Digital Circuits and Systems
Lecture 11 Microcode

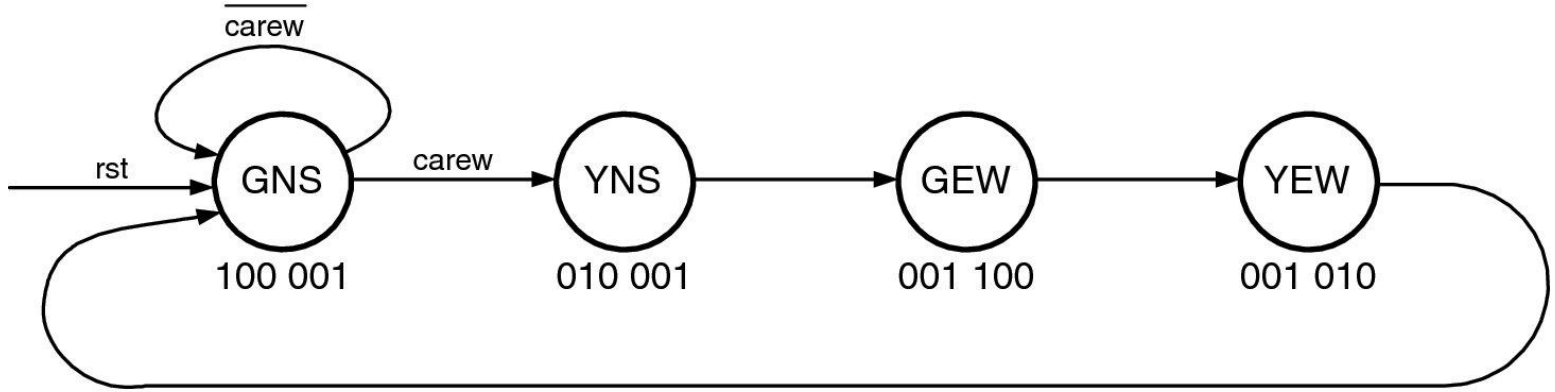
Tian Sheuan Chang

傳統FSM設計後都固定寫死了，可以不用重新設計硬體，也可以有新的FSM嗎?
用state table，而且讓table可以自由變

概念 [Dally. Ch. 18]

- 傳統FSM設計後都固定寫死了，可以不用重新設計硬體，也可以有新的FSM嗎?

State diagram



State table

讓table可以自由變

Current State	Next State		Output	
	!carew	carew		
gns	gns	yns	100001	lgns
yns	gew	gew	010001	lyns
gew	yew	yew	001100	lgew
yew	gns	gns	001010	lyew

概念 [Dally. Ch. 18]

- 傳統FSM設計後都固定寫死了，可以不用重新設計硬體，也可以有新的FSM嗎?
  - state用table，
    - 而且讓table可以自由變 => 程式
  - 那next state logic or output logic
    - 一併存成表格一部分，或
    - 不外乎加減乘除與邏輯運算 =>那就做ALU (arithmetic logic unit)
- 如何做呢
  - Table 用RAM來做
    - ROM也可以
    - PLA也是
    - 類似的memory型式: Flash, EEPROM
  - 演化到後來，就是CPU了

Sir Maurice Wilkes with a piece of EDSAC

Sir Maurice Vincent Wilkes, (born June 26, 1913, Dudley, Worcestershire, Eng.—died Nov. 29, 2010, Cambridge, Cambridgeshire), British computer science pioneer who helped build the Electronic Delay Storage Automatic Calculator (EDSAC), the first full-size stored-program computer, and invented microprogramming.

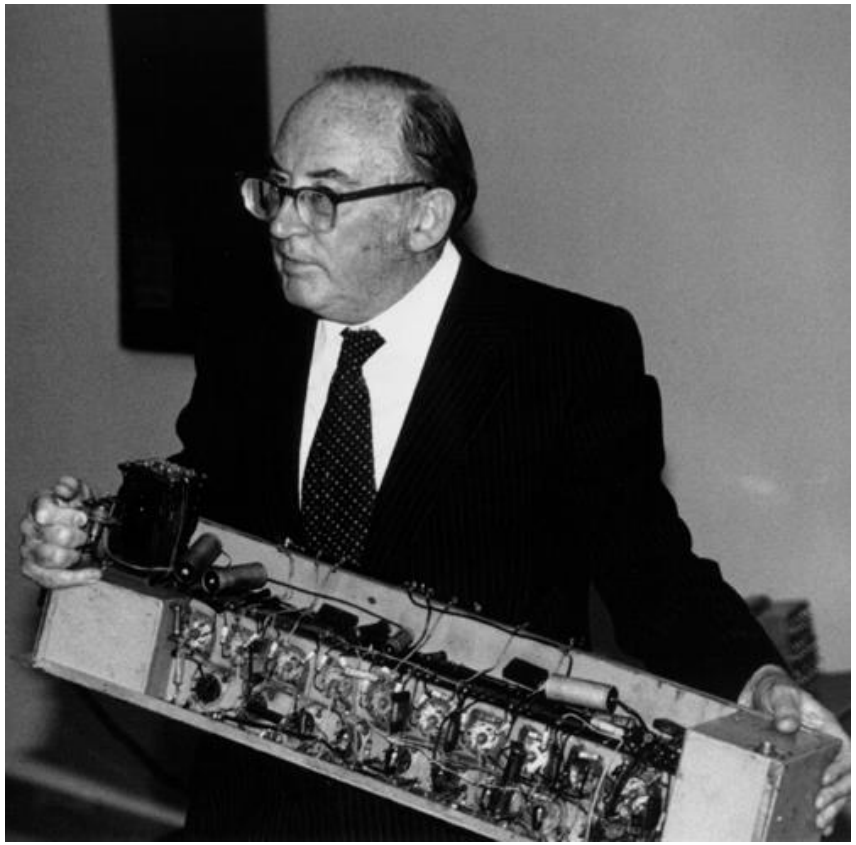
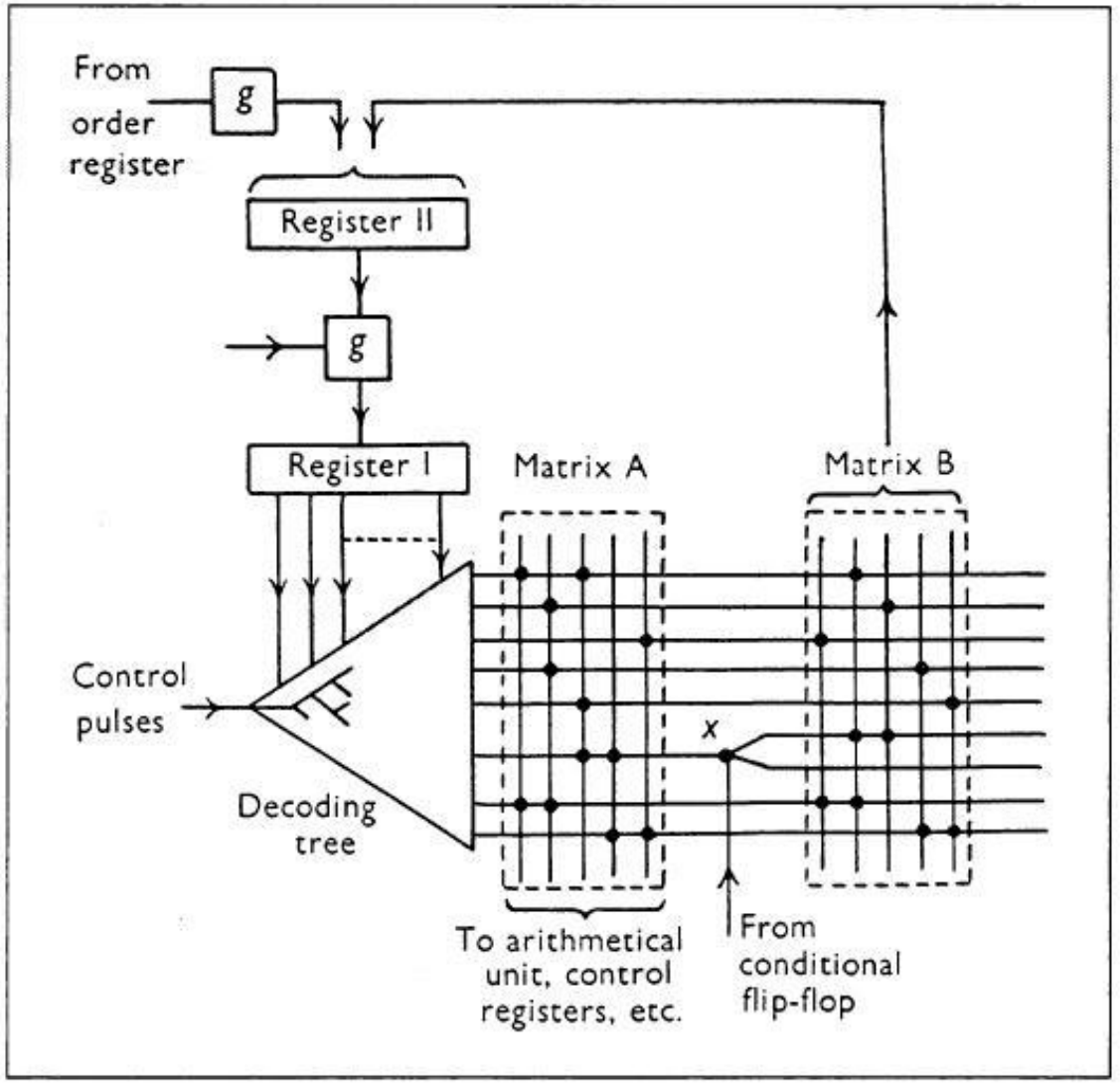


