performance instructions, millicode is required to implement the more complex instructions, as well as to provide additional support functions related primarily to the central processor. This paper is a review of millicode on previous zSeries CMOS systems and also describes enhancements made to the z990 system for processing of the millicode. It specifically discusses the flexibility millicode provides to the z990 system.



Nanocode Concept Illustrated

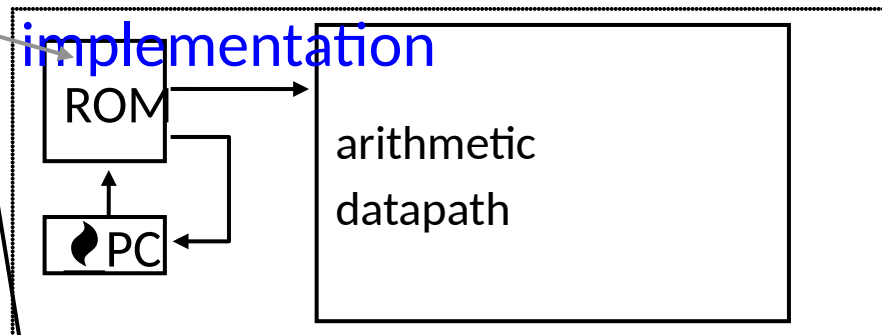
a “coded” processor implementation



תכנון יותר מודולרי

We refer to this as “nanocode” when a coded subsystem is embedded in a coded system

a “coded” FPU implementation



Microcode

credit: <https://everything2.com/title/microcode>

Microcode is often used to refer to assembly language or even raw machine language, but it isn't either of these things. **It is actually lower-level code**, hard-coded on the silicon itself, that determines how the processor responds to a given opcode and operand.

...

What is done is that a miniature ROM is on the chip. Every clock cycle, based on an internal counter, a given set of bits is read from the ROM. This set of bits covers all of the outputs needed to manipulate the internal logic as well as a few of the external signals. This set of bits is what asserts load and store lines, chip select lines, activates adders, and so forth. The idea behind RISC was to make each instruction just one line of these digital outputs.

Nanocode

credit: <https://everything2.com/title/nanocode>

As the name suggests, nanocode is at a lower-level than microcode.

Machine code is the raw instructions fed into a processor, this is first decoded by the microcode. This is a translation from the hierarchical organization of instructions that makes sense to humans, to the actual steps that need to be taken in various parts of the processor.

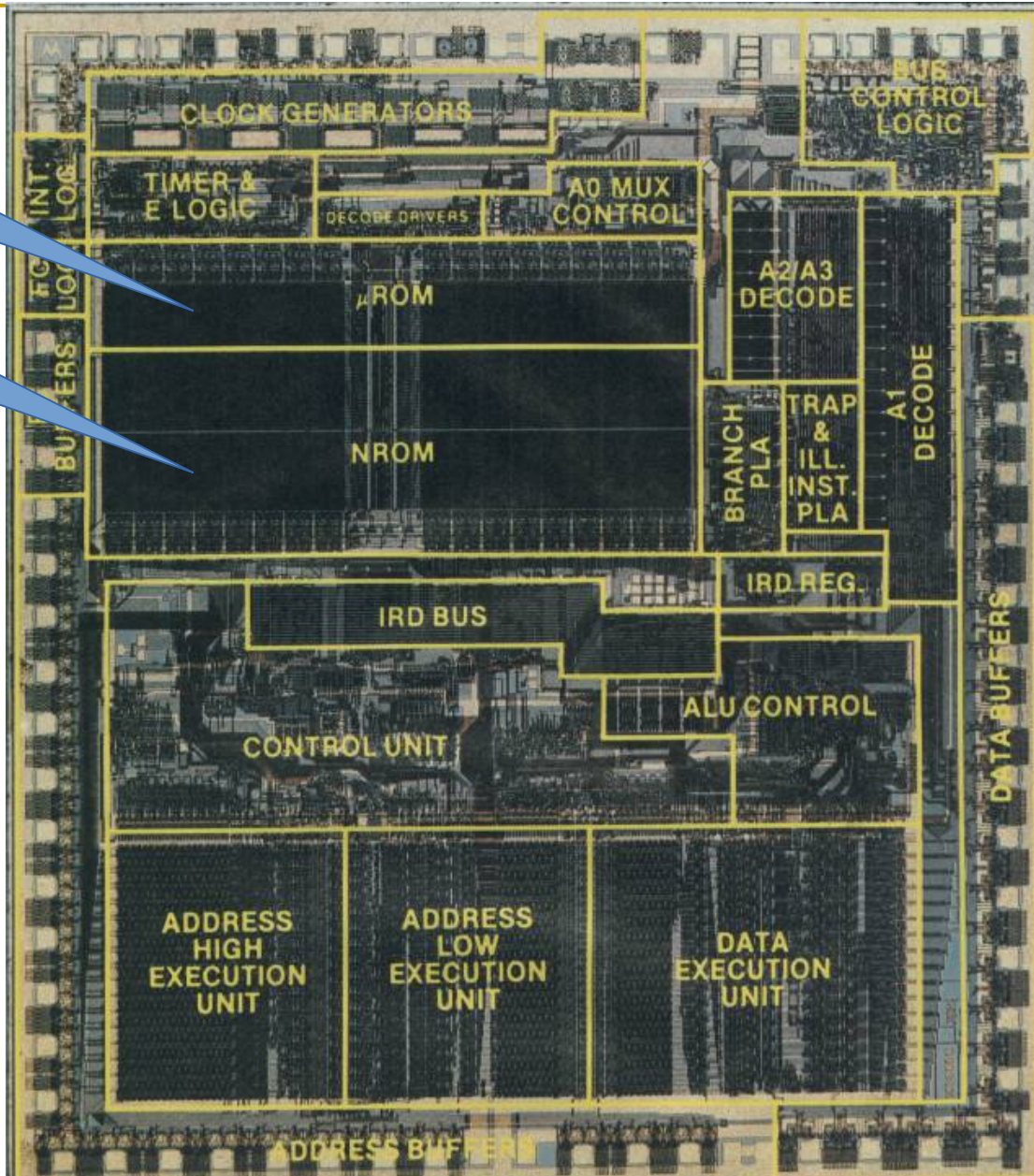
Microcode is written in a lookup table to be referenced as each instruction is executed.

Nanocode is more finely grained than microcode. It is responsible for converting the logic of the microcode to the low-level electrical signals that will cause the desired result. Nanocode is hardwired to do such things as enabling logic gates to fire their output onto interconnects, enabling the proper gates to input same, and setting the proper flags on the ALU. Very low level stuff that each individual block of a processor needs to carry out its little part of the instruction.

Motorola 68000 processor

Microcode

Nanocode



Cre
dit:
http
s://g
end
ev.s
prie
smi
nd.n
etfo
rum/
view
topi
c.ph
p?e
109
1
http:
//
ww
w.e
asy
68k.
com
/
paul
rsm/
doc/
gdp
m98

U.S. Patent 4,325,121

United States Patent [19]

Gunter et al.

[11]

4,325,121

[45]

Apr. 13, 1982

[54] **TWO-LEVEL CONTROL STORE FOR MICROPROGRAMMED DATA PROCESSOR**

[75] Inventors: **Thomas G. Gunter; Harry L. Tredennick**, both of Austin, Tex.

[73] Assignee: **Motorola, Inc.**, Schaumburg, Ill.

[21] Appl. No.: **41,135**

[22] Filed: **May 21, 1979**

Related U.S. Application Data

[63] Continuation-in-part of Ser. No. 961,796, Nov. 17, 1978.

[51] Int. Cl.³ **G06F 9/22**

[52] U.S. Cl. **364/200**

[58] Field of Search ... 364/200 MS File, 900 MS File

[56] **References Cited**

U.S. PATENT DOCUMENTS

3,886,523	5/1975	Ferguson et al.	364/200
4,008,462	2/1977	Kanda	364/200
4,155,120	5/1979	Keefer et al.	364/200
4,156,278	5/1979	Wilhite	364/200
4,156,279	5/1979	Wilhite	364/200
4,168,523	9/1979	Chari et al.	364/200

Attorney, Agent, or Firm—Anthony J. Sarli, Jr.; Jeffrey Van Myers; Robert L. King

[57] **ABSTRACT**

A data processor having an execution unit and which includes a control means having a first and a second control store. The control means has an input for receiving a control store address. In response to the received control store address, the first control store provides sequencing information at a first output for selecting the next control store address. Also, in response to the received control store address, the second control store supplies control information at a second output for controlling the execution unit. The data processor also includes means for receiving a macroinstruction and selection means responsive to the macroinstruction and to the sequencing information for generating the control store address. In a preferred embodiment, the control store address is received by both the input of the first control store and the input of the second control store. Each control word in the first control store has a unique control store address. However, a control word, in the second control store may be selected by many different control store addresses.

