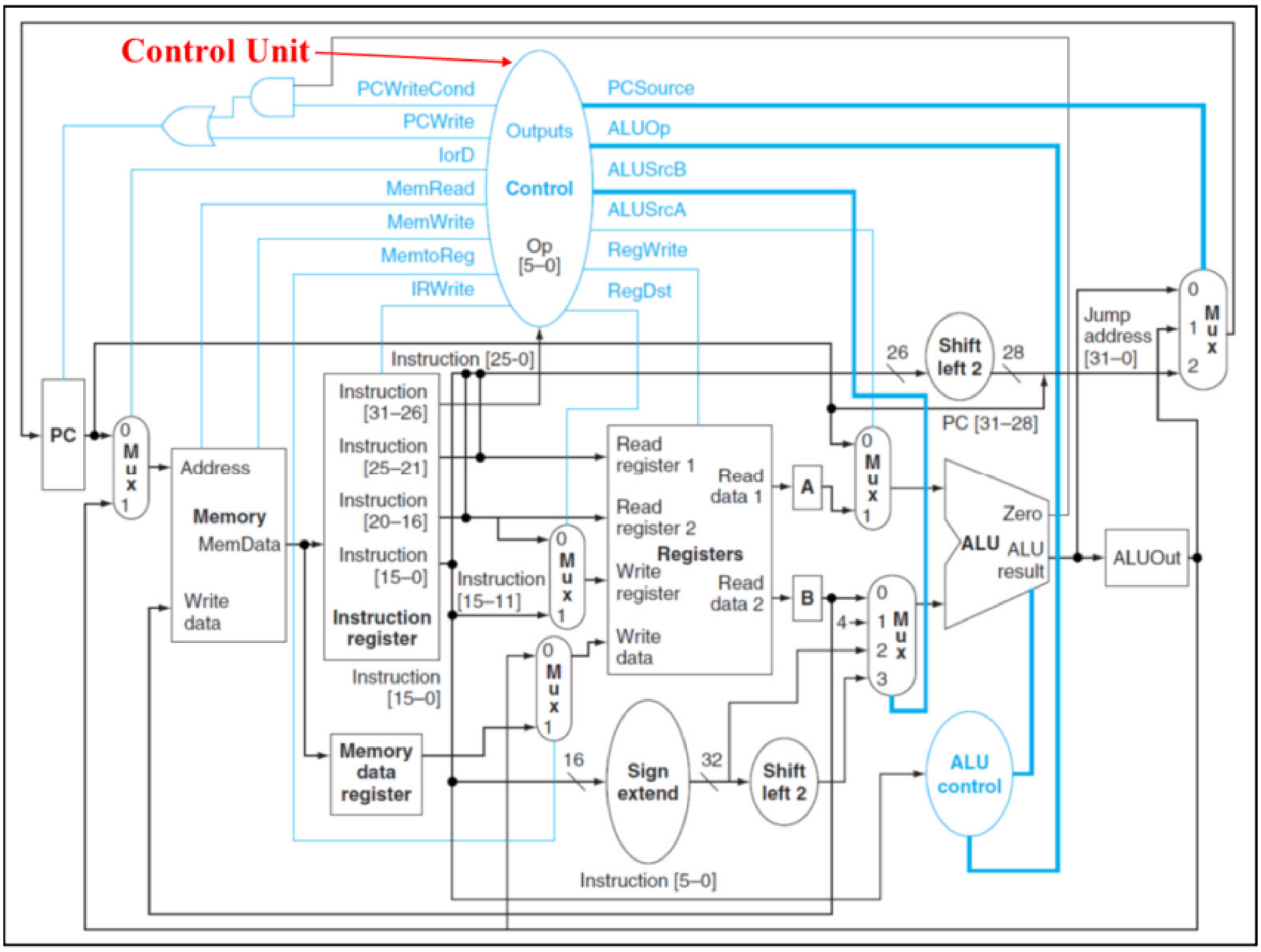
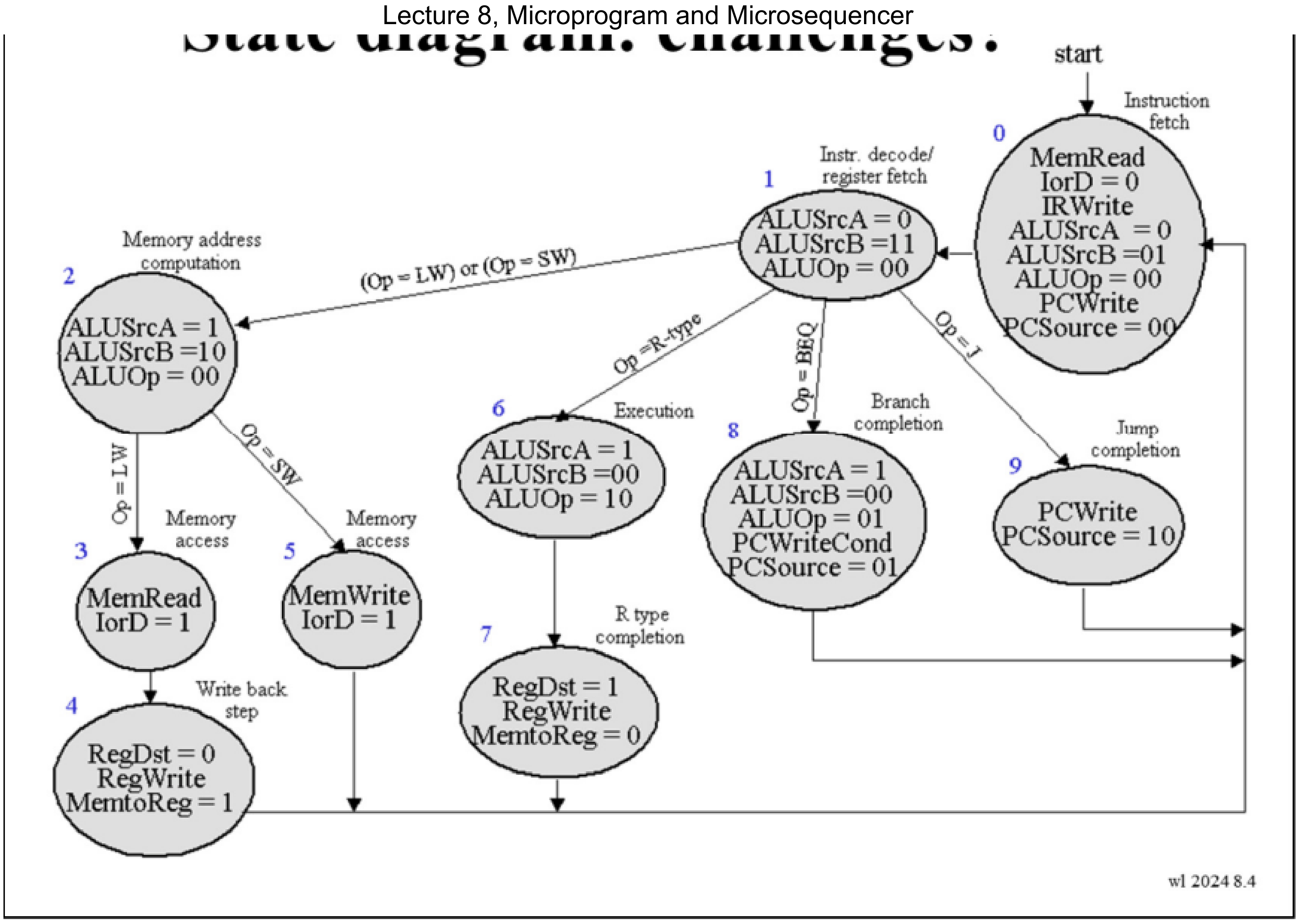
Lecture 8, Microprogram and Microsequencer



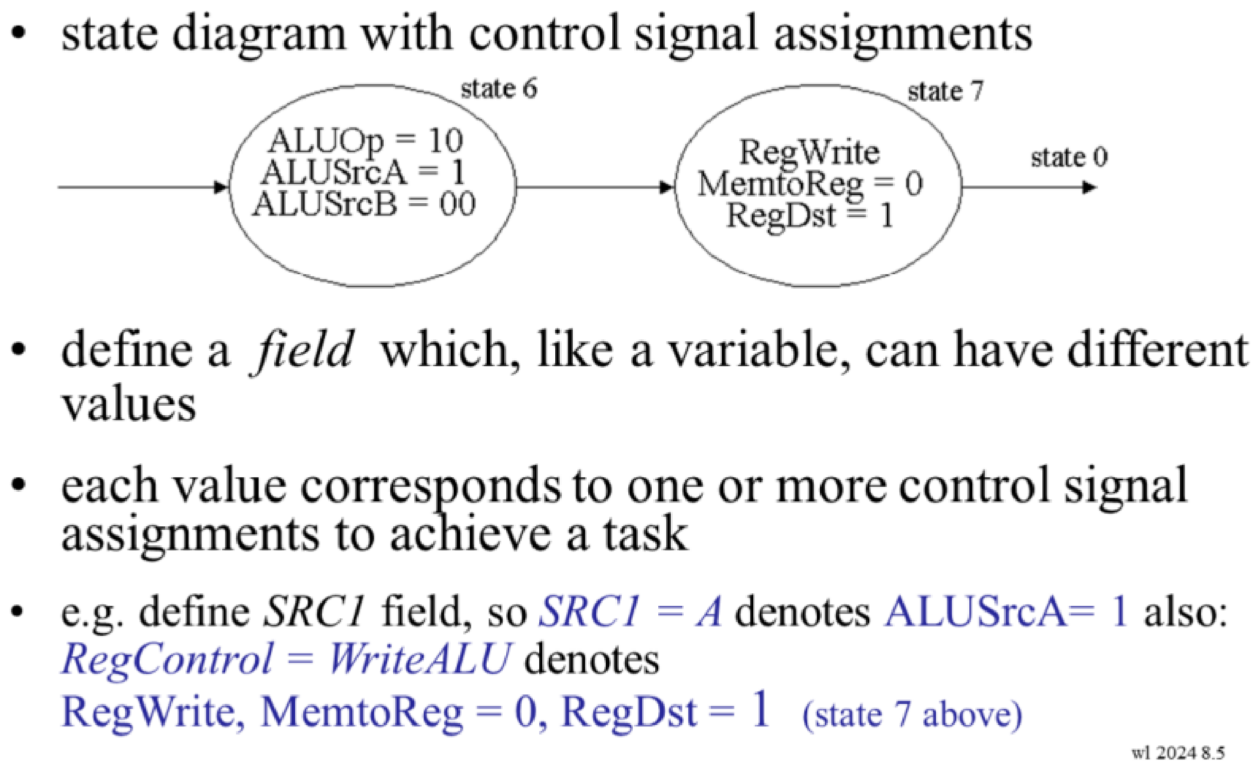
Just in case someone missed the last lecture, we shall be covering how the control unit can be implemented. The input to the control unit is the 6-bit opcode from bit 26-31 of the instruction register. The output of the control unit contains 13 control signals. Three of them, `PCSource`, `ALUOp` and `ALUSrcB`, have two bits, so the output of the control unit is 16-bit wide.