Figure from Wilkes 1953 paper on Microcode



Microcode – an FSM realized with a memory array

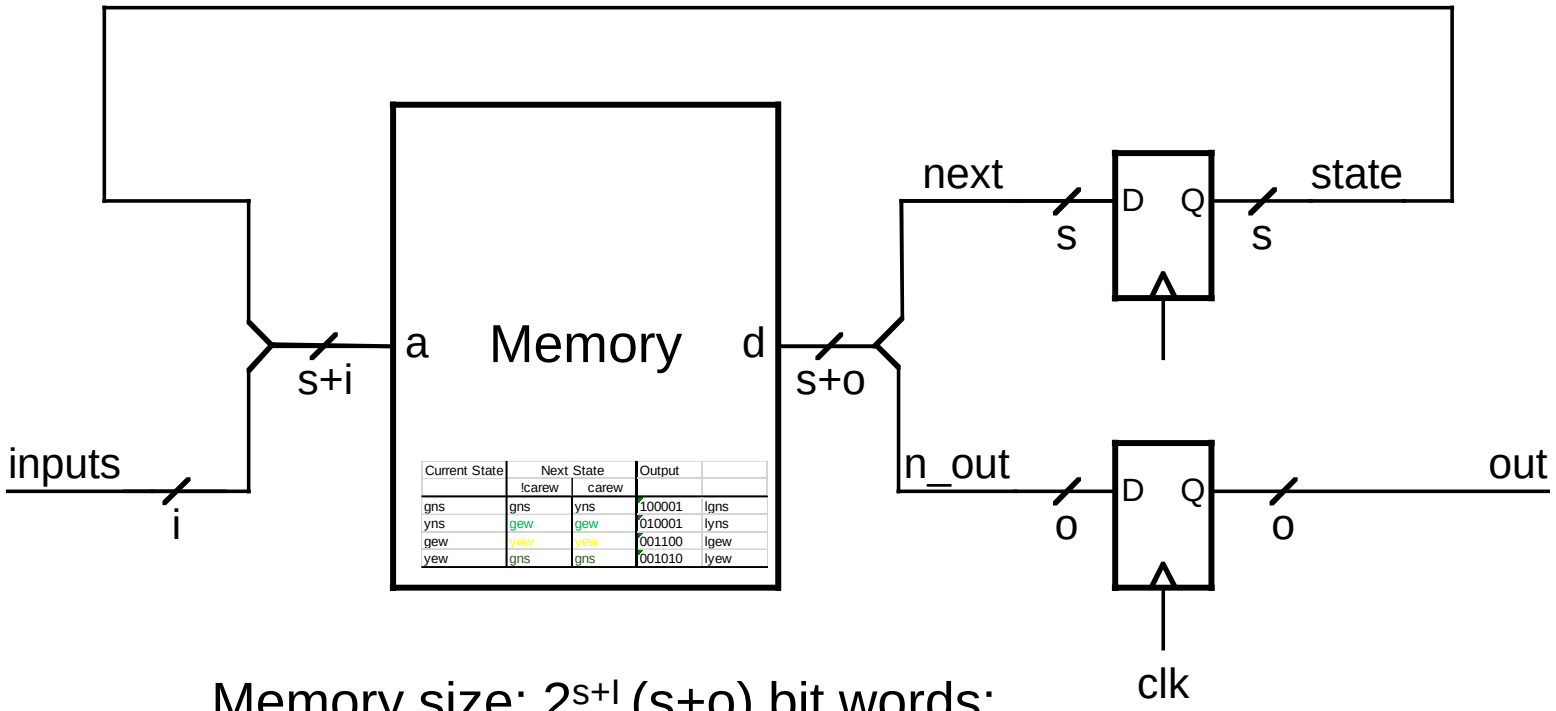
- Original concept by Wilkes (1951)
  - Put state table in a memory (ROM or RAM)
  - Address: current state and input
  - Output (Data): next state and output

Current State	Next State		Output	
	!carew	carew		
gns	gns	yns	100001	lgns
yns	gew	gew	010001	lyns
gew	yew	yew	001100	lgew
yew	gns	gns	001010	lyew