Summary of findings

מומלץ לעיין בערך Microcode
ב-Wikipedia

<https://en.wikipedia.org/wiki/Microcode>

Multi-Cycle vs. Single-Cycle uArch

- Advantages
- Disadvantages
- You should be very familiar with this right now

Microprogrammed vs. Hardwired Control

- Advantages
- Disadvantages

השקף הבא מתייחס לכך

Comparison



Attributes	Hardwired Control	Microprogramming Control
Speed	Fast	Slow
Cost of Implementation	More expensive	Cheaper
Flexibility	Difficult to modify	Flexible
Ability to handle complex instruction	Difficult	Easier
Decoding	Complex	Easy
Application	RISC	CISC
Instruction Set Size	Small	Large

Credit: <https://ictbyte.com/microprocessor/difference-between-hardwired-and-micro-programmed-control-unit/>

Microcode today

Architecture	Microcode	Reason
-----	-----	-----
x86	Yes (visible, updateable)	Complex variable-length instructions
MIPS	Mostly no	Simple fixed-length, RISC philosophy
RISC-V	“, Optional (cmplx cores)	Designed for simplicity
ARM	Internal μops	Performance optimization
-----	-----	-----

On my computer:

```
telzur@TUF:~$ cat /proc/cpuinfo | grep microcode
microcode : 0x6134
```

עד כאן אודות מעבד ה-Multi-Cycle והעיסוק במיקרו-קוד

361.1.4201

Computer Architecture

Pipelining I

Dr. Guy Tel-Zur

Based on slides by Prof. Onur Mutlu

Carnegie Mellon University

Spring 2015

With Dr. Guy Tel-Zur & Danny's modifications

עבור ל-7 Chapter H&H שיקף 101



Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
- Issues in OoO Execution: Load-Store Handling, ...

Recap of Last & this Lecture

- Multi-cycle and Microprogrammed Microarchitectures
 - Benefits vs. Design Principles
 - When to Generate Control Signals
 - “RISC-V” State Machine, Datapath, Control Structure
 - Microprogrammed Control: uInstruction, uSequencer, Control Store
- Microprogramming benefits
 - Power of abstraction (for the HW designer)
 - Advantages of uProgrammed Control
 - Update of Machine Behavior

Why Pipelining?

Can We Do Better?

- What limitations do you see with the multi-cycle design?
- Limited concurrency
 - Some hardware resources are idle during different phases of instruction processing cycle
 - “Fetch” logic is idle when an instruction is being “decoded” or “executed”
 - Most of the datapath is idle when a memory access is happening

Can We Use the Idle Hardware to Improve Concurrency?

- Goal: **More concurrency**  **Higher instruction throughput** (i.e., more “work” completed in one cycle)
- Idea: When an instruction is using some resources in its processing phase, **process other instructions on idle resources** not needed by that instruction
 - E.g., when an instruction is being decoded, fetch the next instruction
 - E.g., when an instruction is being executed, decode another instruction
 - E.g., when an instruction is accessing data memory (ld/st), execute the next instruction
 - E.g., when an instruction is writing its result into the register file, access data memory for the next instruction

Pipelining

Pipelining: Basic Idea

- More systematically:

- Pipeline the execution of multiple instructions

Analogy: “*Assembly line processing*” of instructions – דוגמה אנלוגית: קו ייצור

- Idea:

- Divide the instruction processing cycle into distinct “stages” of processing
- Ensure there are enough hardware resources to process one instruction in each stage
- Process a different instruction in each stage
 - Instructions consecutive in program order are processed in consecutive stages

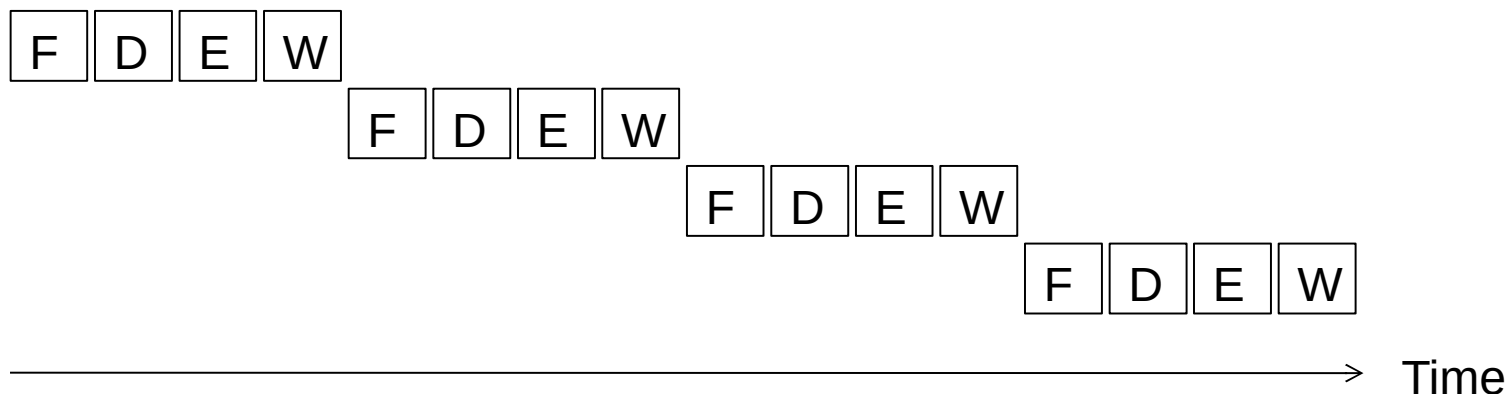
Benefit: **Increases instruction processing throughput = IPC**
(=1/CPI)

- Downside: Start thinking about this... רעיונות???

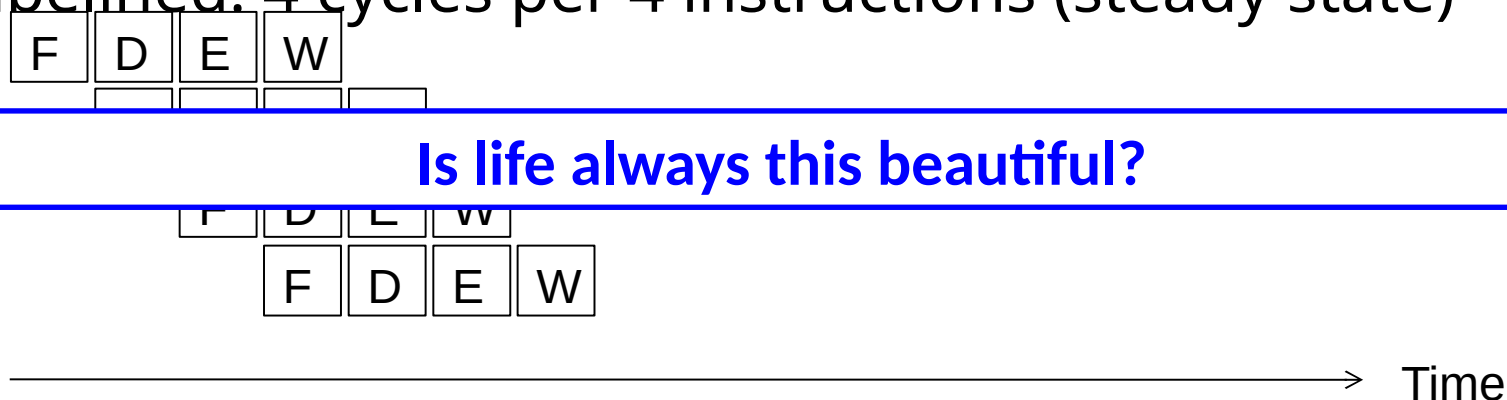
Example: Execution of Four Independent ADDs

אנימציה

- Multi-cycle: 4 cycles per instruction

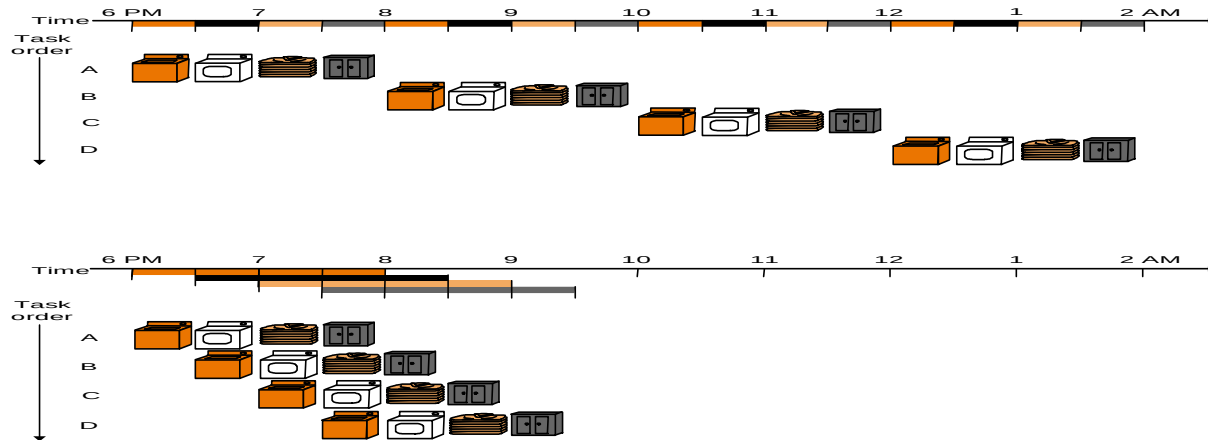


- Pipelined: 4 cycles per 4 instructions (steady state)