State Diagram: Challenges?



Compared to the state diagram containing RTL assignments, the state diagram containing the control signal assignments is difficult to understand. Also its structure is not explicit, and it can become a mess for realistic instructions. So can we adopt a program-like description for specifying the control signal assignments?

Grouping and naming control signals


state 6
ALUOp = 10
ALUSrcA = 1
ALUSrcB = 00
state 7
RegWrite
MemtoReg = 0
RegDst = 1
state 0

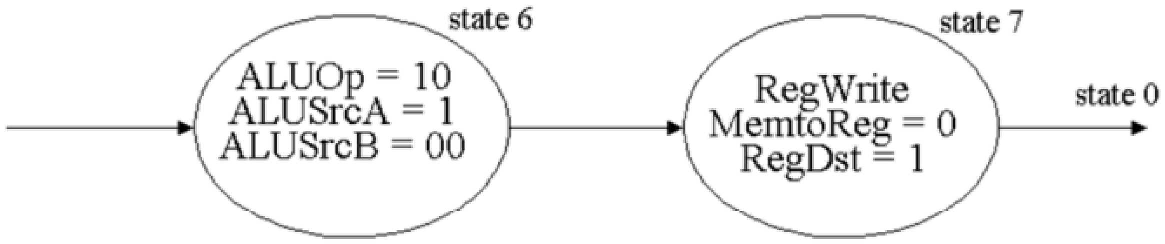
- state diagram with control signal assignments
- define a *field* which, like a variable, can have different values
- each value corresponds to one or more control signal assignments to achieve a task
- e.g. define *SRC1* field, so *SRC1 = A* denotes ALUSrcA= 1 also: *RegControl = WriteALU* denotes RegWrite, MemtoReg = 0, RegDst = 1 (state 7 above)

wl 2024 8.5

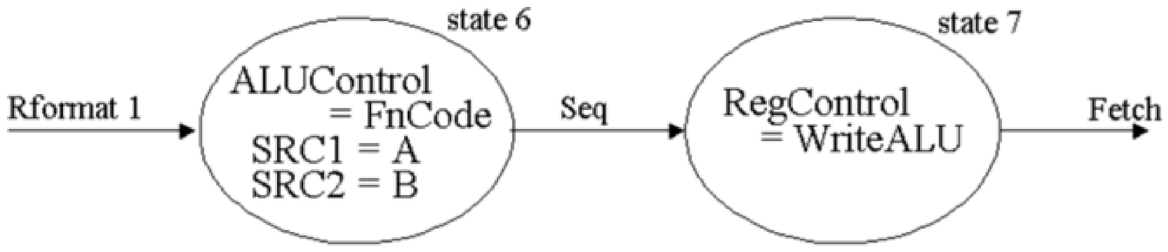
The insight is to use assignment statements to describe how groups of control signals are given their values in a specific clock cycle. For example, in state 7, instead of specifying the three control signal assignments $RegWrite$, $MemtoReg = 0$, $RegDst = 1$, we specify them using a single assignment $RegControl = WriteALU$. $RegControl$ is called a field, and when it is assigned the value $WriteALU$, it corresponds to the three control signal assignments shown earlier.

Relating State Diagram and field assignment diagram

- state diagram with control signal assignments

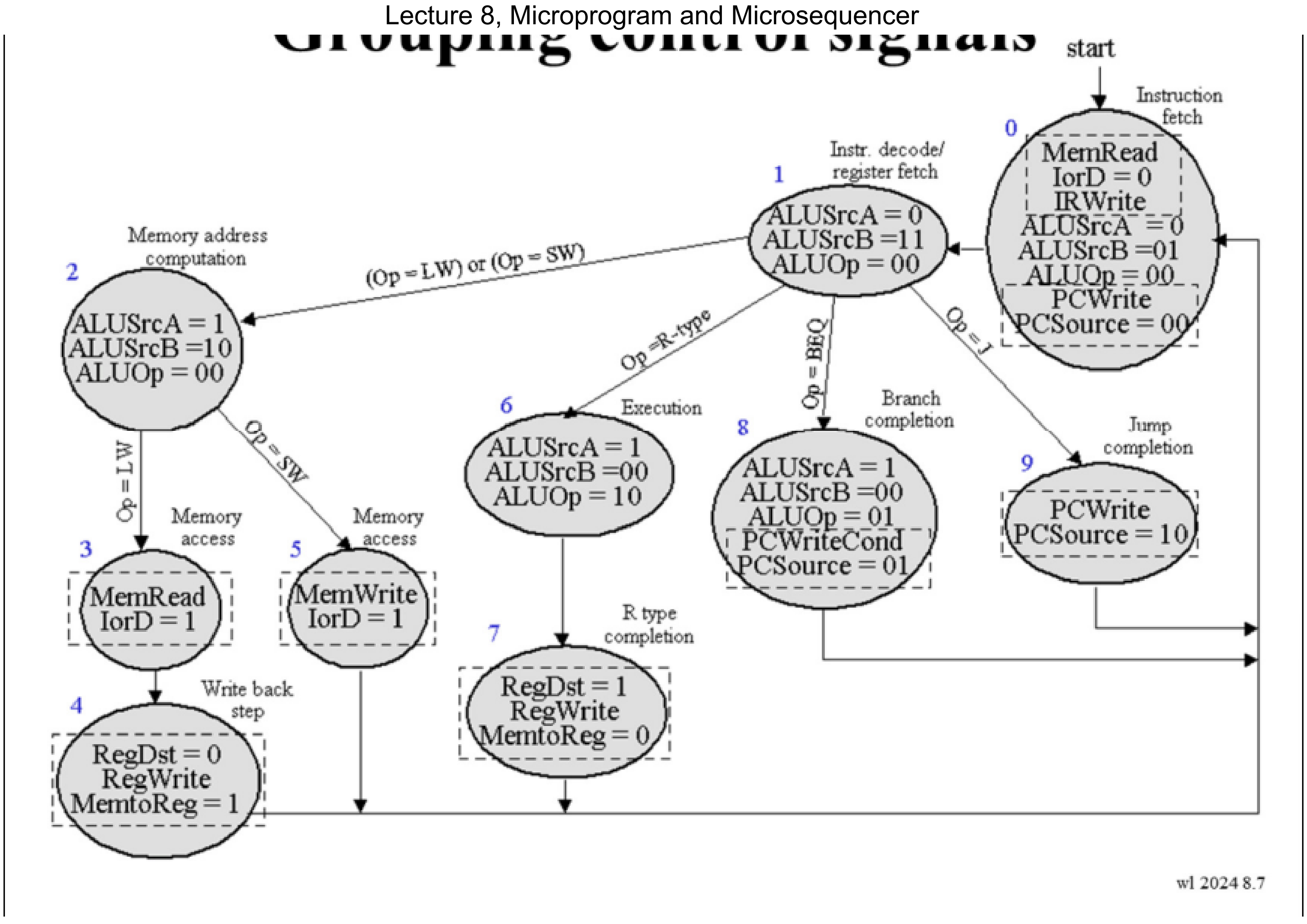


- field assignment diagram



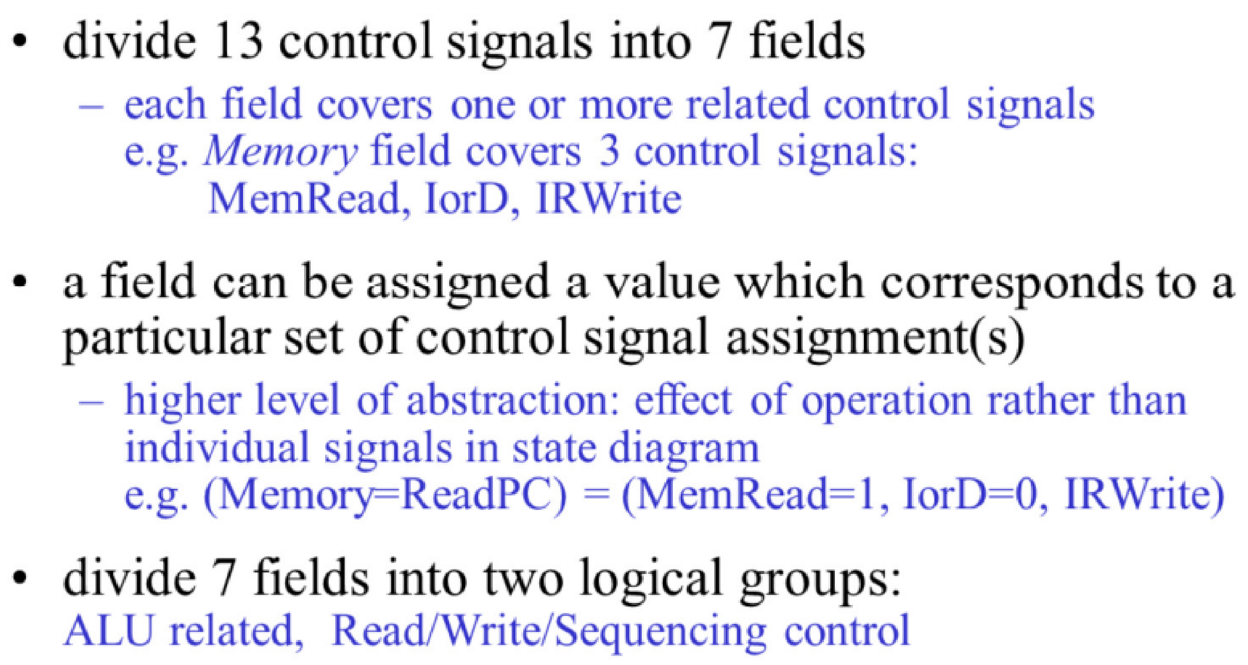
So by using meaningful names for fields and their values, the control signal assignments can be much clearer. So the state diagram with control signal assignments can be replaced by a field assignment diagram, which is often more concise and understandable.