Address
Current state, input

DATA
Next state, output

Microcode – the picture

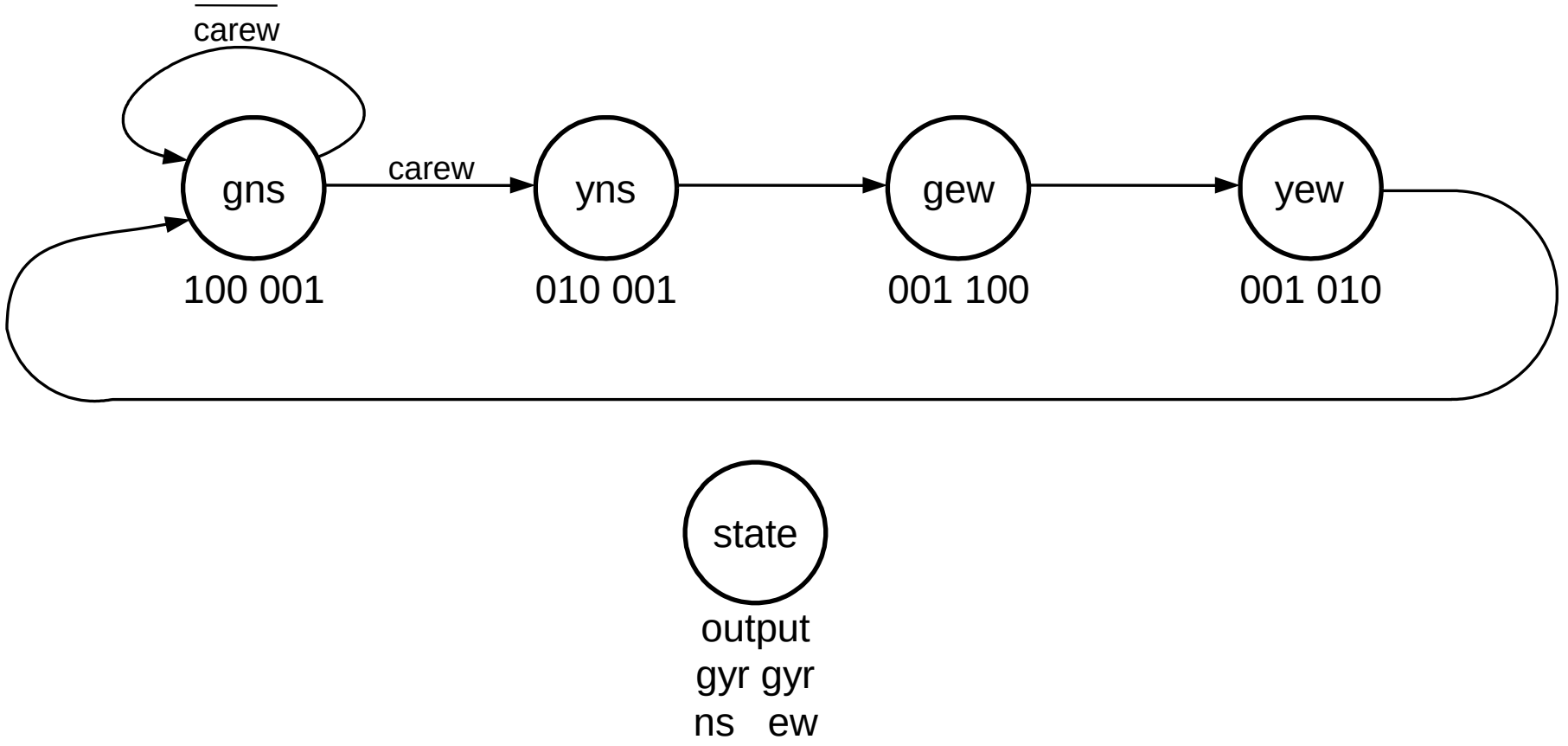


Current State	Next State		Output	
	!carew	carew		
gns	gns	yns	100001	lgns
yns	gew	gew	010001	lyns
gew	yew	yew	001100	lgew
yew	gns	gns	001010	lyew

Memory size: 2^{s+I} (s+o) bit words;
Exponentially increases, (s+i)

Similar to combinational logic with full-addressing space

Example: Simple Light-Traffic Controller



Re-writing State Table Of Example

原來的寫法太簡化，要跟據所有state和輸入變化完全展開

	State	Next State		Output
		!carew	carew	
gns	00	00	01	100001
yns	01	11	11	010001
gew	11	10	10	001100
yew	10	00	00	001010

	State	carew	address	data
gns	00	0	000	00100001
	00	1	001	01100001
yns	01	0	010	11010001
	01	1	011	11010001
gew	11	0	110	10001100
	11	1	111	10001100
yew	10	0	100	00001010
	10	1	101	00001010

Re-writing State Table Of Example

	State	Next State		Output
		!carew	carew	
gns	00	00	01	100001
yns	01	11	11	010001
gew	11	10	10	001100
yew	10	00	00	001010

ROM data = {next state, output}

ROM address = {state, input}

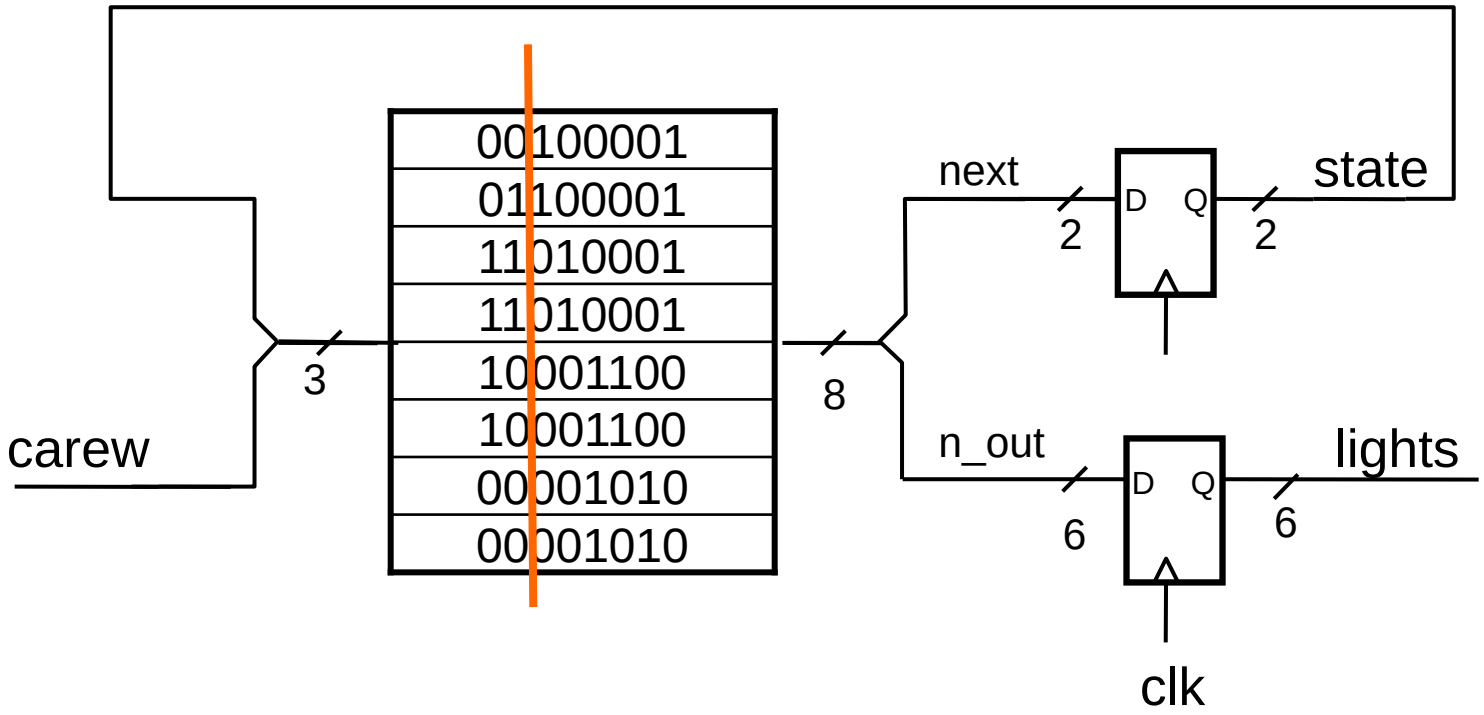
Next state

output

	State	carew	address	data
gns	00	0	000	00100001
	00	1	001	01100001
yns	01	0	010	11010001
	01	1	011	11010001
gew	11	0	110	10001100
	11	1	111	10001100
yew	10	0	100	00001010
	10	1	101	00001010

注意: 要全部展開

Microcode Of Light-Traffic Controller



當state數量和輸入變化很多造成ROM太大，有辦法縮小嗎?

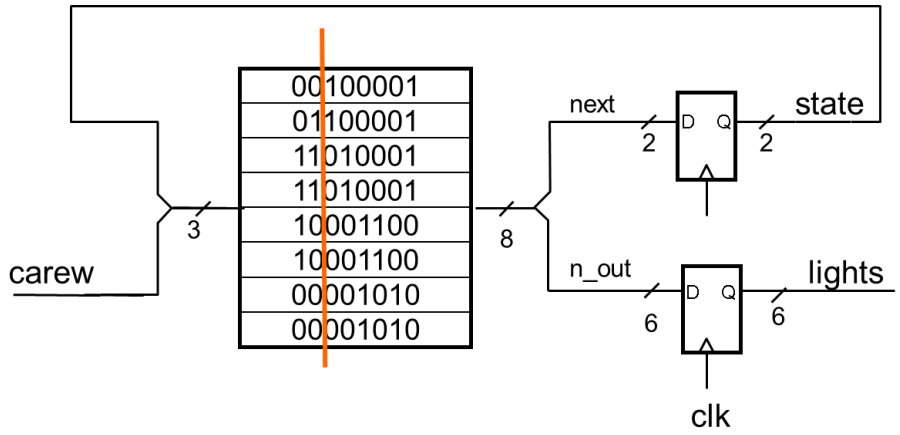
Hint: 拆成兩塊，一個存next state，
一個存output。

divide into 2 ROM's; one for state, the other for Output; so total size = 8*2+4*6=40 (vs. 8*8=64)