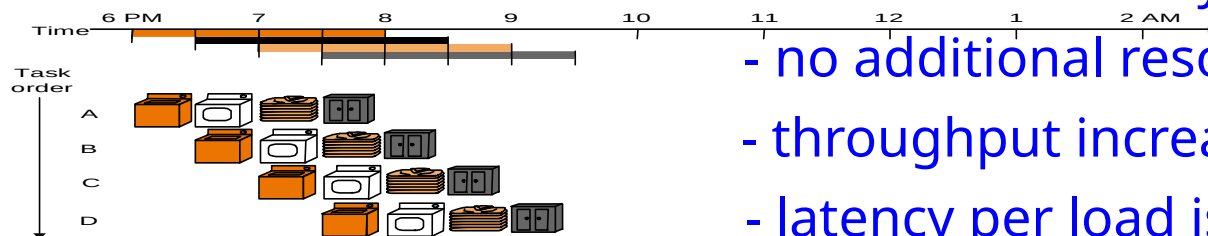
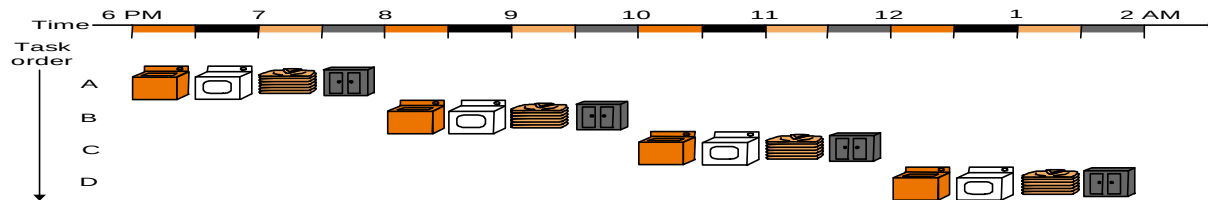
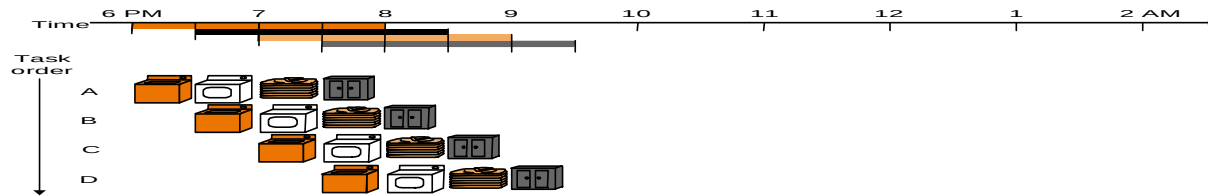
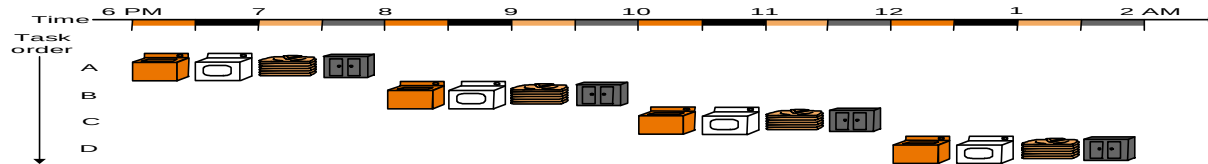
The Laundry Analogy

אנימציה



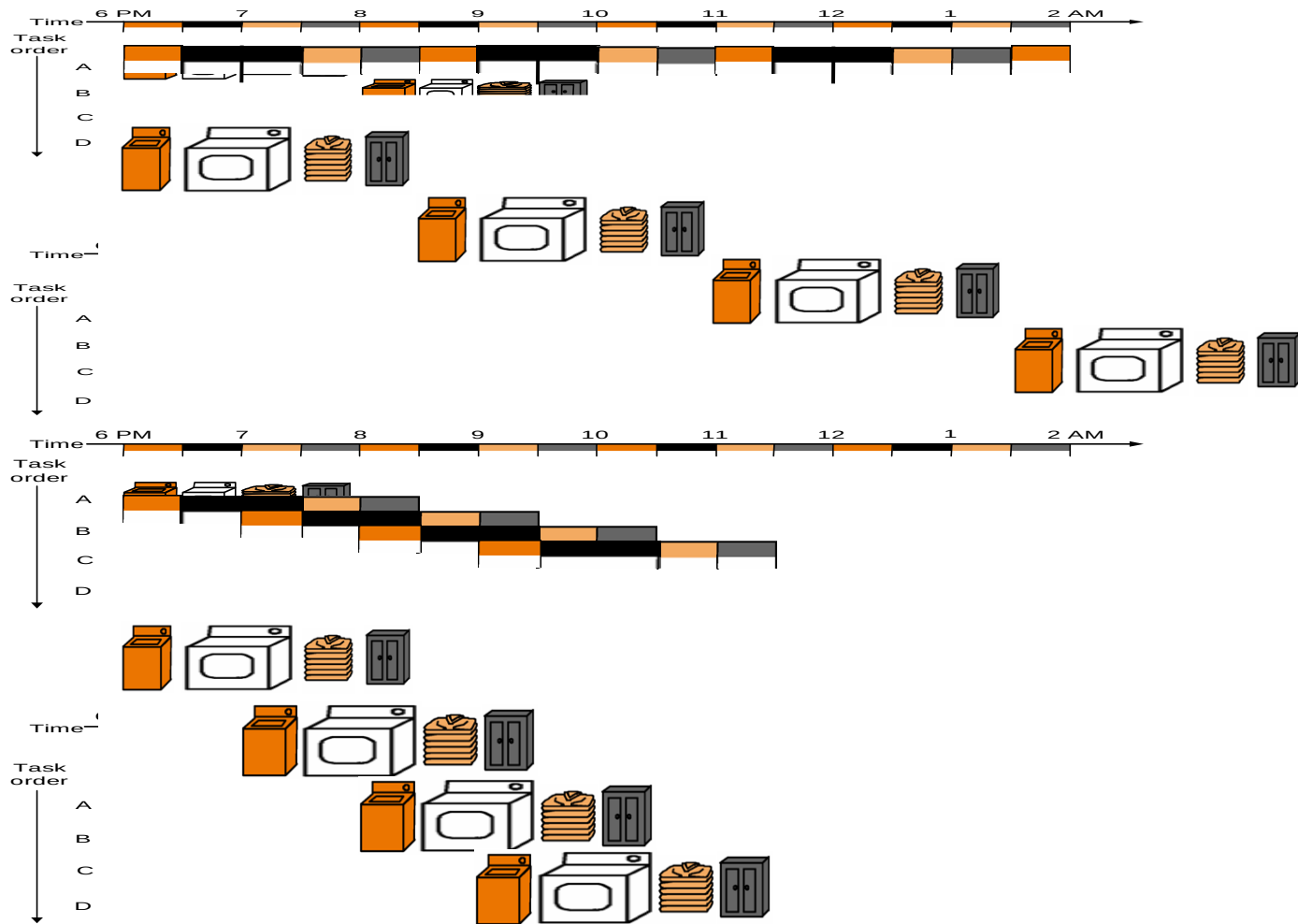
- “place one dirty load of clothes in the washer”
- “when the washer is finished, place the wet load in the dryer”
- “when the dryer is finished, take out the dry load and fold”
- “when folding is finished, ask your roommate (??) to put the clothes away”
 - steps to do a load are sequentially dependent
 - no dependence between different loads
 - different steps do not share resources

Pipelining Multiple Loads of Laundry



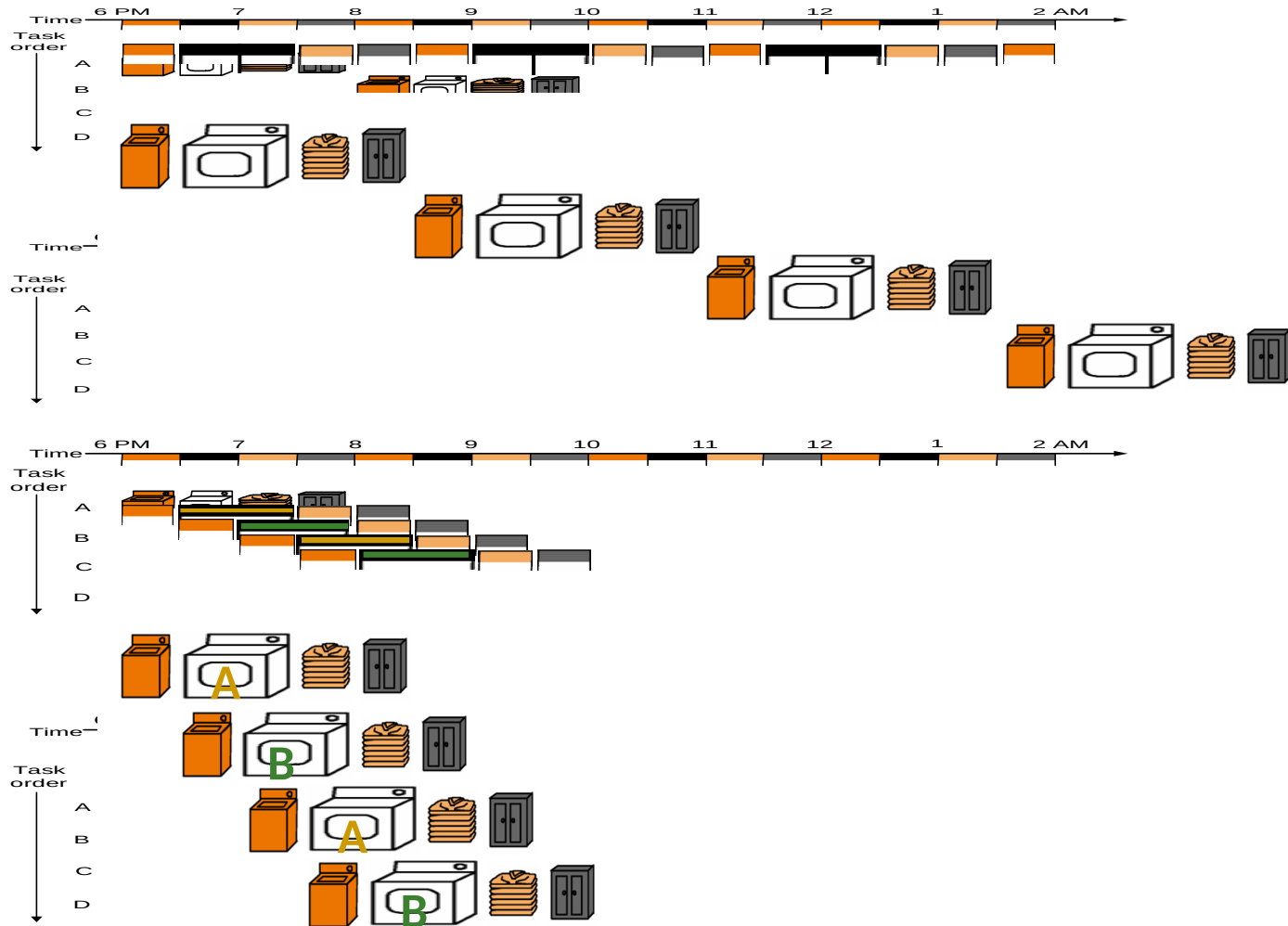
- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4
- latency per load is the same