Grouping Control Signals



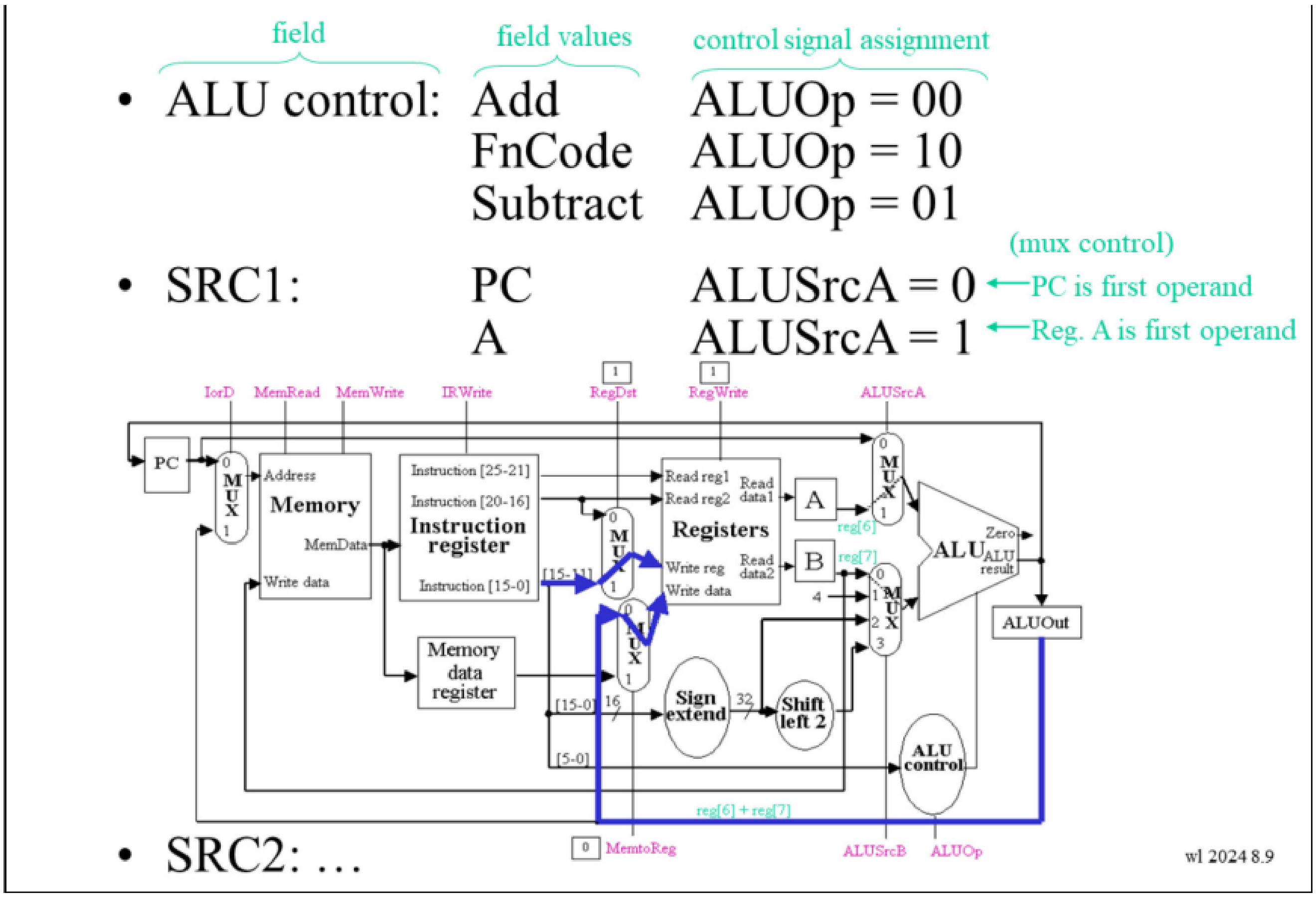
This diagram shows how the related control signal assignments in each state are grouped together to become one or more field assignments.

Field Assignments: Details


- divide 13 control signals into 7 fields
  - each field covers one or more related control signals
    e.g. *Memory* field covers 3 control signals:
    MemRead, IorD, IRWrite
- a field can be assigned a value which corresponds to a particular set of control signal assignment(s)
  - higher level of abstraction: effect of operation rather than individual signals in state diagram
    e.g. (Memory=ReadPC) = (MemRead=1, IorD=0, IRWrite)
- divide 7 fields into two logical groups:
  ALU related, Read/Write/Sequencing control

The 13 control signals are arranged into 7 fields, so that each field covers related control signals. Assigning a value to a field denotes one or more assignment of values to related control signals.

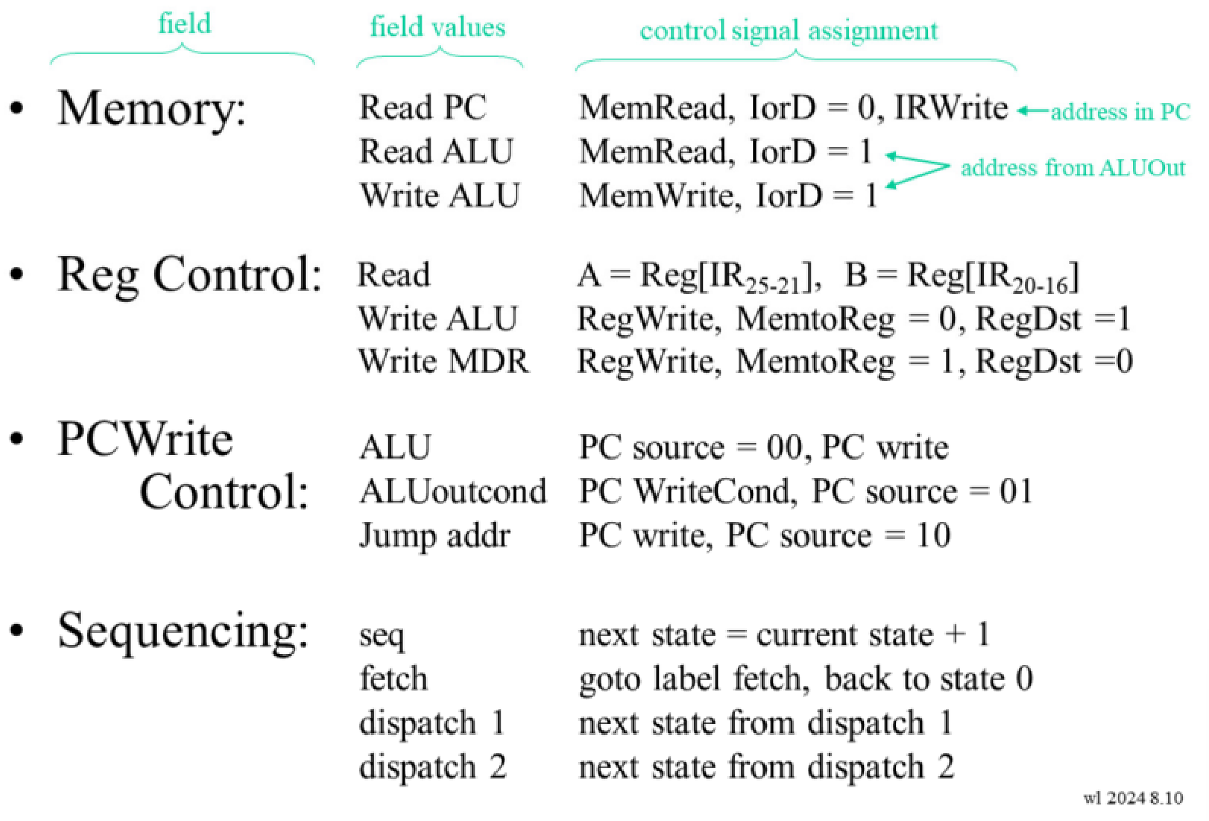
3 ALU related fields



field	field values	control signal assignment	
• ALU control:	Add	ALUOp = 00	
	FnCode	ALUOp = 10	
	Subtract	ALUOp = 01	
			(mux control)
• SRC1:	PC	ALUSrcA = 0	← PC is first operand
	A	ALUSrcA = 1	← Reg. A is first operand
• SRC2: …			

So the field assignment $ALUcontrol = Add$ means the control signal assignment $ALUOp = 00$, and the field assignment $SRC1 = PC$ means the control signal assignment $ALUSrcA = 0$.