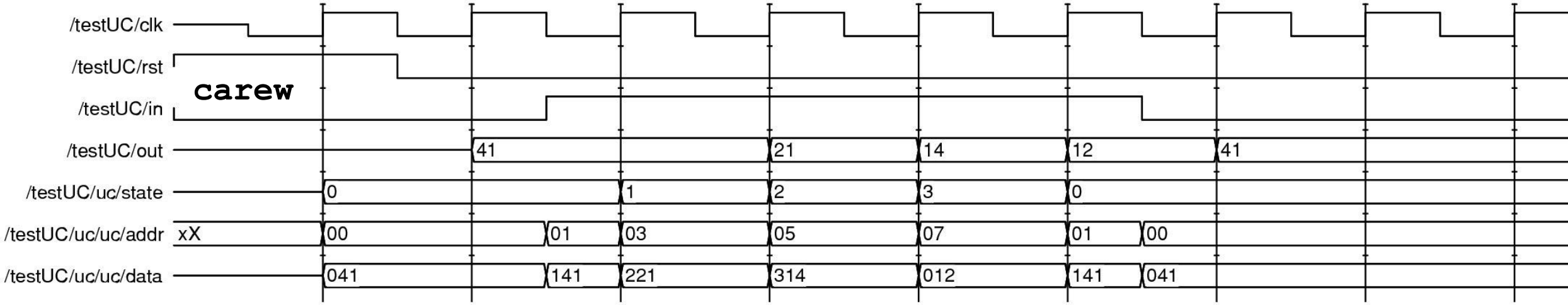
Microcoded Traffic Light Controller – in Verilog

Exercise: revise this Controller using 2 ROM's

```
module ucodeTLC(clk,rst,in,out) ;
  parameter n = 1 ; // input width
  parameter m = 6 ; // output width
  parameter k = 2 ; // bits of state

  input  clk, rst ;
  input  [n-1:0] in ;
  output [m-1:0] out ;

  wire   [k-1:0] next, state ;
  wire [k+m-1:0] uinst ;

  DFF #(k) state_reg(clk, next, state) ;  // state register
  DFF #(m) out_reg(clk, uinst[m-1:0], out) ; // output register
  ROM #(n+k,m+k) uc({state, in}, uinst) ; // microcode store
  assign next = rst ? {k{1'b0}} : uinst[m+k-1:m] ; // reset state
endmodule
```



Waveforms Of Light-Traffic Controller Microcode




```
00_100_001
01_100_001
10_010_001
10_010_001
11_001_100
11_001_100
00_001_010
00_001_010
```

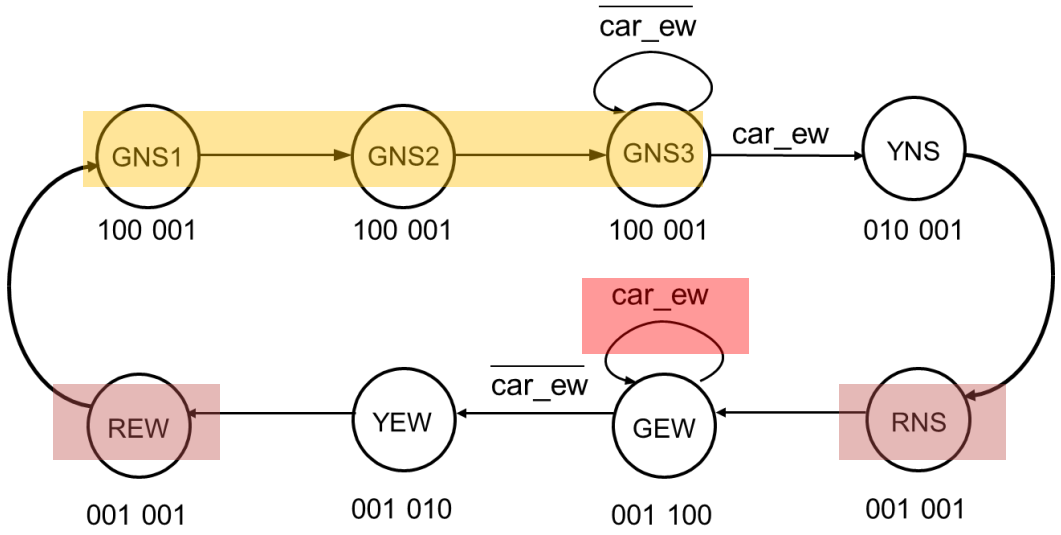
Next state

Output (rgb, rgb)

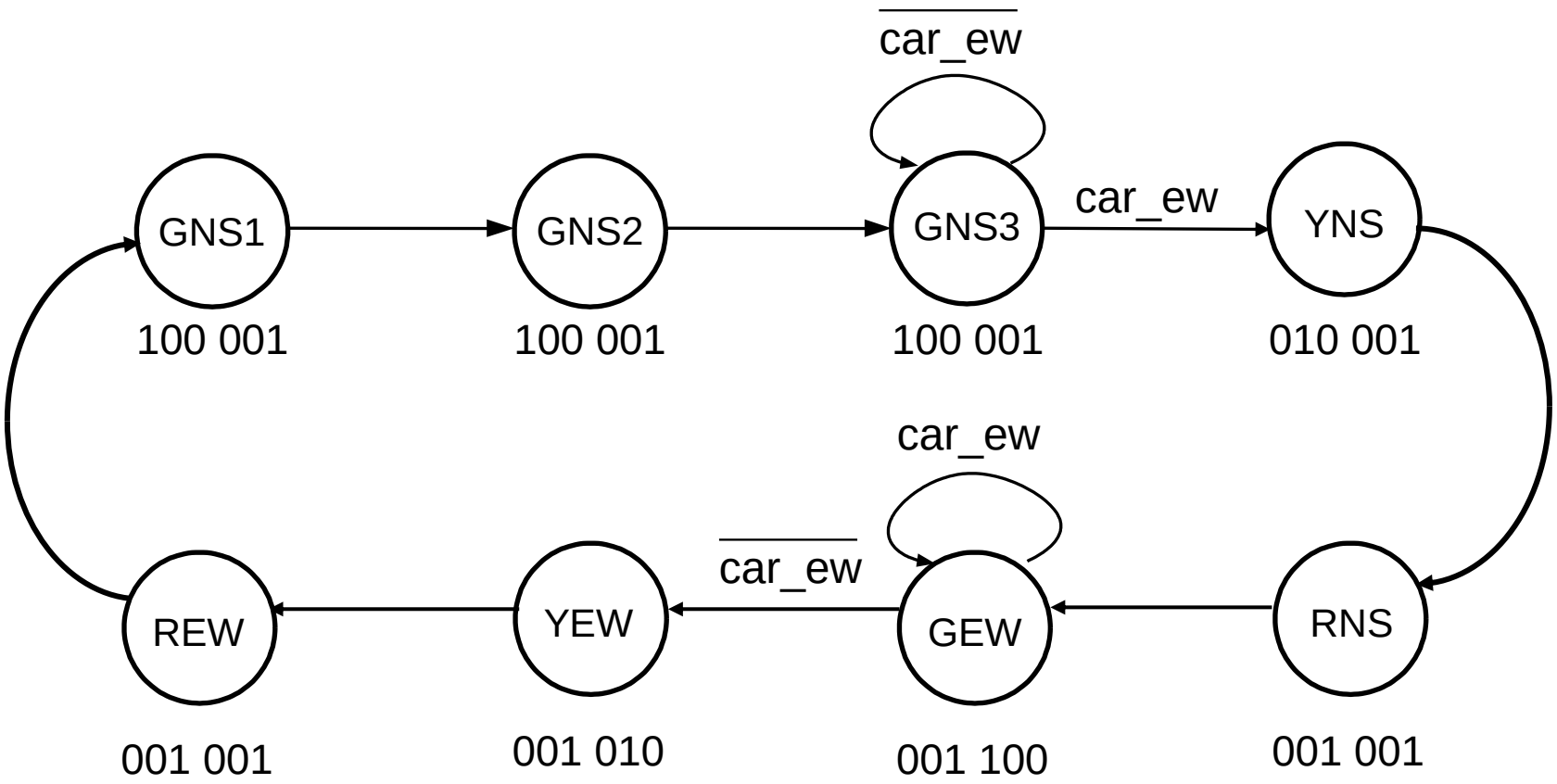
A New-and-Improved Light-Traffic Controller

不改設計，改state table，創造新的FSM

- Additional functions to original Light-Traffic Controller:
  - Light stays green in eat-west direction as long as car_ew is true.
  - Light stays green in north-south direction for a minimum of 3 states (GNS1, GNS2, and GNS3).
  - After a yellow light, lights should go red in both directions for 1 cycle before turning new light green.