Pipelining Multiple Loads of Laundry: In Practice



the slowest step decides throughput

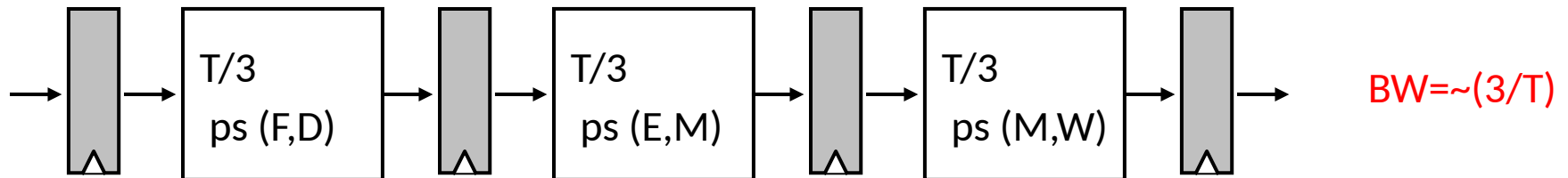
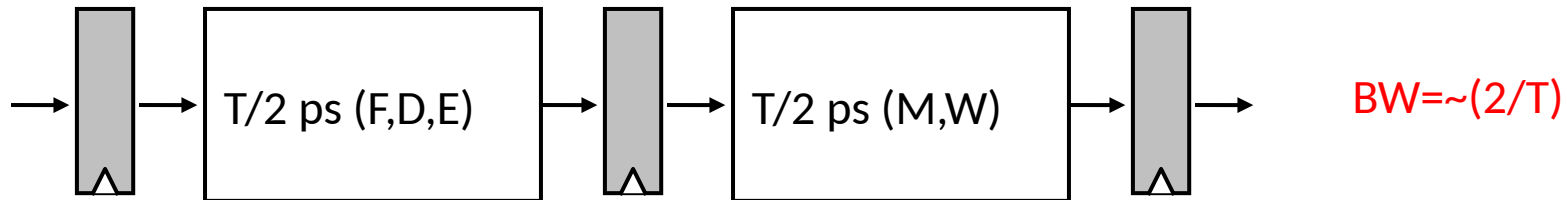
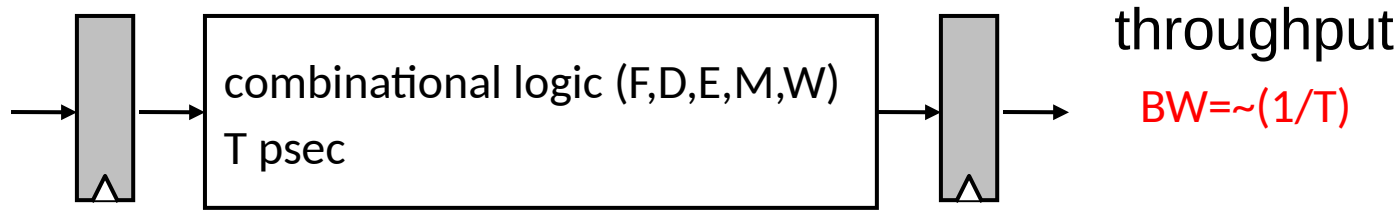
Pipelining Multiple Loads of Laundry: In Practice



throughput restored (2 loads per hour) using 2 dryers

- Goal: **Increase throughput with little increase in cost** (hardware cost, in case of instruction processing)
- **Repetition** of **identical operations**
 - The same operation is repeated on a large number of different inputs (e.g., all laundry loads go through the same steps)
- **Repetition** of **independent operations**
 - No dependencies between repeated operations
- **Uniformly partitionable suboperations** חלוקה אחידה לחבילות זמן
 - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)
- Fitting examples: automobile assembly line, doing laundry
 - What about the instruction processing “cycle”?

Ideal Pipelining



More Realistic Pipeline: Throughput

- Nonpipelined version with delay T

$$BW = 1/(T+S) \text{ where } S = \text{latch delay}$$

T =combinational logic delay

S =latch delay



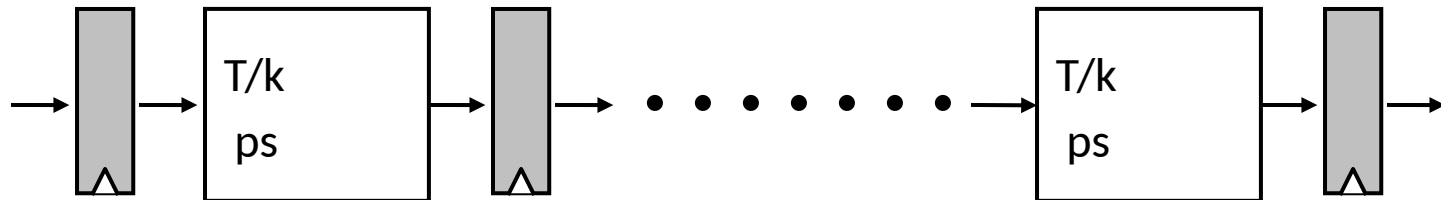
- k-stage pipelined version

$$BW_{k\text{-stage}} = 1 / (T/k + S)$$

**Latch delay reduces throughput
(switching overhead b/w stages)**

$$BW_{\max} = 1 / (1 \text{ gate delay} + S)$$

בגבול כאשר k שואף לערך הכי גדול שאפשר



More Realistic Pipeline: Cost

- Nonpipelined version with combinational cost G

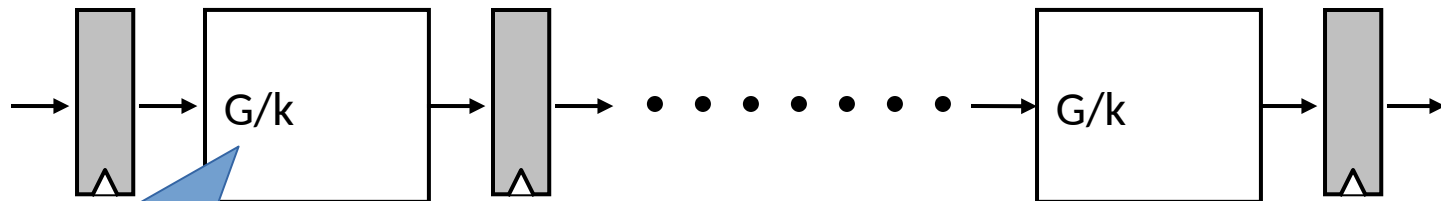
Cost = $G+L$ where L = latch cost



חסרונות:
עלות-
אנרגיה

- k -stage pipelined version

Cost _{k -stage} = $G + L * k$



Latches increase hardware cost

כמות השערים קטנה
בפקטור K

מה המספר האידאלי של דרגות מיקבול?