4 Read/Write/Sequencing Fields

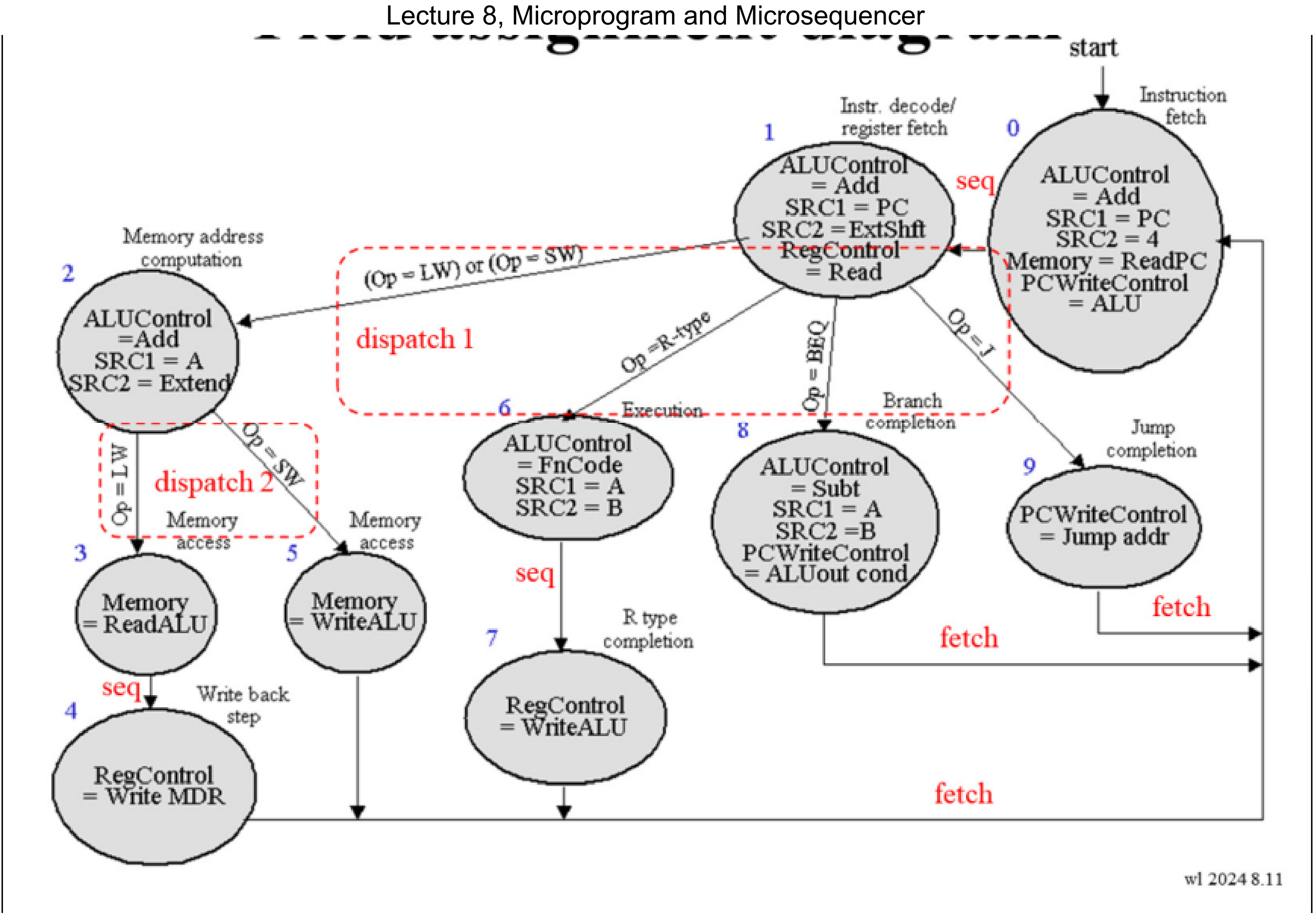


field	field values	control signal assignment
• Memory:	Read PC	MemRead, IorD = 0, IRWrite ← address in PC
	Read ALU	MemRead, IorD = 1 ← address from ALUOut
	Write ALU	MemWrite, IorD = 1 ← address from ALUOut
• Reg Control:	Read	A = Reg[IR_{25-21}], B = Reg[IR_{20-16}]
	Write ALU	RegWrite, MemtoReg = 0, RegDst =1
	Write MDR	RegWrite, MemtoReg = 1, RegDst =0
• PCWrite Control:	ALU	PC source = 00, PC write
	ALUoutcond	PC WriteCond, PC source = 01
	Jump addr	PC write, PC source = 10
• Sequencing:	seq	next state = current state + 1
	fetch	goto label fetch, back to state 0
	dispatch 1	next state from dispatch 1
	dispatch 2	next state from dispatch 2

wl 2024 8.10

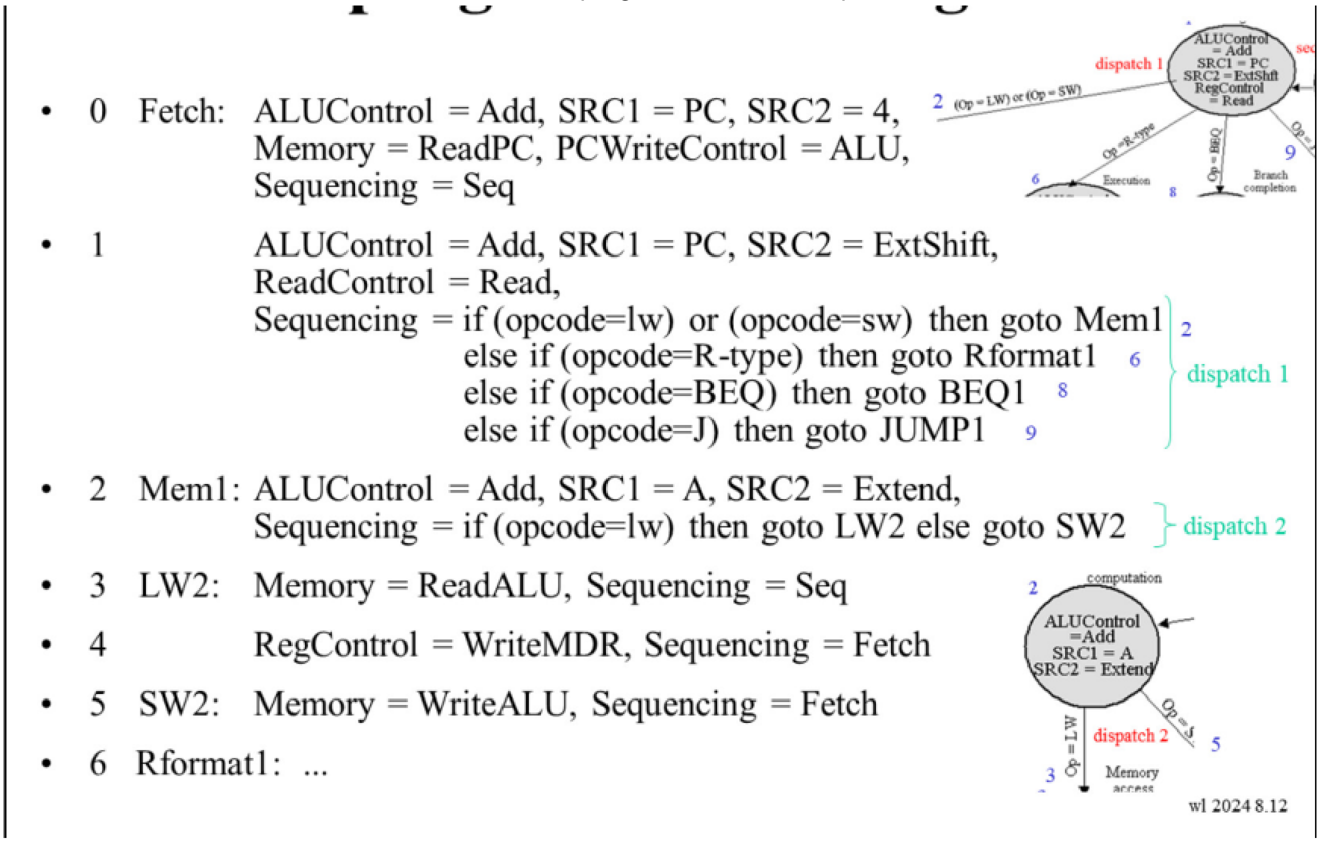
This slide summarises the field assignments for the Read/Write/Sequencing fields. So for example, the field assignment $Memory = ReadPC$ means the following three control signal assignments: $MemRead, IorD = 0, IRWrite$. The field values dispatch 1 and dispatch 2 are lookup tables to produce the next state from the opcode.

Field Assignment Diagram

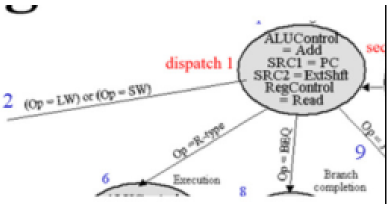


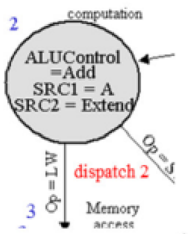
So now we can transform the state diagram with control signal assignments to the corresponding field assignment diagram. The red values are those assigned to for the Sequencing field.

Microprogram: Field Assignment



- 0 Fetch: ALUControl = Add, SRC1 = PC, SRC2 = 4, Memory = ReadPC, PCWriteControl = ALU, Sequencing = Seq
- 1 ALUControl = Add, SRC1 = PC, SRC2 = ExtShift, ReadControl = Read,
  Sequencing = if (opcode=lw) or (opcode=sw) then goto Mem1 2
  else if (opcode=R-type) then goto Rformat1 6
  else if (opcode=BEQ) then goto BEQ1 8
  else if (opcode=J) then goto JUMP1 9
  } dispatch 1
- 2 Mem1: ALUControl = Add, SRC1 = A, SRC2 = Extend,
  Sequencing = if (opcode=lw) then goto LW2 else goto SW2 } dispatch 2
- 3 LW2: Memory = ReadALU, Sequencing = Seq
- 4 RegControl = WriteMDR, Sequencing = Fetch
- 5 SW2: Memory = WriteALU, Sequencing = Fetch
- 6 Rformat1: ...





wl 2024 8.12

Instead of a diagram, we can describe the field assignment as a microprogram or microcode – note that $Sequencing = Seq$ means go to the next state (current state plus one), while $sequencing = if\ldots$ means the next state depends on the opcode, which specifies which instruction is being executed. The next state will depend on a lookup table: either dispatch 1 in state 1 or dispatch 2 in state 2.