Light-Traffic Control State Diagram



Light-Traffic Controller State Microcode

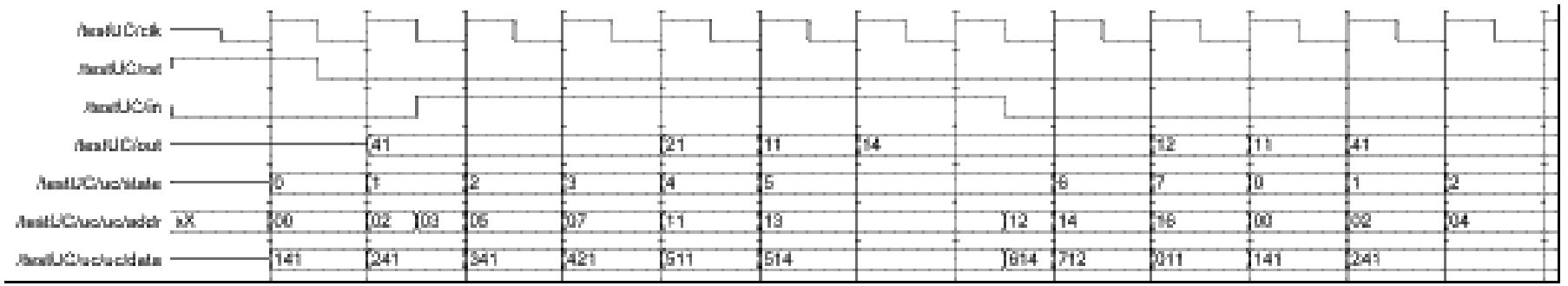
不改設計，改state table，創造新的FSM

Address	State	car_ew	Next State	Output	Data
0000	GNS1(000)	0	GNS2(001)	100001	001100001
0001	GNS1(000)	1	GNS2(001)	100001	001100001
0010	GNS2(001)	0	GNS3(010)	100001	010100001
0011	GNS2(001)	1	GNS3(010)	100001	010100001
0100	GNS3(010)	0	GNS3(010)	100001	010100001
0101	GNS3(010)	1	YNS (011)	100001	011100001
0110	YNS (011)	0	RNS (100)	010001	100010001
0111	YNS (011)	1	RNS (100)	010001	100010001
1000	RNS (100)	0	GEW (101)	001001	101001001
1001	RNS (100)	1	GEW (101)	001001	101001001
1010	GEW (101)	0	YEW (110)	001100	110001100
1011	GEW (101)	1	GEW (101)	001100	101001100
1100	YEW (110)	0	REW (111)	001010	111001010
1101	YEW (110)	1	REW (111)	001010	111001010
1110	REW (111)	0	GNS (000)	001001	000001001
1111	REW (111)	1	GNS (000)	001001	000001001

ROM address = {state, input}

ROM data = {next state, output}

Waveforms Of Light-Traffic Controller Microcode




```
001100001      101001001
001100001      101001001
010100001      110001100
010100001      101001100
010100001      111001010
011100001      111001010
100010001      000001001
100010001      000001001
```

FROM FSM TABLE TO INSTRUCTION SEQUENCING
邁向CPU設計之路

簡單ROM based table

- 缺點: 面積大
  - 尤其輸入很多的時候，指數型增加
- 觀察
  - 大部分時間next state 都單純指向下一個state
    - 不管輸入是0 或 1，和輸入沒半點關係
  - 少數需要branch到新的state
- 作法
  - 用一個microprogram counter (μPC) register紀錄接下來的state順序 (state sequencing)
    - 大部分時間就像counter，直接加一
    - 少部分時間load新值，branch到較遠的state
  - =>可以用程式控制=>microprogramming =>CPU 設計

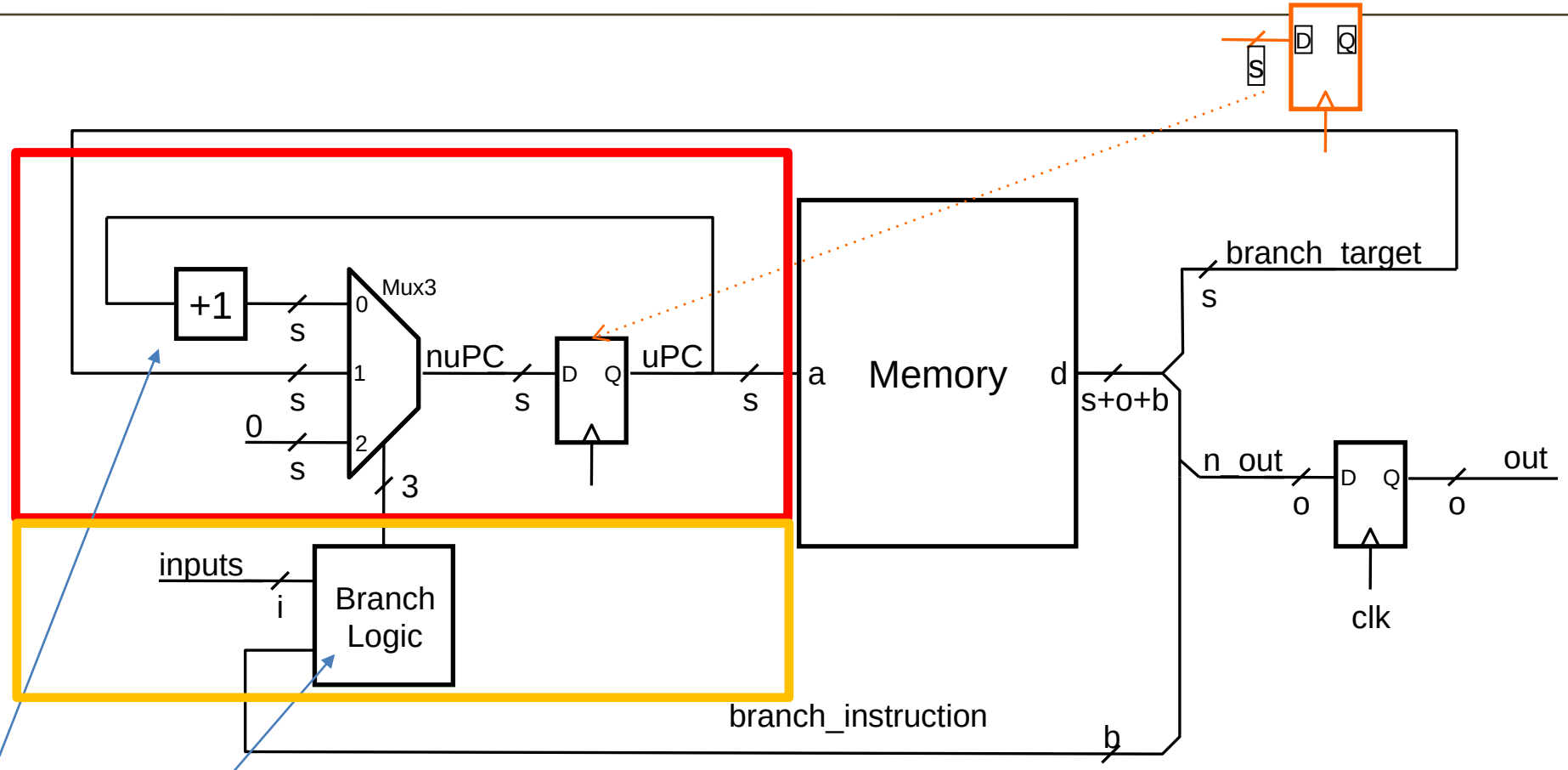
	State	Next State		Output
		!carew	carew	
gns	00	00	01	100001
yns	01	11	11	010001
gew	11	10	10	001100
yew	10	00	00	001010

address	data
000	00100001
001	01100001
010	11010001
011	11010001
110	10001100
111	10001100
100	00001010
101	00001010