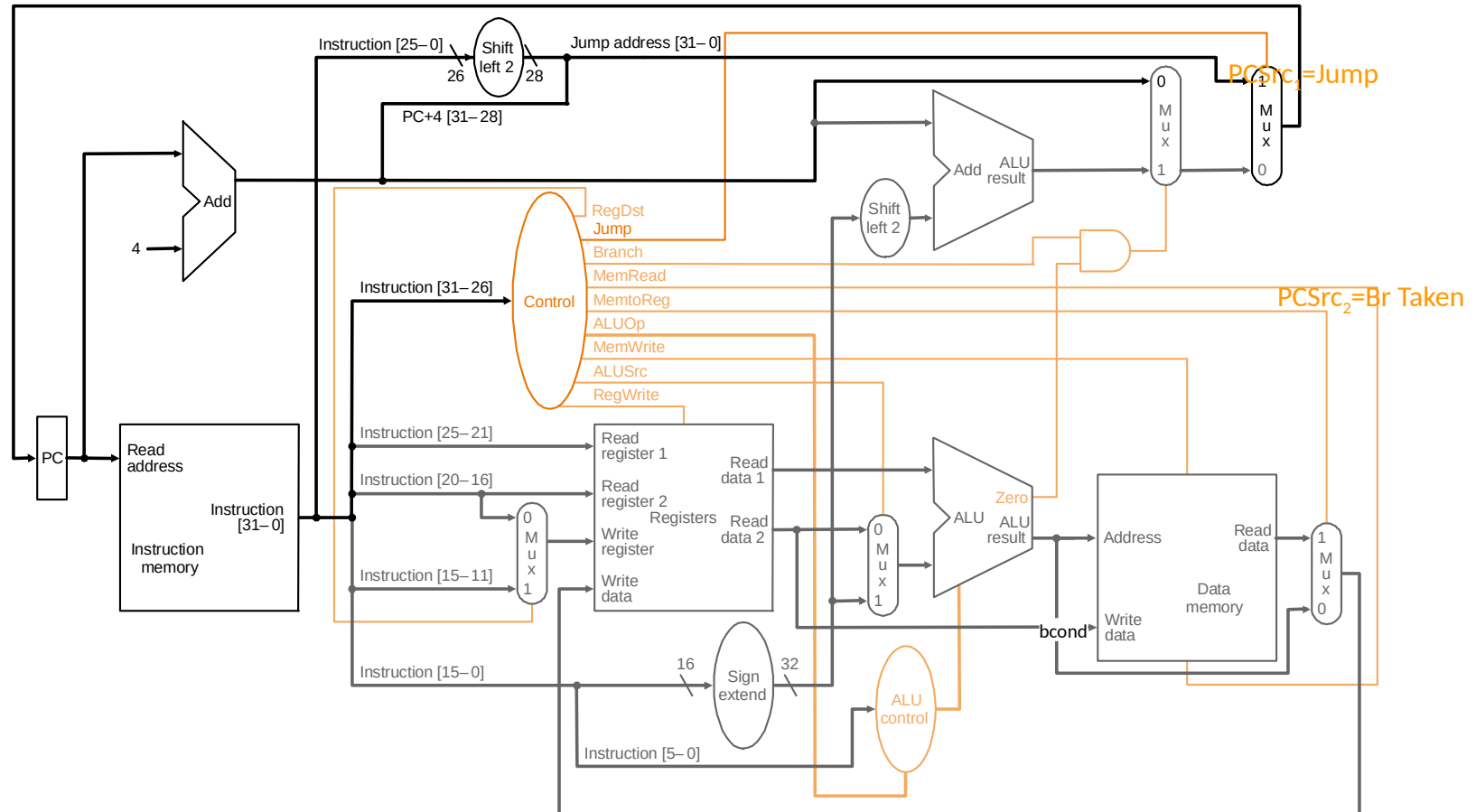
Pipelining Instruction Processing

Remember: The Instruction Processing Cycle

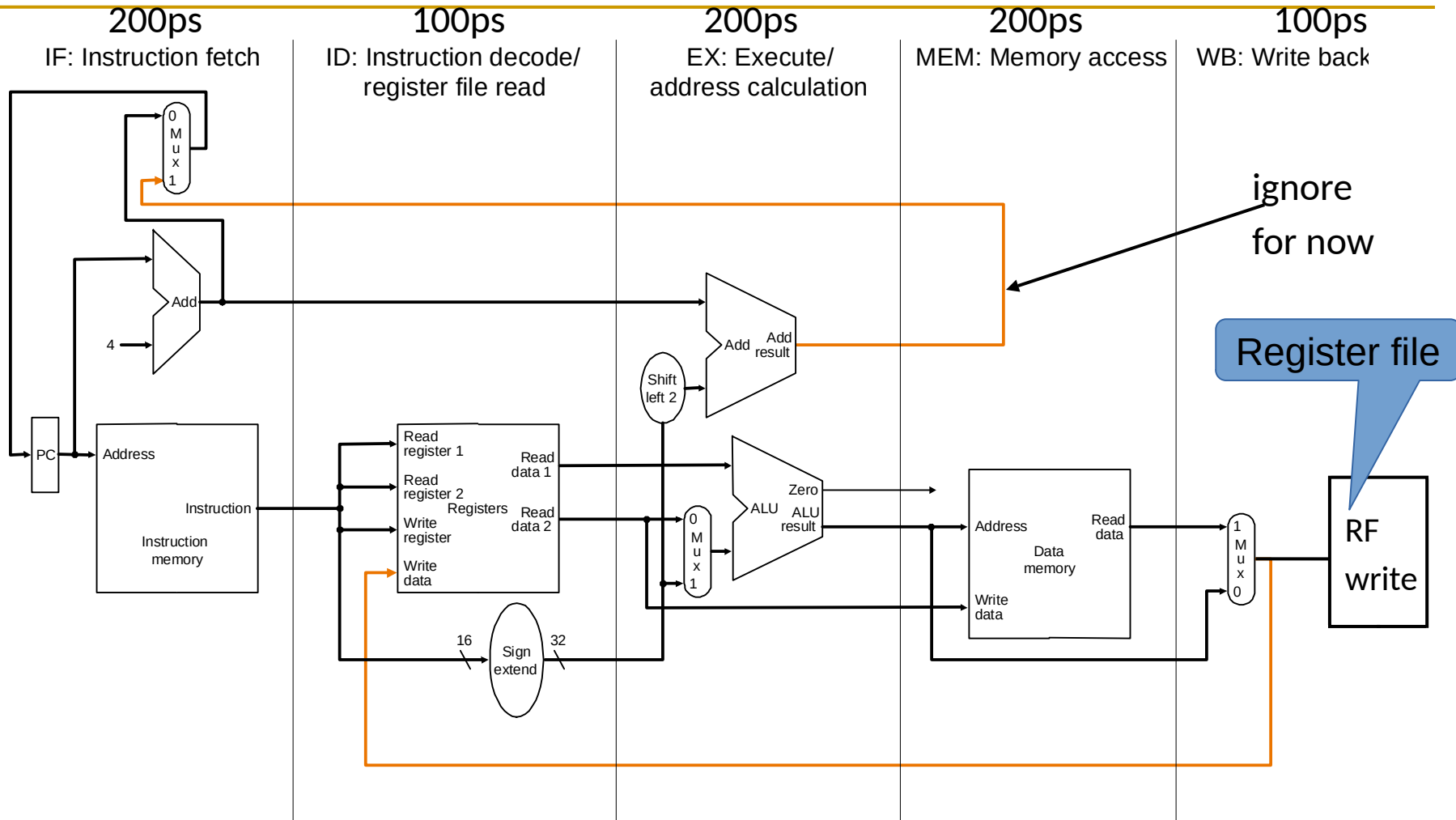
- 
1. Instruction fetch (IF)
 2. Instruction decode and register operand fetch (ID/RF)
 3. Execute/Evaluate memory address (EX/AG)
 4. Memory operand fetch (MEM)
 5. Store/writeback result (WB)