Microprogram: From assignment to table

- assignment (field as LHS, value as RHS):

  6 Rformat1 ALUControl = FnCode, SRC1 =A, SRC2 = B, Sequencing=Seq

  7 RegControl = WriteALU, Sequencing=Fetch

- table

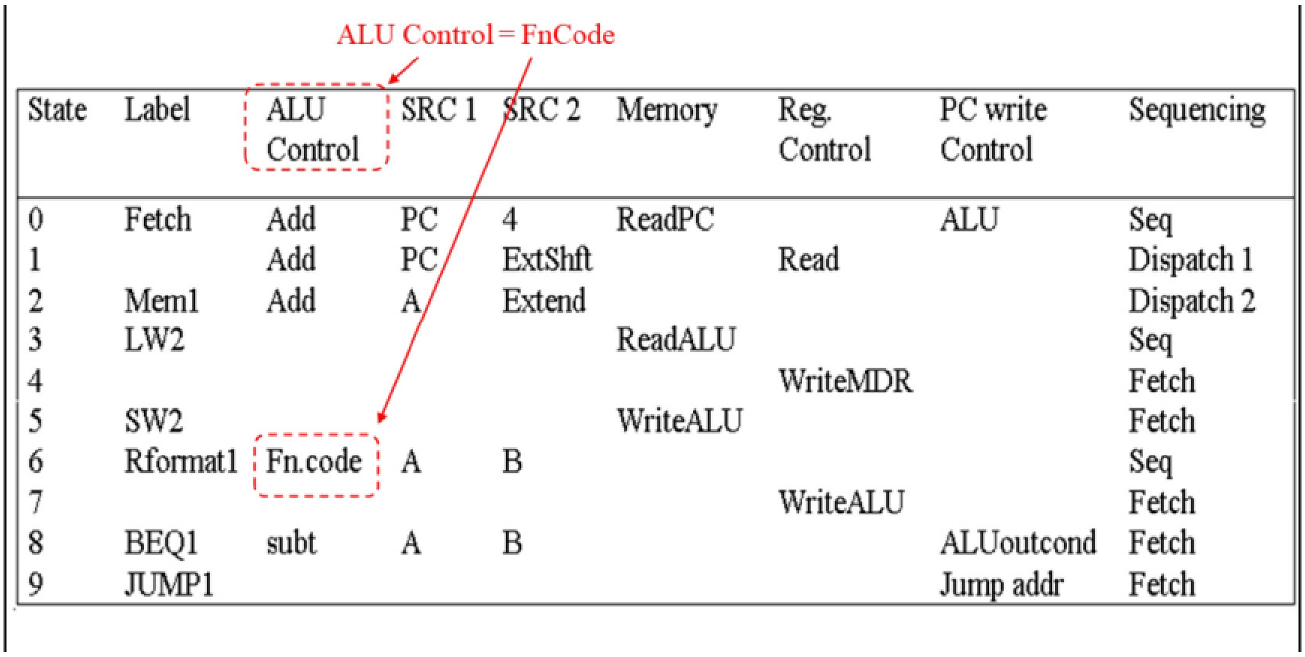
LHS of the above field assignment

RHS of the above field assignment

Label	ALU Control	SRC1	SRC2	Memory	Reg Control	PCWrite Control	Sequencing
6 Rformat1	FnCode	A	B				Seq
7					WriteALU		Fetch

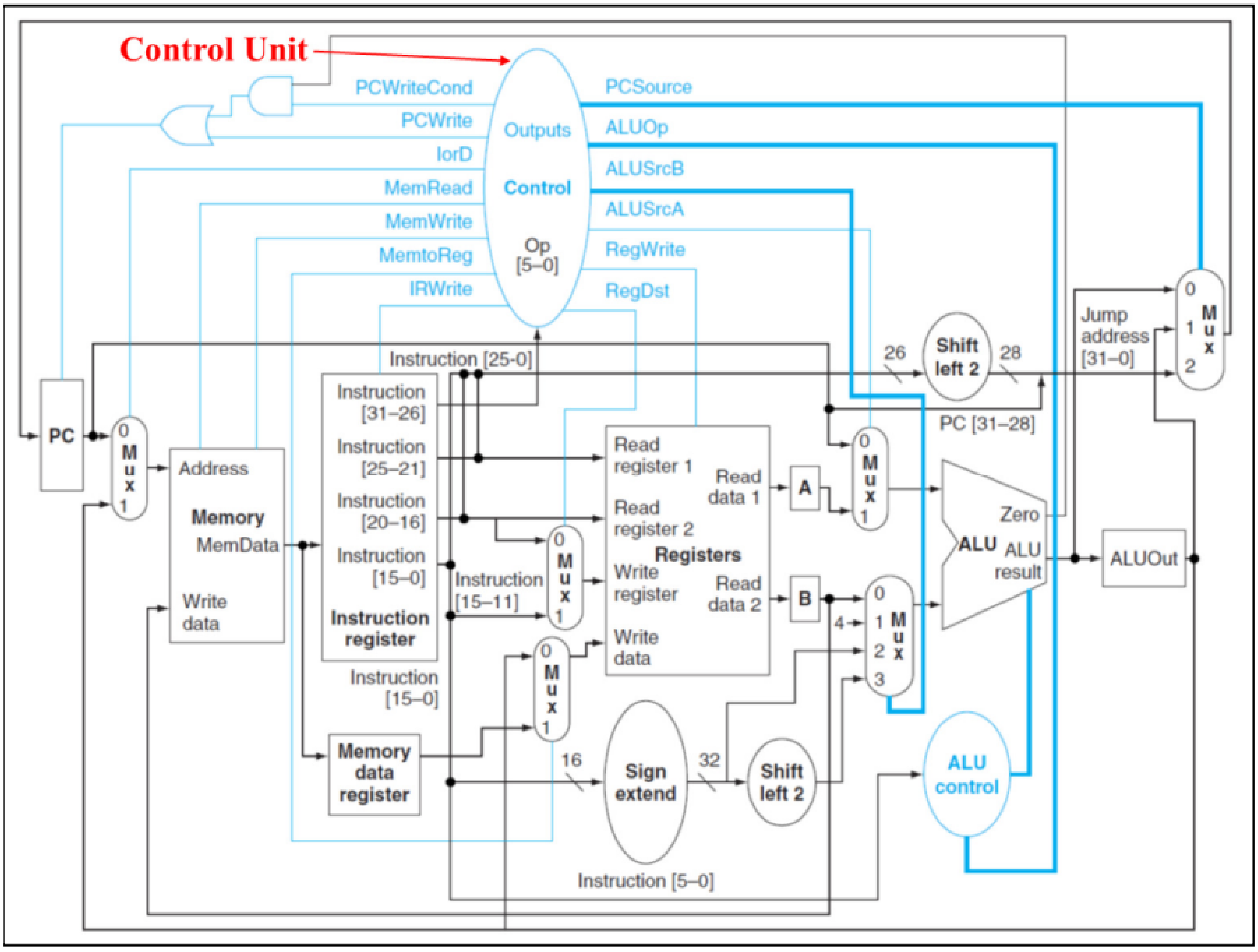
Instead of the program format, we can adopt a table format for the microprogram. The fields are placed at the first row; each of the other rows corresponds to a state showing the values to which specific fields (on the top) are assigned.

Microprogram: table form


ALU Control = FnCode

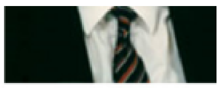
State	Label	ALU Control	SRC 1	SRC 2	Memory	Reg. Control	PC write Control	Sequencing
0	Fetch	Add	PC	4	ReadPC		ALU	Seq
1		Add	PC	ExtShft		Read		Dispatch 1
2	Mem1	Add	A	Extend				Dispatch 2
3	LW2				ReadALU			Seq
4						WriteMDR		Fetch
5	SW2				WriteALU			Fetch
6	Rformat1	Fn.code	A	B				Seq
7						WriteALU		Fetch
8	BEQ1	subt	A	B			ALUoutcond	Fetch
9	JUMP1						Jump addr	Fetch

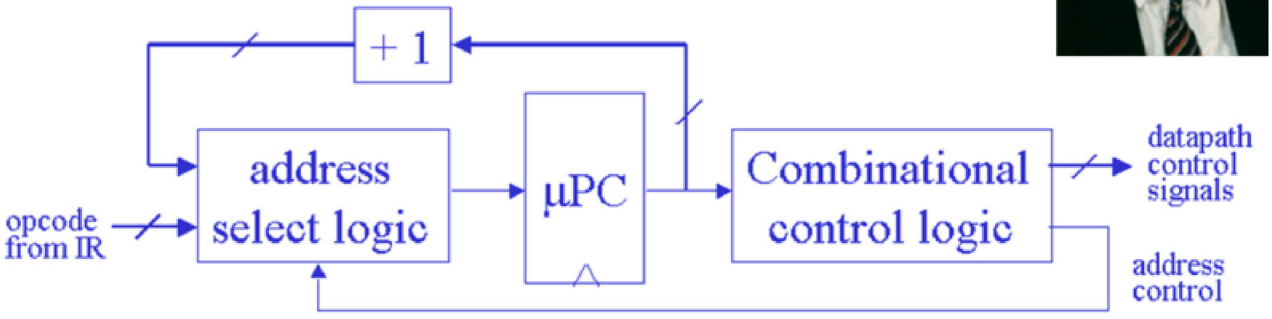
So the microprogram can be described by this table, illustrating a specific field assignment.



Lets go back to the control unit – are there better ways of implementing it?