Next state | output

ROM address = {state, input}
ROM data = {next state, output}

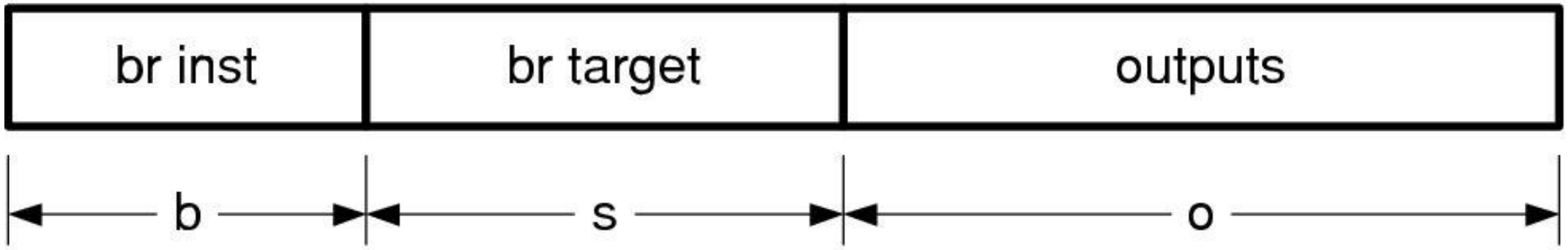
Instruction Sequencing – the picture



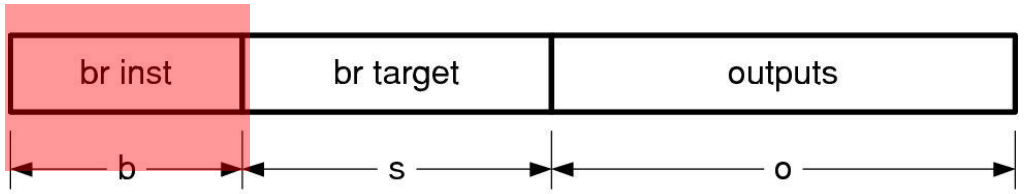
- 大部分時間就像counter，直接加一
- 少部分時間load新值，branch到較遠的state

Branching Logic

- Branch logic selects between μPC+1 and branch_target, depending on input and branch_instruction.
- Instructions are of the form branch *if f(inputs)*
  - For example *branch if car_ew or branch if not car_ew*.

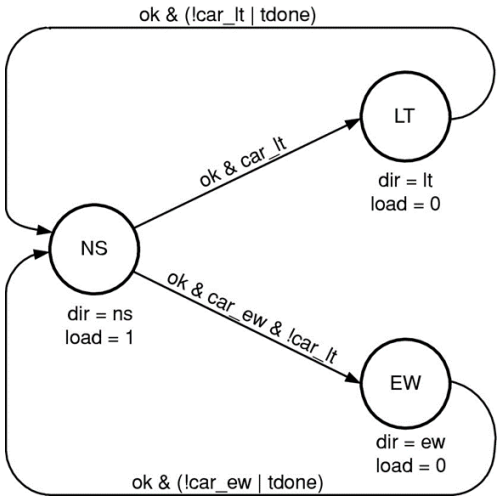


Branch Microinstructions



- Define several instructions for traffic light control:

Opcode	Encoding	Description
NOP	000	No branch – always go to next instruction
br	100	Always branch
brlt	001	Branch when left-turn car is detected
brnlt	101	Branch if no left-turn car is detected
brew	010	Branch when east-west car is detected
brnew	110	Branch if no east-west car is dected



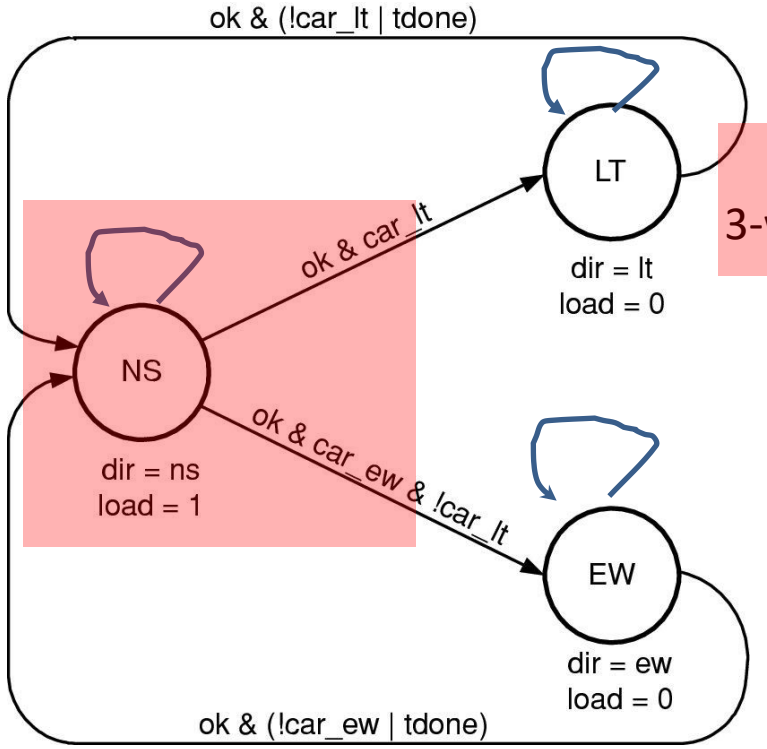
用之前有left turn的例子

branch = (brinst[0] & in[0] | brinst[1] & in[1]) ^ brinst[2] ;

Other encodings are possible

Brinst[2]: decide to branch or not
Brinst[1]: east-west car
Brinst[0]: left-turn car

Microcode Of Traffic-Light Controller With Branches



3-way

State	Addr	Inst	Target	Output	
nsg1	0000	brlt	lt1	100001001	green ns
nsg2	0001	brnew	nsg1	100001001	green ns
ew1	0010	nop		010001001	yellow ns
ew2	0011	brew	ew2	001001100	green ew
ew3	0100	br	nsg1	001001010	yellow ew
lt1	0101	nop		010001001	yellow ns
lt2	0110	brlt	lt2	001100001	green lt
lt3	0111	br	nsg1	001010001	yellow lt

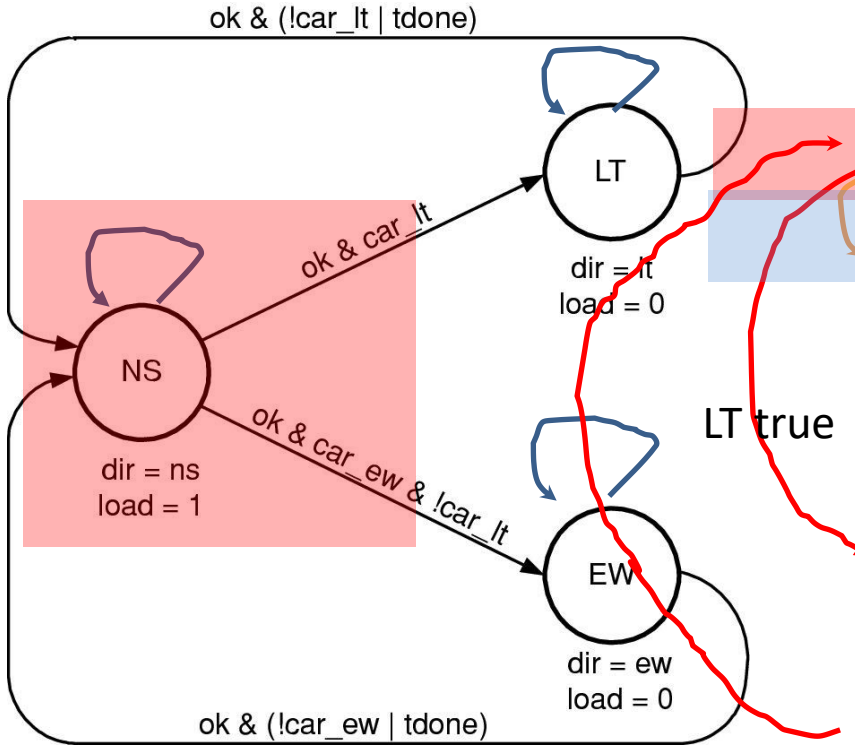
每個state預設繞回自己有畫出來

3-way: 因為每個branch只能跳往一個方向(預設往下一個)，3-way要用2個state (NS1,NS2) (預設往下執行)