AG=Address Generation

Remember the Single-Cycle Uarch



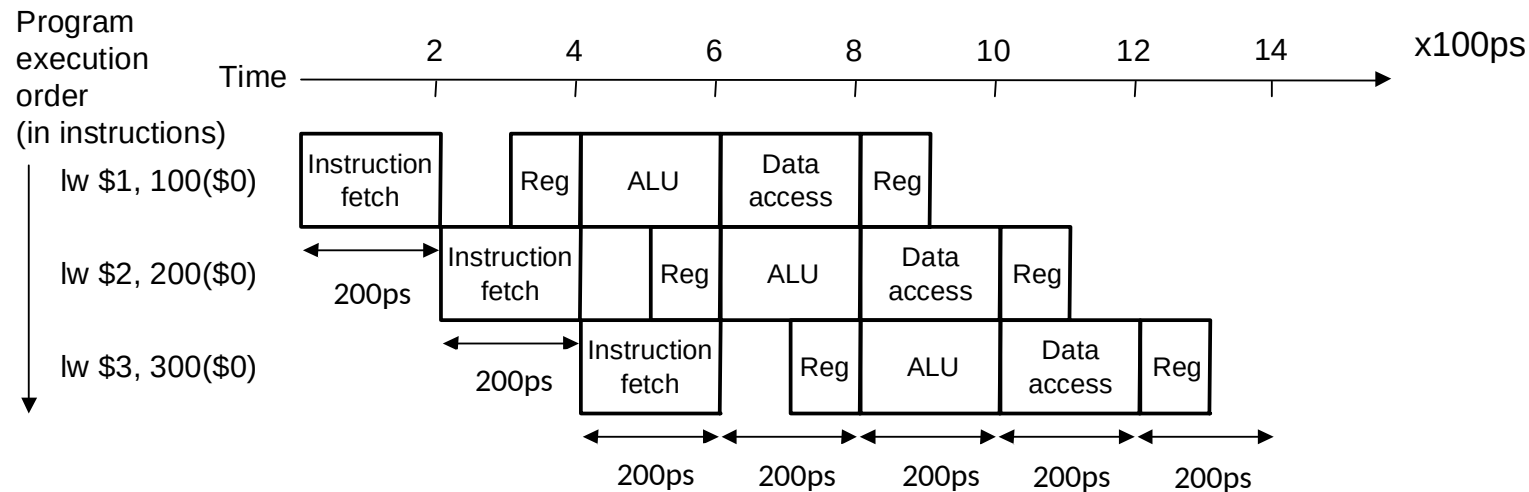
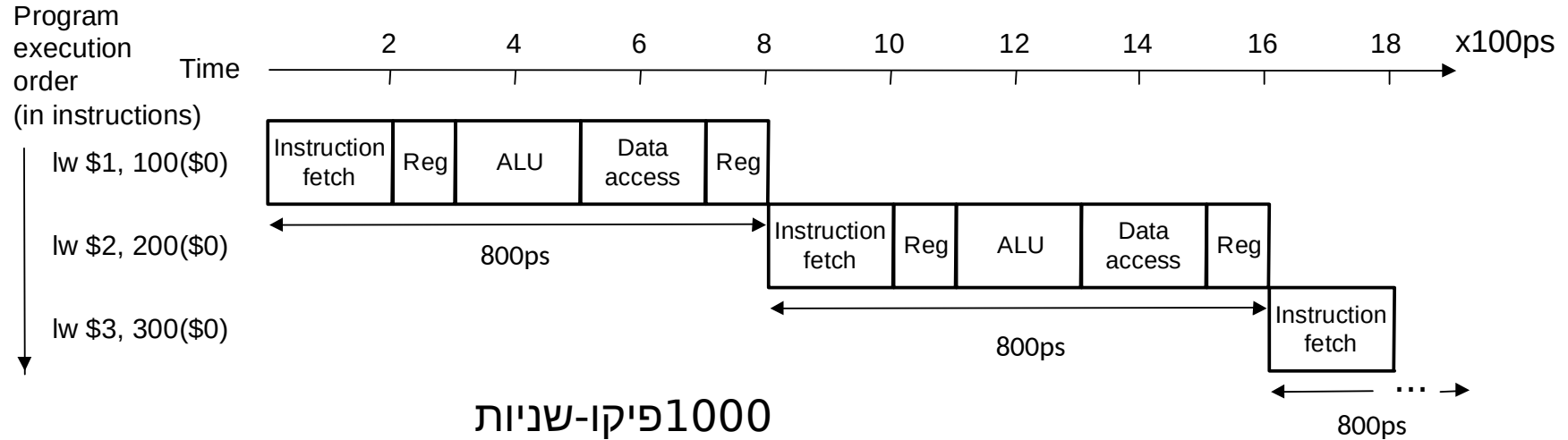
Dividing Into Stages



Is this the correct partitioning?

Why not 4 or 6 stages? Why not different boundaries?

Instruction Pipeline Throughput

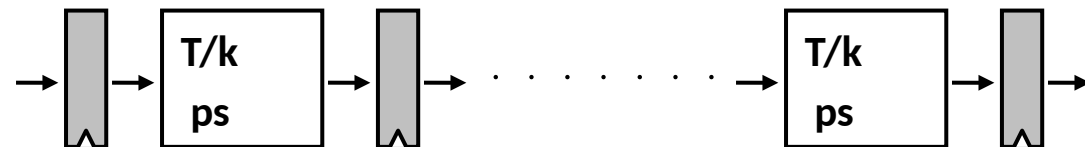
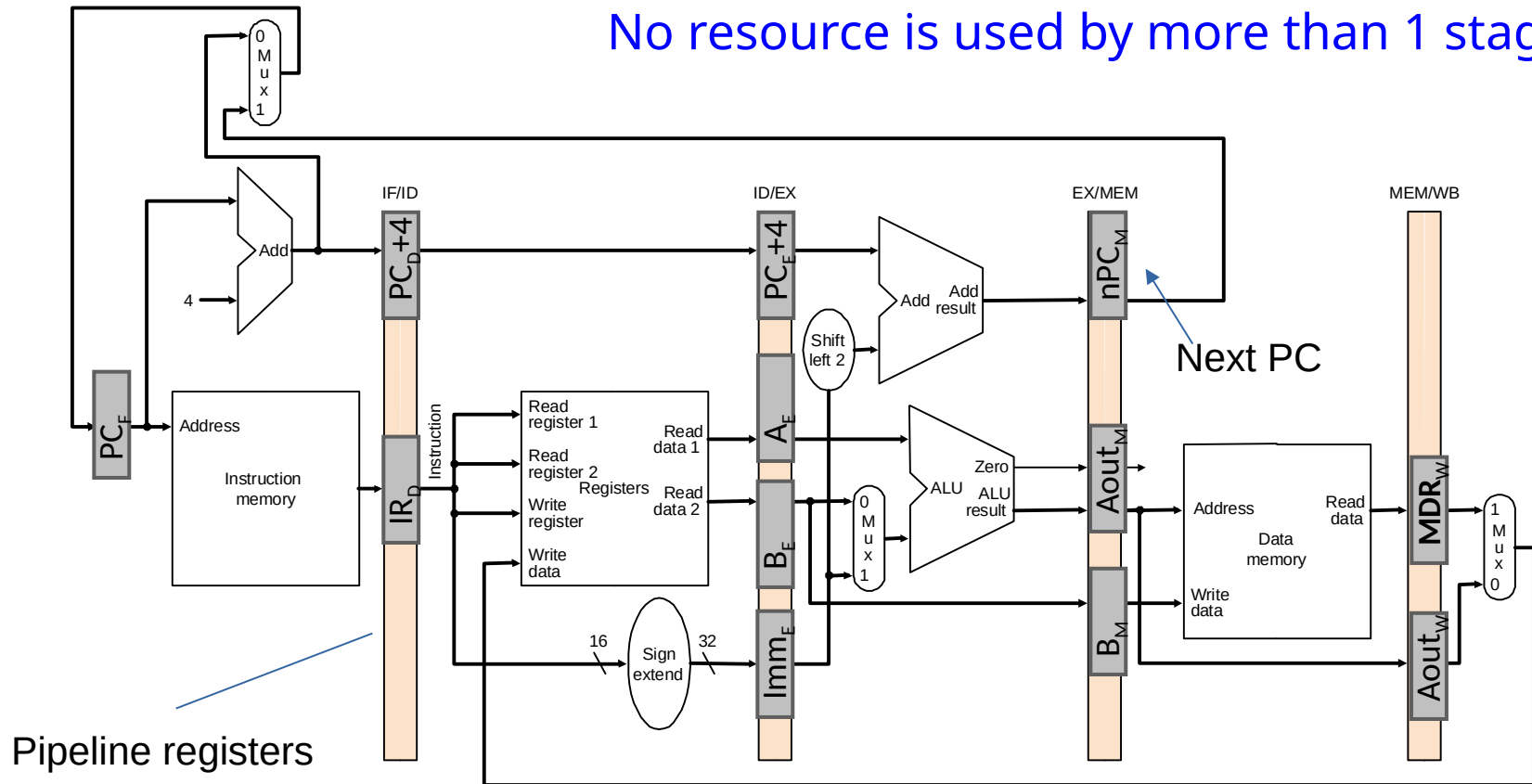


5-stage speedup is 4, not 5 as predicted by the ideal model. Why?

תשובה: בתרשים העליון: $BW=1/800$, בתרשים התחתון: $BW=1/200$. ההאצה היא היחס בין המצב הישן למצב החדש והמשופר: 4 (לא לחשב בטעות)

Enabling Pipelined Processing: Pipeline Registers

No resource is used by more than 1 stage!