Microsequencer

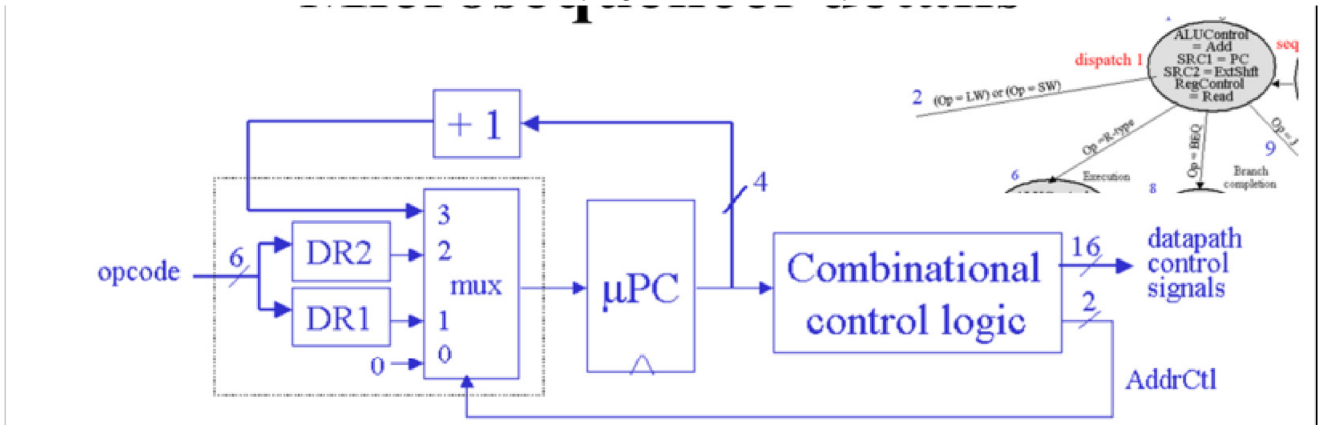




- microprogram counter μPC: 4 bits (state 0-9)
- address select: +1 or state 1, state 2 branching
- combin. logic: μPC output → datapath control and address control

Microprogramming and microsequencer were invented by Maurice Wilkes, who also developed EDSAC, the first full-size stored-program computer. The finite state machine is useful for implementing irregular state diagrams, where any state can go to any other states. However, the state diagram for the control unit is more structured, where several next states, for example, are just increments of the current state. So we use a microprogram counter μPC which can be incremented to cover the case when the next state is just the increment of the current state. The address select logic decides whether the next state should be the increment of the current one, or state 0 (fetch), or from one of the two dispatch tables shown earlier.

Microsequencer Details



- Dispatch ROM 1 (DR1):

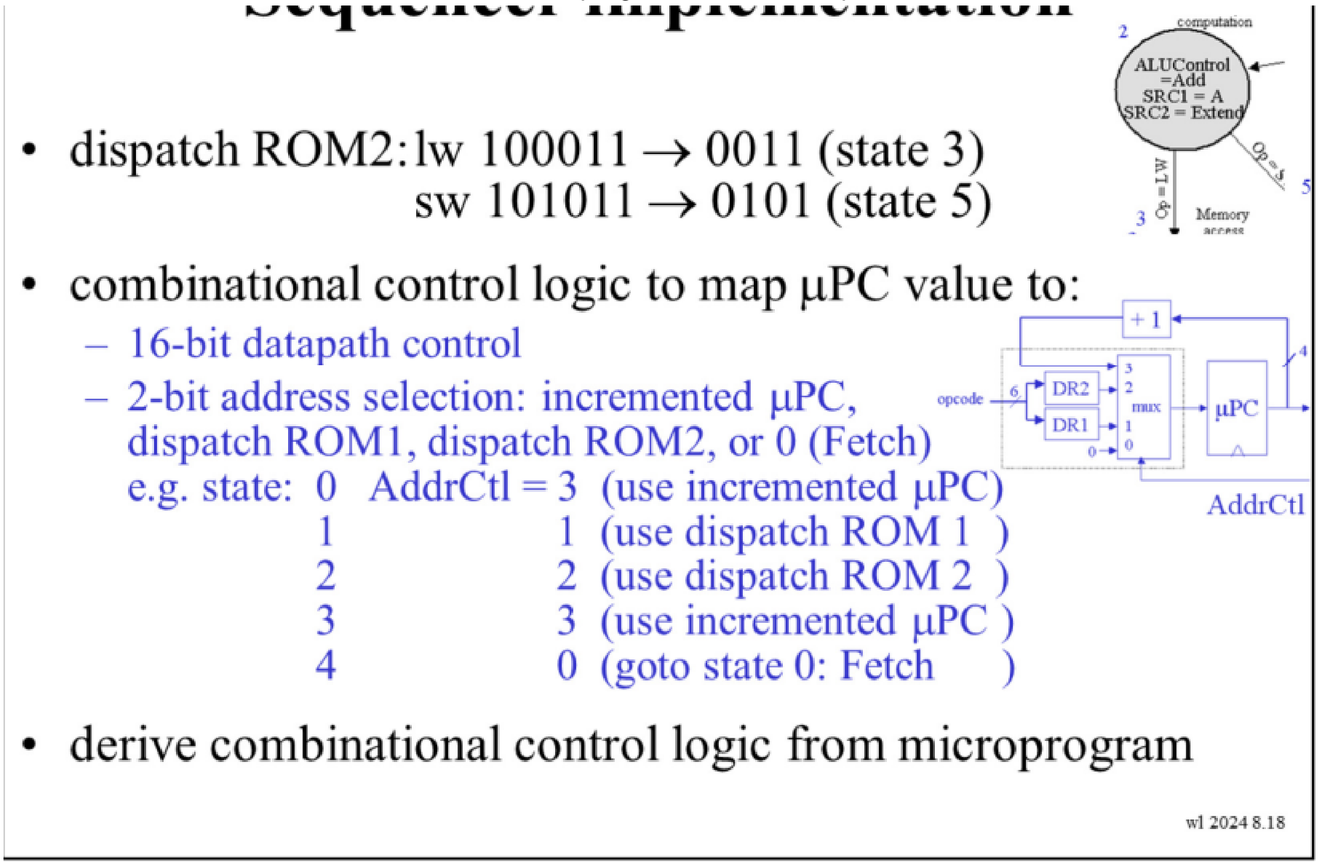
	opcode in	state out	
R-type	000000	0110	state 6
jump	000010	1001	state 9
beq	000100	1000	state 8
lw	100011	0010	state 2
sw	101011	0010	state 2

if R-type then state 6
elseif jump then state 9

wl 2024 8.17

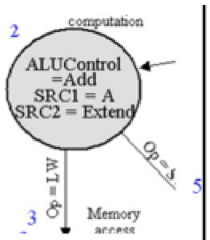
The two dispatch tables are implemented by ROMs, DR1 and DR2. The content of DR1 just maps an opcode into the appropriate state for processing that instruction. So If the opcode is R-type, then the output of DR1 is 6, denoting state 6, which is specific to processing R-type instructions.

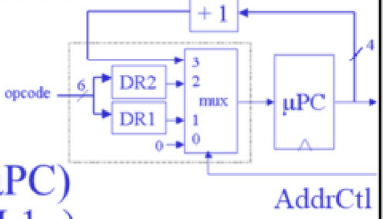
Sequencer Implementation



- dispatch ROM2: lw 100011 → 0011 (state 3)
  sw 101011 → 0101 (state 5)
- combinational control logic to map μPC value to:
  - 16-bit datapath control
  - 2-bit address selection: incremented μPC, dispatch ROM1, dispatch ROM2, or 0 (Fetch)
    e.g. state: 0 AddrCtl = 3 (use incremented μPC)
    1 1 (use dispatch ROM 1)
    2 2 (use dispatch ROM 2)
    3 3 (use incremented μPC)
    4 0 (goto state 0: Fetch)
- derive combinational control logic from microprogram

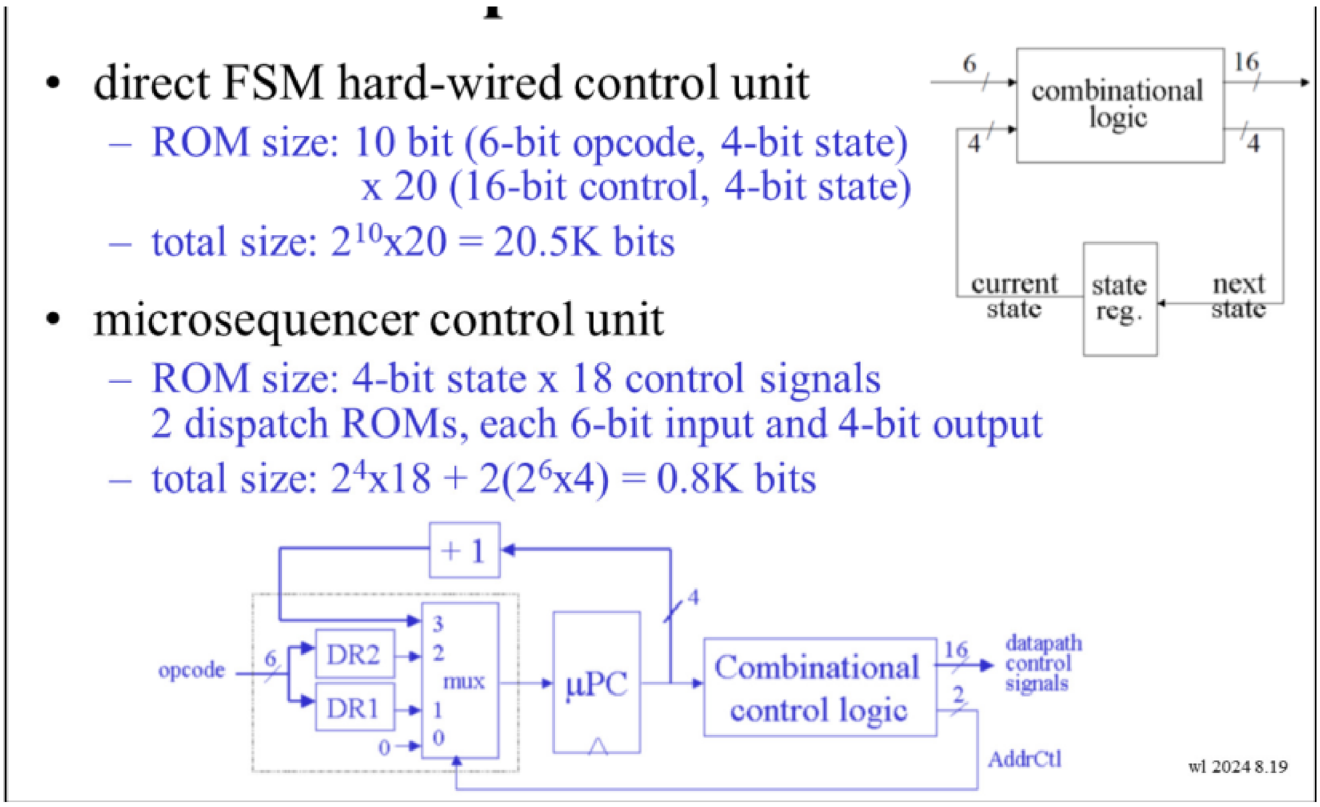
wl 2024 8.18





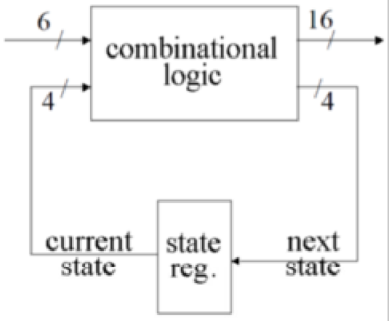
The address control `AddrCtl` is the control input to a multiplexor selecting one of the 4 possibilities for the next state of the microprogram counter μPC: incrementing it, use dispatch ROM 1, use dispatch ROM 2, or go to state 0.