Ex. 開始nsg1: if lt==truen, brlt, else next inst: nsg2

nsg2: brnew: no ew. If no ew == true, branch to self, nsg1, else next instr: ew1

Microcode Of Traffic-Light Controller With Branches

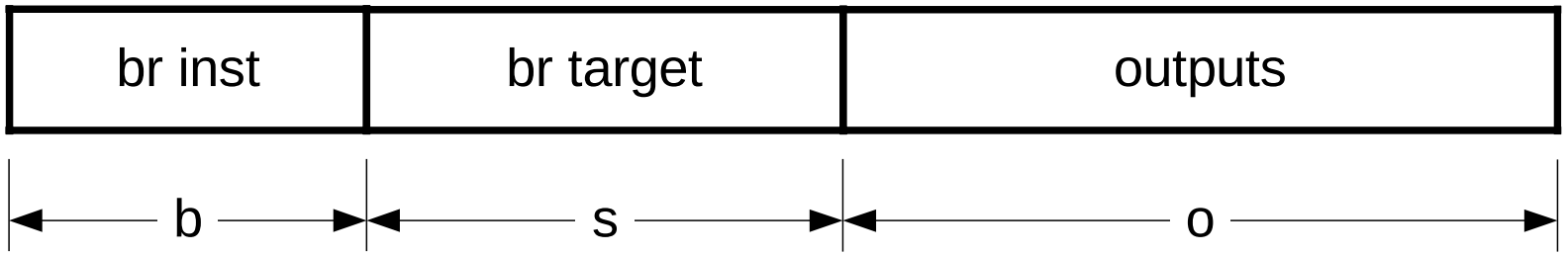


LT true

State	Addr	Inst	Target	Output	
nsg1	0000	brlt	lt1	100001001	green ns
nsg2	0001	brnew	nsg1	100001001	green ns
ew1	0010	nop		010001001	yellow ns
ew2	0011	brew	ew2	001001100	green ew
ew3	0100	br	nsg1	001001010	yellow ew
lt1	0101	nop		010001001	yellow ns
lt2	0110	brlt	lt2	001100001	green lt
lt3	0111	br	nsg1	001010001	yellow lt

每個state預設繞回自己有畫出來

Microinstruction Format



For our example:

b = 3

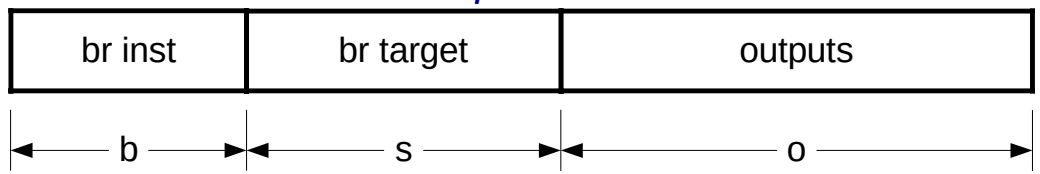
s = 4

o = 9

Implementing Microcode Of Light-Traffic Controller With Branches Using Verilog

```
module ucodeIS(clk,rst,in,out) ;
  parameter n = 2 ; // input width
  parameter m = 9 ; // output width
  parameter k = 4 ; // bits of state
  parameter j = 3 ; // bits of instruction

  input  clk, rst ;
  input  [n-1:0] in ;
  output [m-1:0] out ;
  wire   [k-1:0] nupc, upc ; // microprogram counter
  wire [j+k+m-1:0] uinst ;   // microinstruction word
  // split off fields of microinstruction
  wire [m-1:0] nxt_out ; // = uinst[m-1:0] ;
  wire [k-1:0] br_upc  ; // = uinst[m+k-1:m] ;
  wire [j-1:0] brinst  ; // = uinst[m+j+k-1:m+k] ;
  assign {brinst, br_upc, nxt_out} = uinst ;

  DFF #(k) upc_reg(clk, nupc, upc) ;  // microprogram counter
  DFF #(m) out_reg(clk, nxt_out, out) ; // output register
  ROM #(k,m+k+j) uc(upc, uinst) ; // microcode store

  // branch instruction decode; branch to new or next (+1) address
  wire branch = (brinst[0] & in[0] | brinst[1] & in[1]) ^ brinst[2] ;
  // sequencer
  assign nupc = rst ? {k{1'b0}} : branch ? br_upc : upc + 1'b1 ;
endmodule
```

Ref. L5 SystemVerilog p. 35-37

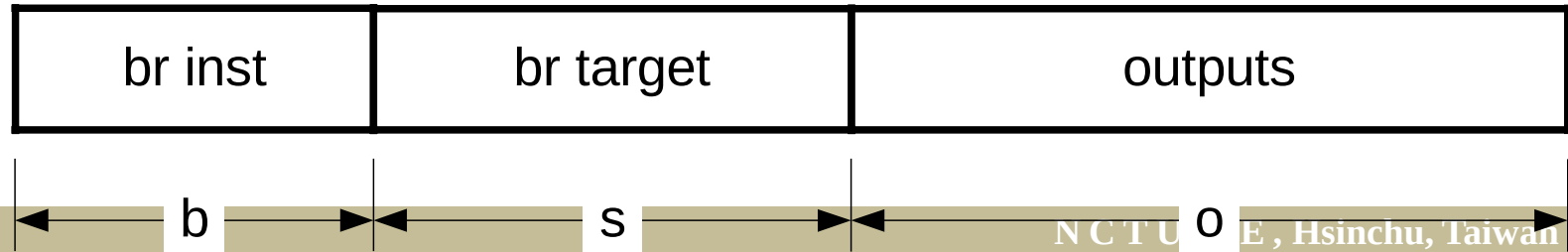


Microcode Of Light-Traffic Controller With Left Turn

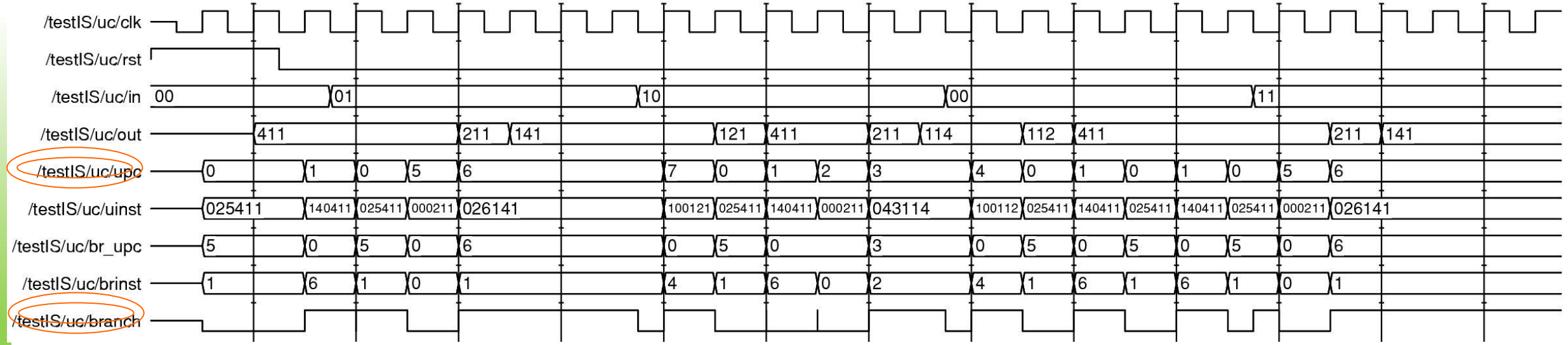
Traffic light

Address	State	Br Inst	Target	NS LT EW	Data
0000	NS1	BLT (001)	LT1(0101)	10000100	0010101100001001
0001	NS2	BNEW (110)	NS1(0000)	10000100	1100000100001001
0010	EW1	NOP (000)		01000100	0000000010001001
0011	EW2	BEW (010)	EW2(0011)	00100110	0100110001001100
0100	EW3	BR (100)	NS1(0000)	00100101	1000000001001010
0101	LT1	NOP (000)		01000100	0000000010001001
0110	LT2	BLT (001)	LT2(0110)	00110000	0010110001100001
0111	LT3	BR (100)	NS1(0000)	00101000	1000000001010001

```
Data = {instruction_code, branch address, light_control}
```