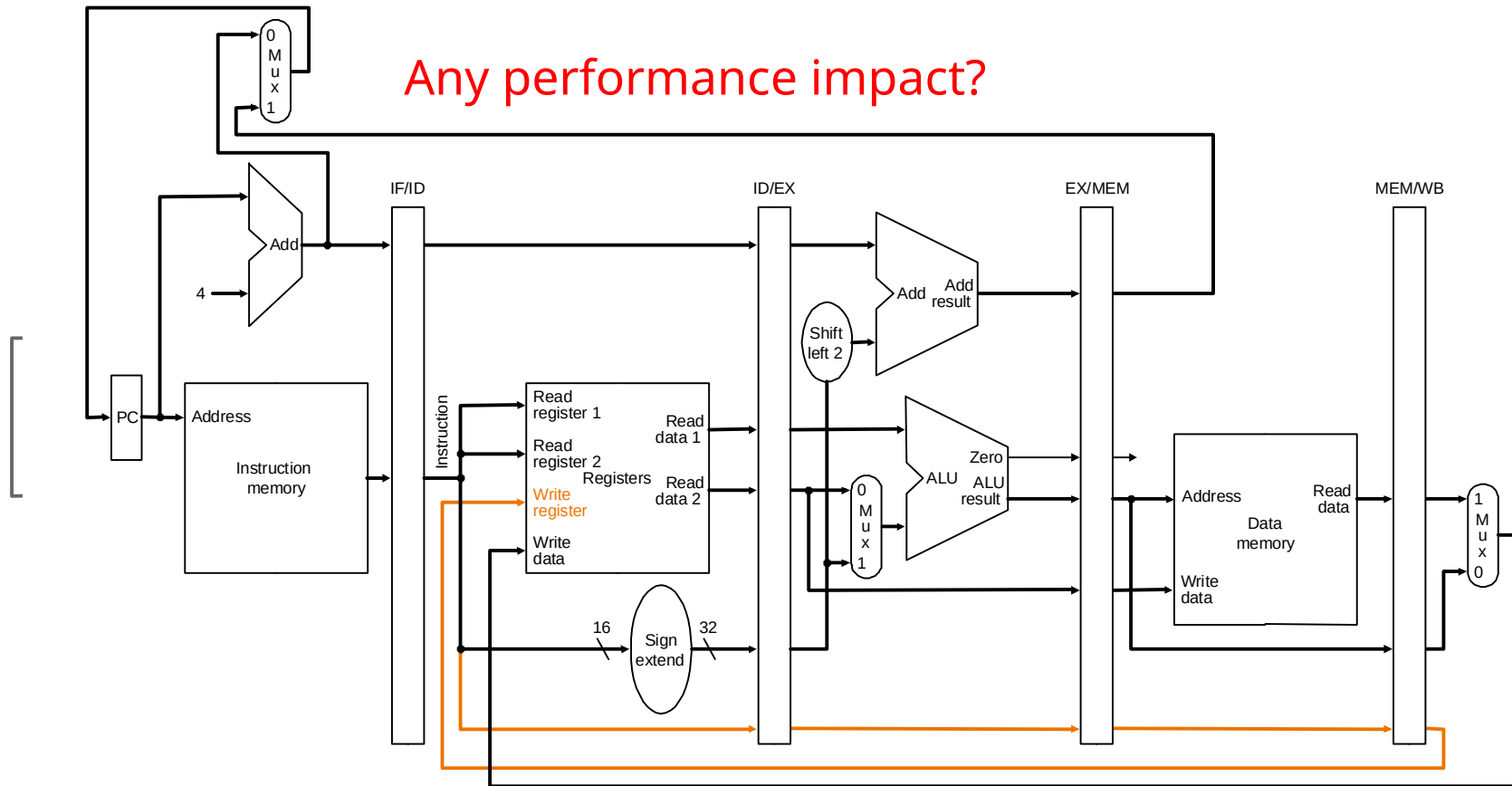


Pipelined Operation Example

אנימציה

All instruction classes must follow the same path and timing through the pipeline stages.

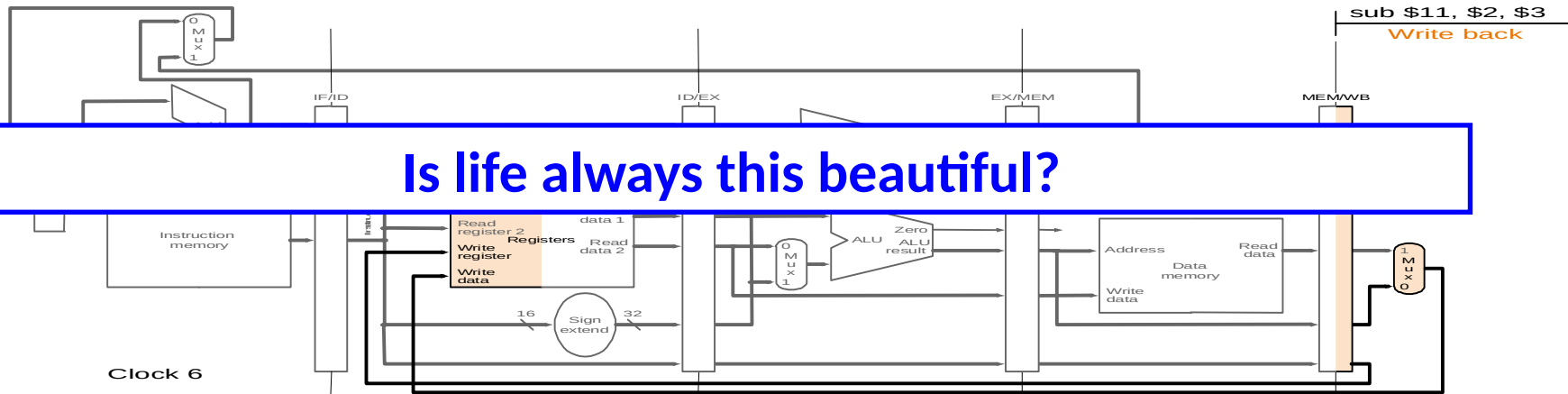
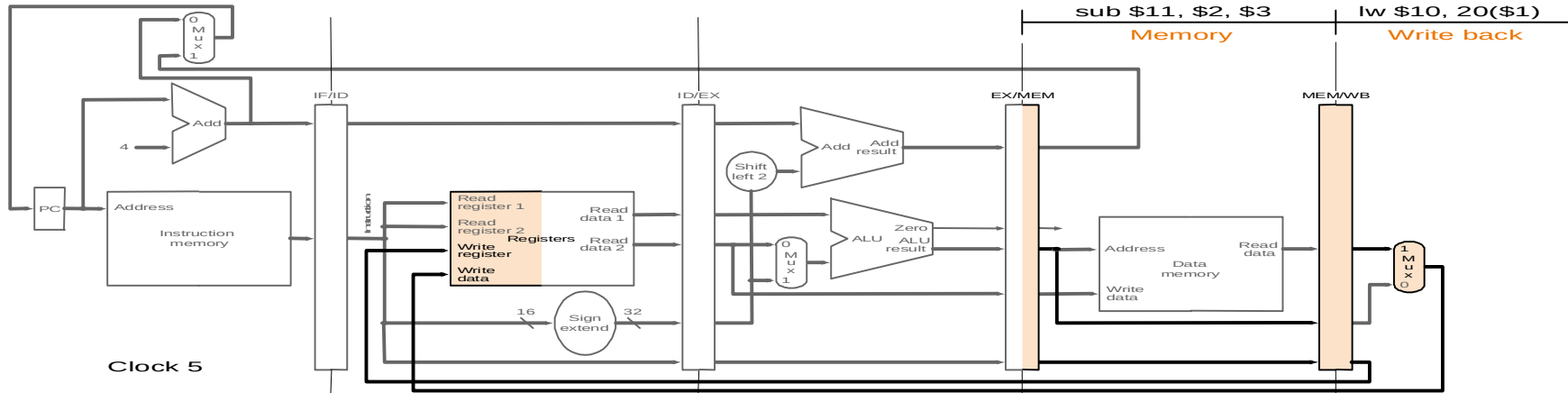
Any performance impact?



971
9710845
Figure C

Pipelined Operation Example

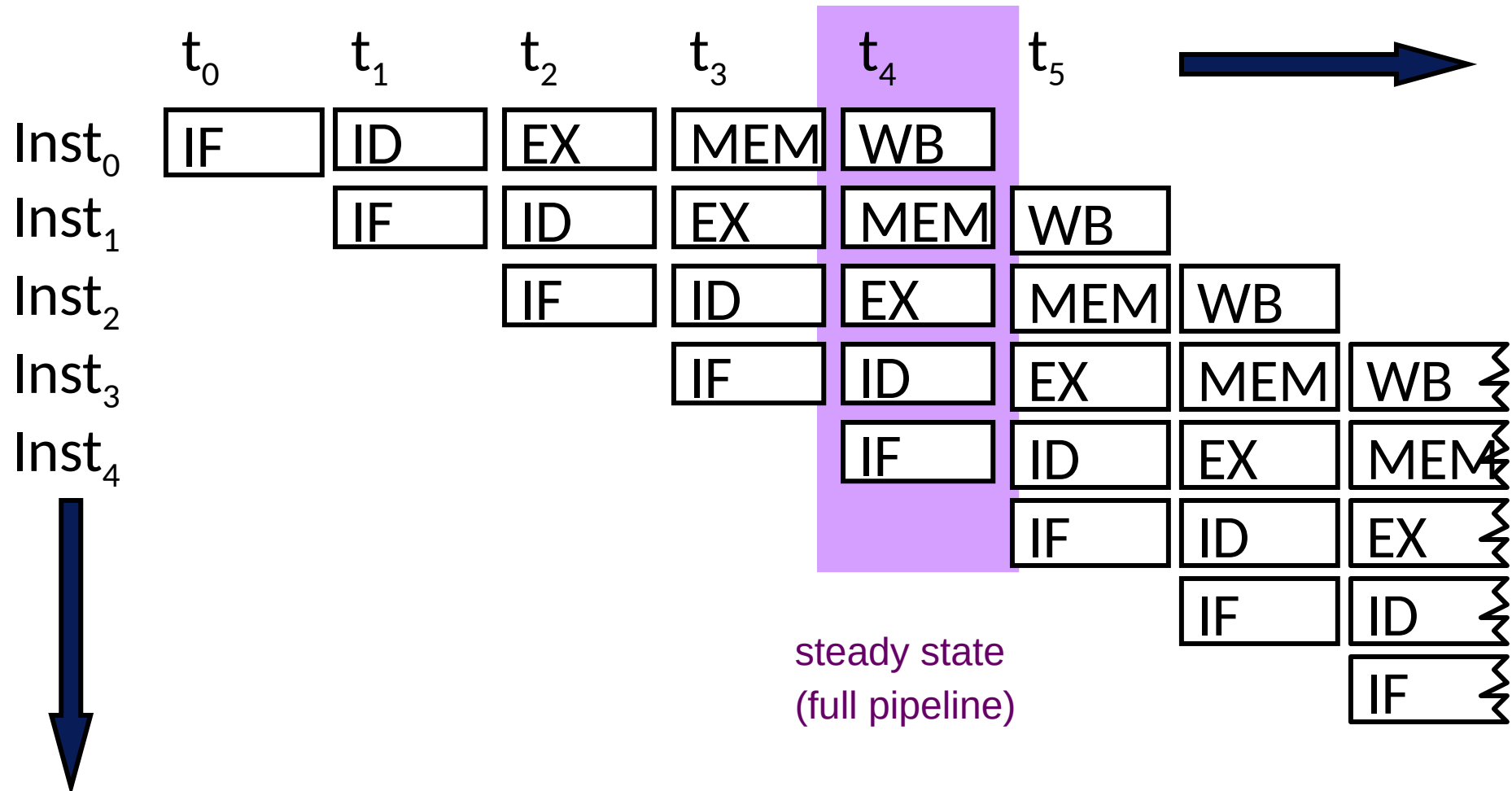
אנימציה



Is life always this beautiful?