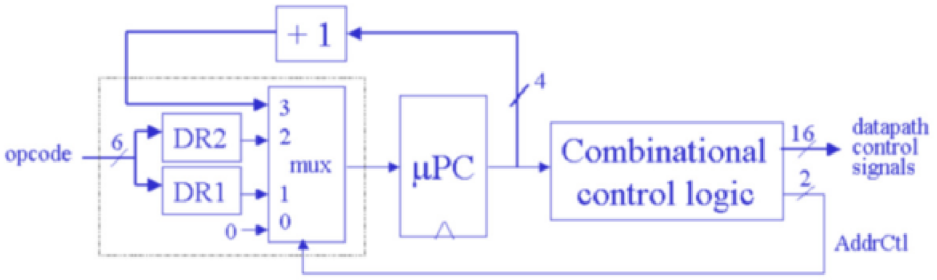
Compare control unit implementations



- direct FSM hard-wired control unit
  - ROM size: 10 bit (6-bit opcode, 4-bit state) x 20 (16-bit control, 4-bit state)
  - total size: 2^{10}x20 = 20.5K bits
- microsequencer control unit
  - ROM size: 4-bit state x 18 control signals
    2 dispatch ROMs, each 6-bit input and 4-bit output
  - total size: 2^4x18 + 2(2^6x4) = 0.8K bits

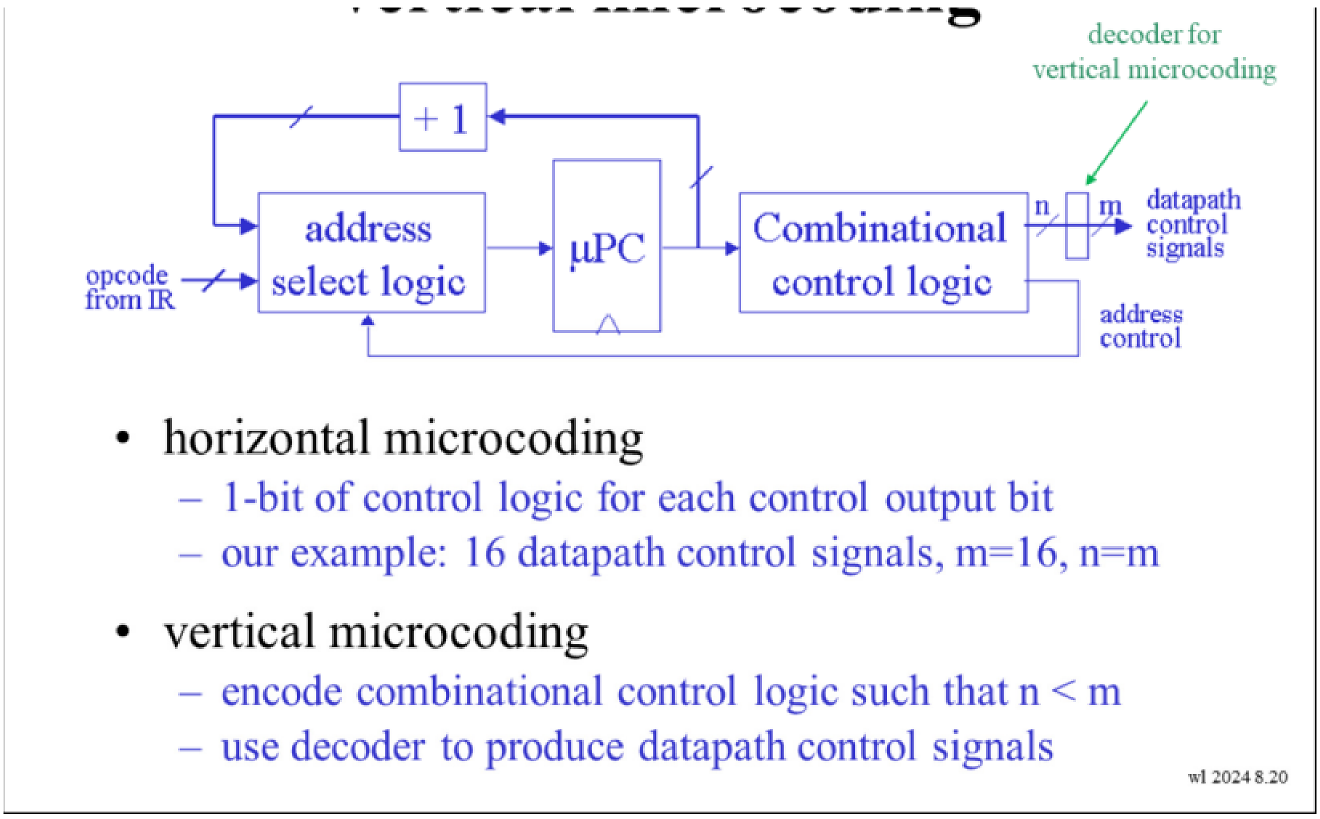
wl 2024 8.19





Since the size of a ROM is $2^m * n$ where m and n are the number of input and output bits, a smaller value for m will particularly help to reduce the ROM size. The ROMs in the microsequencer have fewer inputs than the ROM in the FSM, hence its size is over 25 times smaller. However, it needs additional blocks such as an incrementor and a multiplexor.

Microsequencer implementation: Vertical microcoding

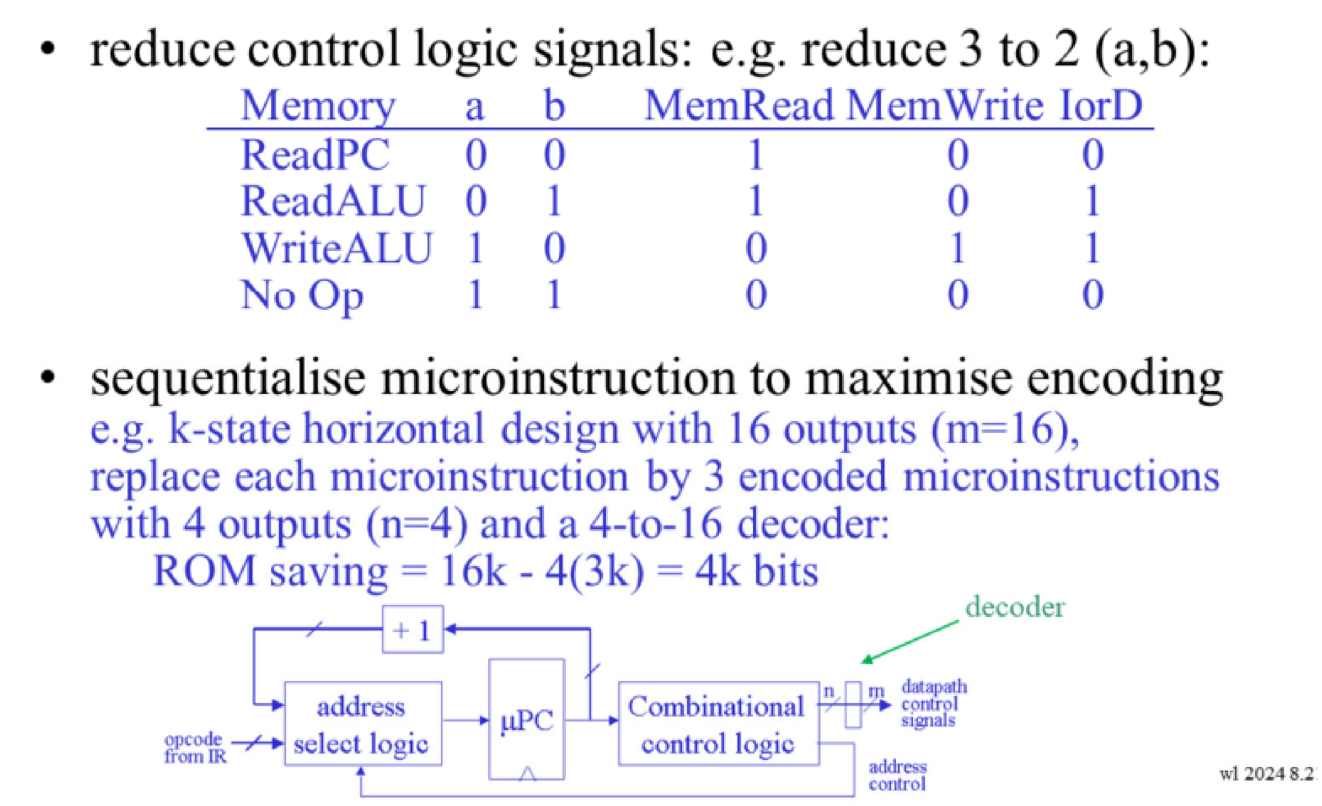


- horizontal microcoding
  - 1-bit of control logic for each control output bit
  - our example: 16 datapath control signals, m=16, n=m
- vertical microcoding
  - encode combinational control logic such that n < m
  - use decoder to produce datapath control signals

wl 2024 8.20

What has been described is horizontal microcoding. In vertical microcoding, the combinational control logic is encoded to use fewer bits (n), and an additional decoder is used to translate the encoded n signals to become the m datapath control signals.