Illustrating Pipeline Operation: Operation View

אנימציה

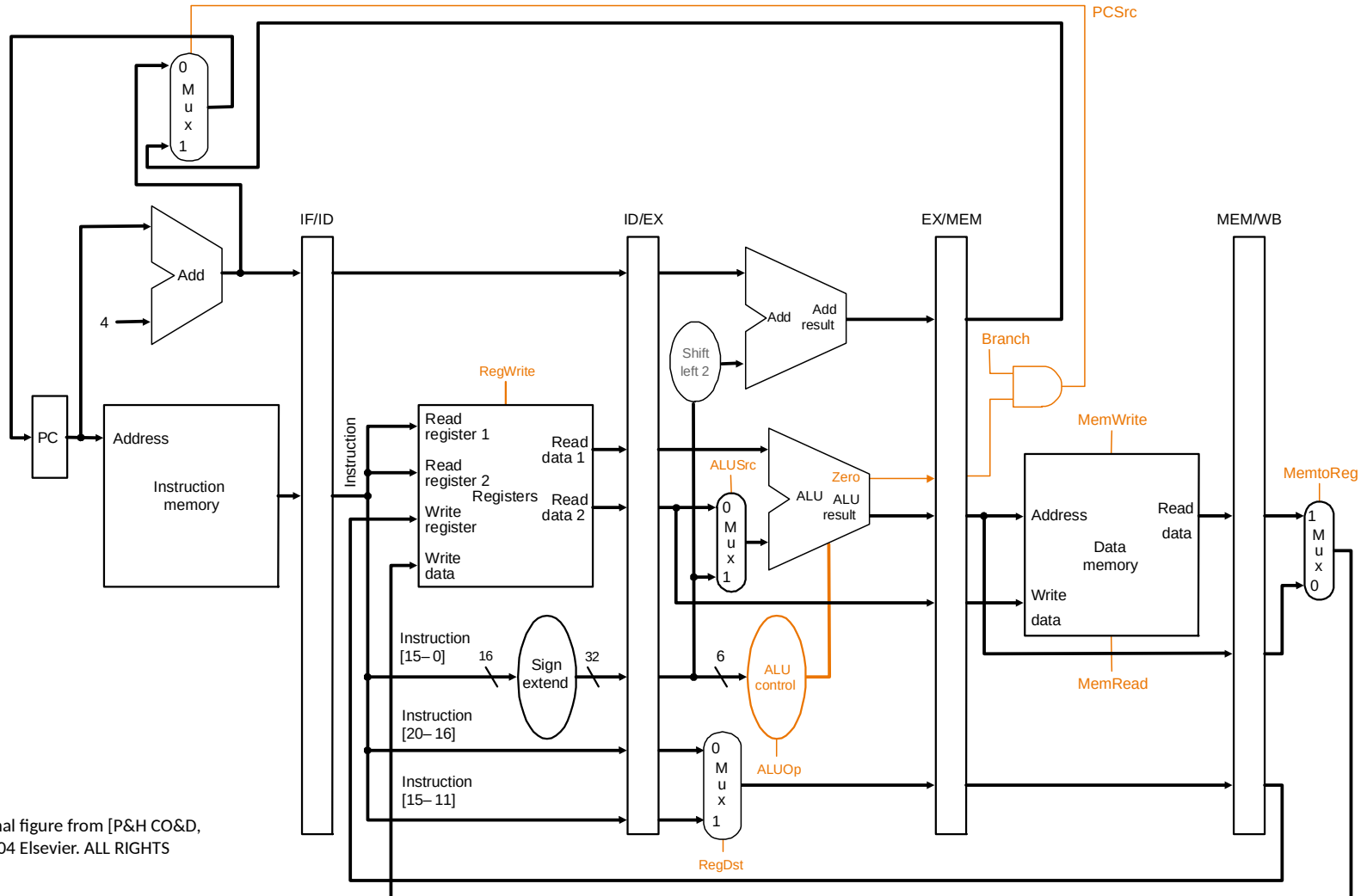


Illustrating Pipeline Operation: Resource View

אנימציה

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}
ID		I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9
EX			I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
MEM				I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
WB					I_0	I_1	I_2	I_3	I_4	I_5	I_6

Control Points in a Pipeline



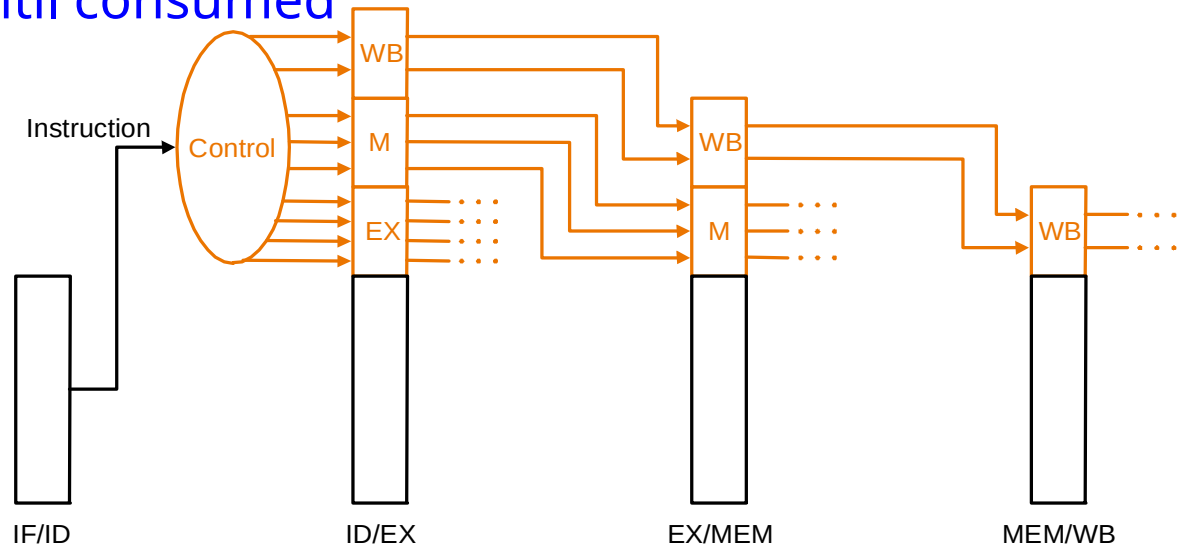
Based on original figure from [P&H CO&D,
COPYRIGHT 2004 Elsevier. ALL RIGHTS
RESERVED.]

Identical set of control points as the single-cycle datapath!!

Control Signals in a Pipeline

- For a given instruction
 - same control signals as single-cycle, but
 - control signals required at different cycles, depending on stage
 - ⇒ Option 1: decode once using the same logic as single-cycle and buffer signals until consumed

Option 1: Decode once and propagate

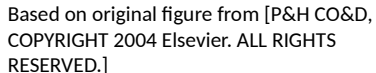


Option 2: Decode on demand

- ⇒ Option 2: carry relevant “instruction word/field” down the pipeline and decode locally within each or in a previous stage

This option may reduce the cost of the latches – reduce the number of control latches

Which one is better?



Remember: An Ideal Pipeline - החזון

- Goal: **Increase throughput with little increase in cost**
(hardware cost, in case of instruction processing) הגדלת כמות החישובים ללא תוספת עלות
- Repetition of **identical operations** חזרתיות על צעדים קבועים
 - The same operation is repeated on a large number of different inputs (e.g., all laundry loads go through the same steps)
- Repetition of **independent operations** יכולת ביצוע פעולות שונות
 - No dependencies between repeated operations אי תלות בין הפעולות
- **Uniformly partitionable suboperations** השהיות אחידות
 - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)
- Fitting examples: automobile assembly line, doing laundry
 - **What about the instruction processing "cycle"?**

Instruction Pipeline: Not An Ideal Pipeline

- Identical operations ... NOT! המציאות
 - ▲ different instructions ✉ not all need the same stages
 - Forcing different instructions to go through the same pipe stages
 - ✉ external fragmentation (some pipe stages idle for some instructions)
- Uniform suboperations ... NOT!
 - ▲ different pipeline stages ✉ not the same latency
 - Need to force each stage to be controlled by the same clock
 - ✉ internal fragmentation (some pipe stages are too fast but all take the same clock cycle time)
- Independent operations ... NOT!
 - ▲ instructions are not independent of each other
 - Need to detect and resolve inter-instruction dependencies to ensure the pipeline provides correct results
 - ✉ pipeline stalls (pipeline is not always moving)

Pipelining continues
in next lecture slides...