Encoding Datapath Control Signals



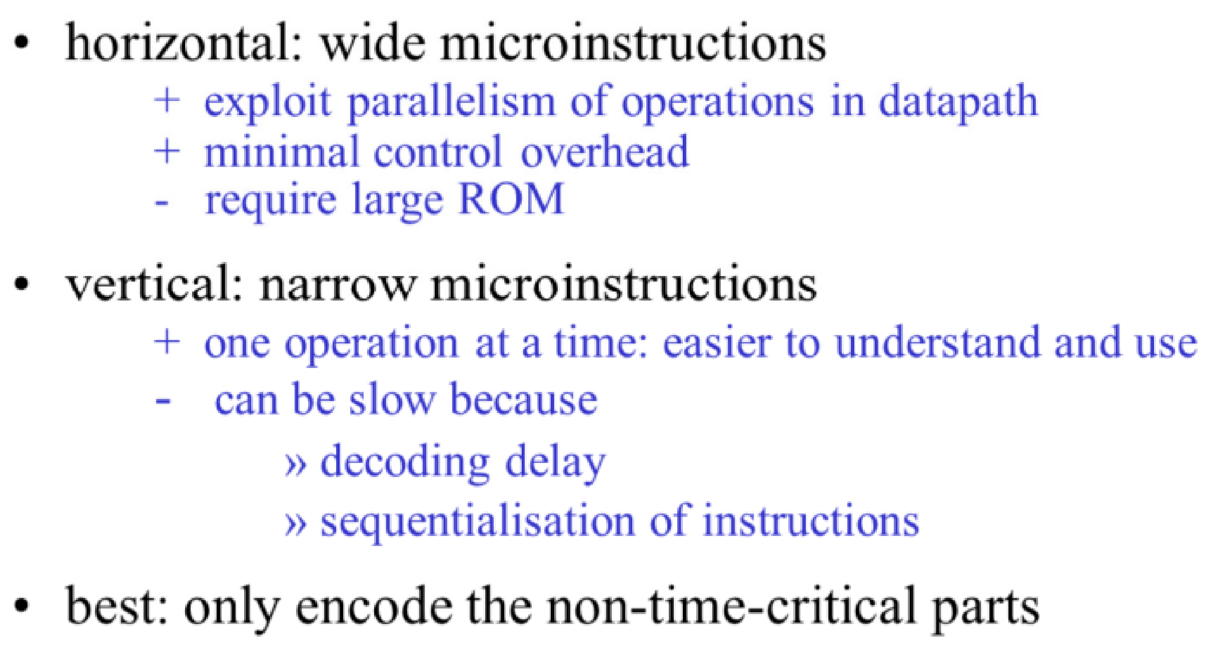
- reduce control logic signals: e.g. reduce 3 to 2 (a,b):

Memory	a	b	MemRead	MemWrite	IorD
ReadPC	0	0	1	0	0
ReadALU	0	1	1	0	1
WriteALU	1	0	0	1	1
No Op	1	1	0	0	0

- sequentialise microinstruction to maximise encoding
  e.g. k-state horizontal design with 16 outputs (m=16), replace each microinstruction by 3 encoded microinstructions with 4 outputs (n=4) and a 4-to-16 decoder:
  ROM saving = 16k - 4(3k) = 4k bits

Instead of mapping the state of the microprogram counter directly into control signals, vertical microcoding maps the state of the microprogram counter into an encoded version of the control signals. The encoded version will then be decoded to produce the control signals. For example, the 3 control signals `MemRead`, `MemWrite` and `IorD` can be driven by 2 bits a and b, as shown in the truth table specifying the decoder logic. So the two bits a and b can be the output of the combinational control logic, rather than the three control signals $MemRead$, $MemWrite$ and $IorD$.

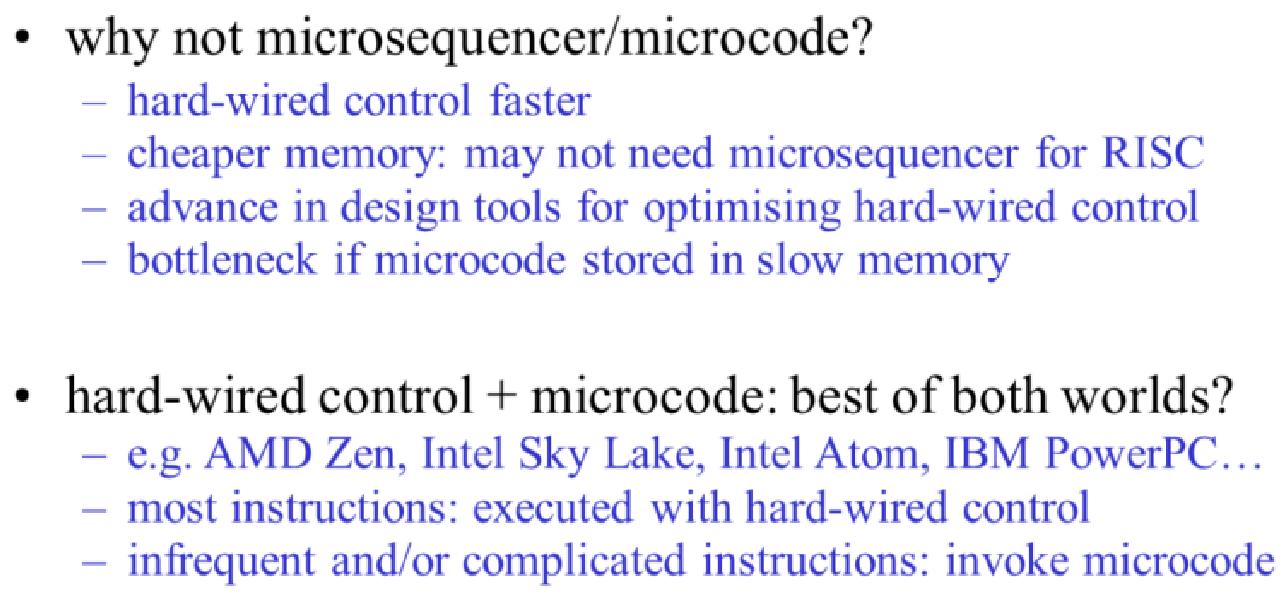
Horizontal vs Vertical Microcoding



- horizontal: wide microinstructions
  - + exploit parallelism of operations in datapath
  - + minimal control overhead
  - - require large ROM
- vertical: narrow microinstructions
  - + one operation at a time: easier to understand and use
  - - can be slow because
    - » decoding delay
    - » sequentialisation of instructions
- best: only encode the non-time-critical parts

This slide compares horizontal and vertical microcoding. Vertical microcode requires smaller instruction lengths and less storage, but requires more time to decode.

Microsequencer and RISC

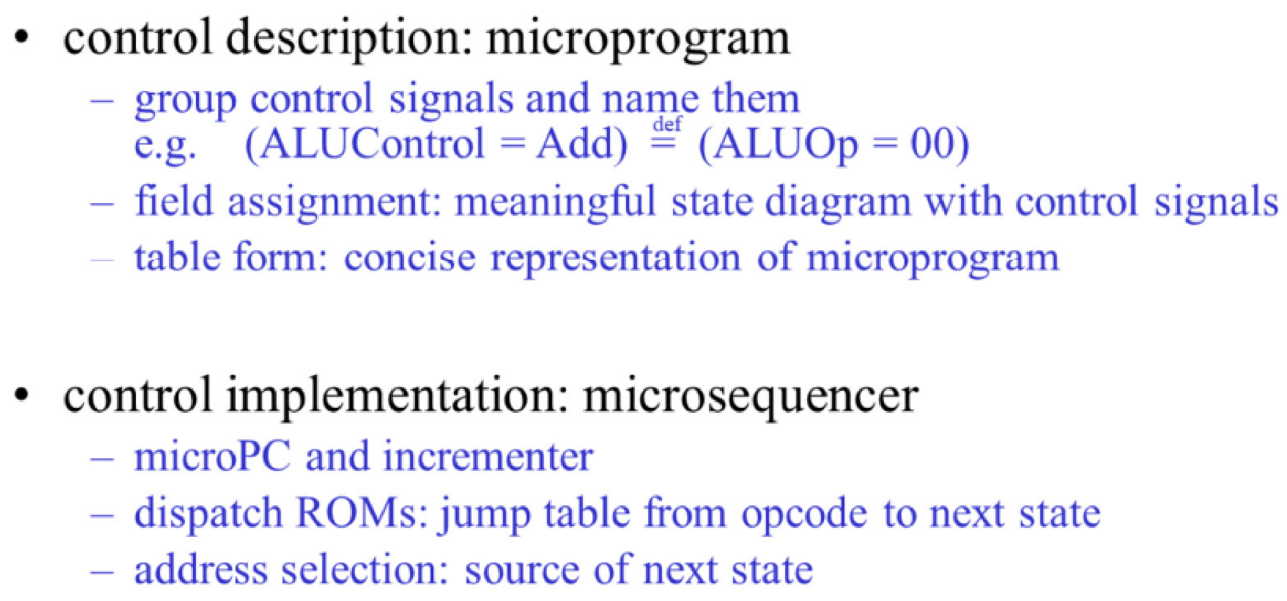


- why not microsequencer/microcode?
  - – hard-wired control faster
  - – cheaper memory: may not need microsequencer for RISC
  - – advance in design tools for optimising hard-wired control
  - – bottleneck if microcode stored in slow memory
- hard-wired control + microcode: best of both worlds?
  - – e.g. AMD Zen, Intel Sky Lake, Intel Atom, IBM PowerPC…
  - – most instructions: executed with hard-wired control
  - – infrequent and/or complicated instructions: invoke microcode

While microcoding is often adopted by CISC processors, some RISC processors, such as the DEC Alpha, support some features which can be seen to be similar to microcode (e.g. access to internal registers). Moreover, microcode still plays a useful role in many modern-day processors. Glacial, for example, is a microcoded

RISC-V core designed for low FPGA resource utilization:
https://github.com/brouhaha/glacial

Summary



- control description: microprogram
  - group control signals and name them
    e.g. (ALUControl = Add) $\overset{\text{def}}{=}$ (ALUOp = 00)
  - field assignment: meaningful state diagram with control signals
  - table form: concise representation of microprogram
- control implementation: microsequencer
  - microPC and incrementer
  - dispatch ROMs: jump table from opcode to next state
  - address selection: source of next state

Microprogram and microsequencer are more advanced ideas in computer architecture. Some said they distinguish the professionals from the nonprofessionals! If you do not feel entirely comfortable with these ideas, please ask for further explanations during the tutorial.