Waveforms Of Light-Traffic Controller With Left Turn Microcode



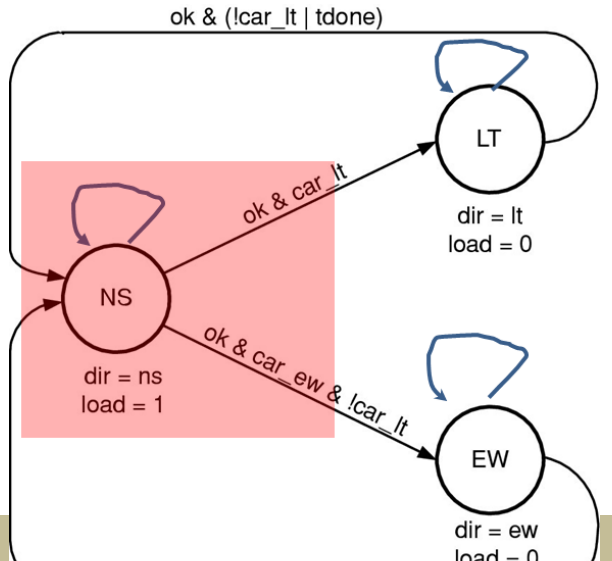
Address	State	Br Inst	Target	NS LT EW	Data
0000	NS1	BLT (001)	LT1(0101)	100001001	0010101100001001
0001	NS2	BNEW (110)	NS1(0000)	100001001	1100000100001001
0010	EW1	NOP (000)		010001001	0000000010001001
0011	EW2	BEW (010)	EW2(0011)	001001100	0100011001001100
0100	EW3	BR (100)	NS1(0000)	001001010	1000000001001010
0101	LT1	NOP (000)		010001001	0000000010001001
0110	LT2	BLT (001)	LT2(0110)	001100001	0010110001100001
0111	LT3	BR (100)	NS1(0000)	001010001	1000000001010001

```
0010101100001001
1100000100001001
0000000010001001
0100011001001100
1000000001001010
0000000010001001
0010110001100001
1000000001010001
```

Alternate Microcode For Light-Traffic Controller With Left Turn (多方向branch指令)

BNA: branch on not any input is true

Address	State	Br Inst	Target	NS LT EW	Data
0000	NS1	BNA (111)	NS1(0000)	100001001	1110000100001001
0001	NS2	BLT (001)	LT1(0100)	010001001	0010100010001001
0010	EW1	BEW (010)	EW1(0010)	001001100	0100010001001100
0011	EW2	BR (100)	NS1(0000)	001001010	1000000001001010
0100	LT1	BLT (001)	LT1(0100)	001100001	0010100001100001
0101	LT2	BR (100)	NS1(0000)	001010001	1000000001010001



Previous microcode needs:(due to only branch one way in one instruct)

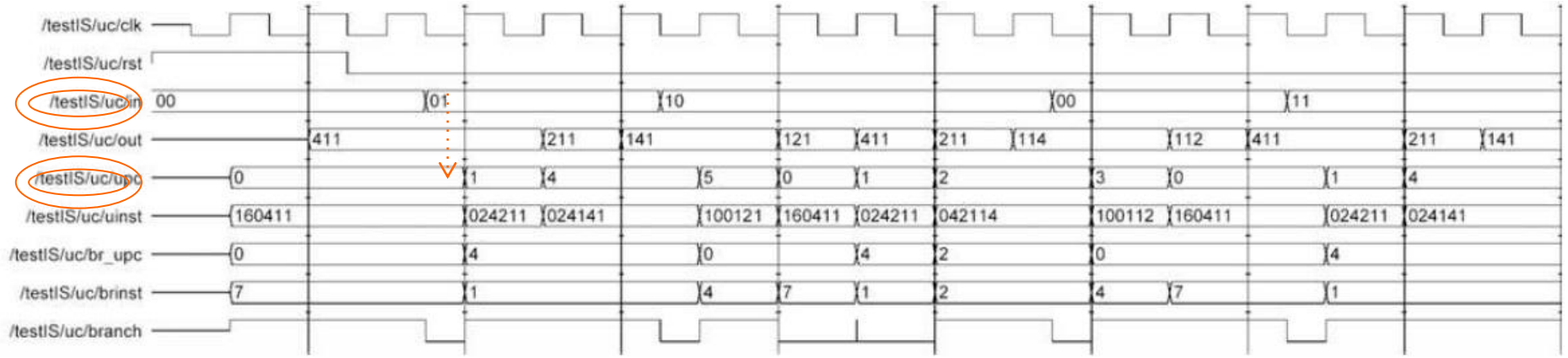
- 2 states with GNS light (NS1 & NS2)
- 2 states with YNS light (EW1 & LT1)

By adding a new branch instruction – BNA (branch on "not any")

- 1 state with GNS light (NS1)
- 1 state with YNS light (NS2)

BNA: 沒有任意輸入條件為true，才跳

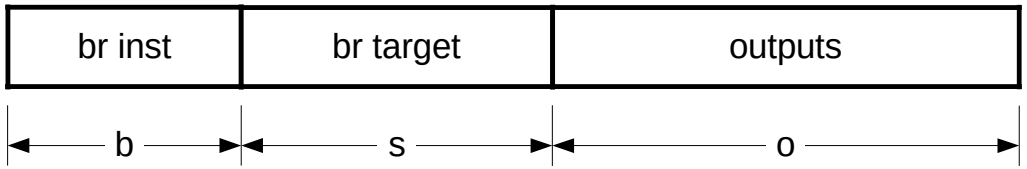
Waveforms Of Alternate Microcode



Address	State	Br Inst	Target	NS LT EW	Data
0000	NS1	BNA (111)	NS1(0000)	100001001	1110000100001001
0001	NS2	BLT (001)	LT1(0100)	010001001	0010100010001001
0010	EW1	BEW (010)	EW1(0010)	001001100	0100010001001100
0011	EW2	BR (100)	NS1(0000)	001001010	1000000001001010
0100	LT1	BLT (001)	LT1(0100)	001100001	0010100001100001
0101	LT2	BR (100)	NS1(0000)	001010001	1000000001010001

Multiple Instruction Types (指令可以縮短嗎)

- For some FSMs the micro-instruction word can start getting a bit long.
  - 要跳的位址很遠的話，指令會變很長

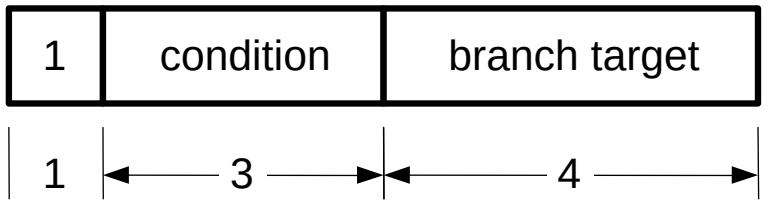


- To shorten it, we observe:
  - Not every state needs a branch
  - Not every output changes on every state
- Define instruction that does just a branch or a load of one register:
  - brx – 1yyyvvvv - branch to value vvvv on condition yyy
  - ldx - 0yyyvvvv - load register yyy with value vvvv

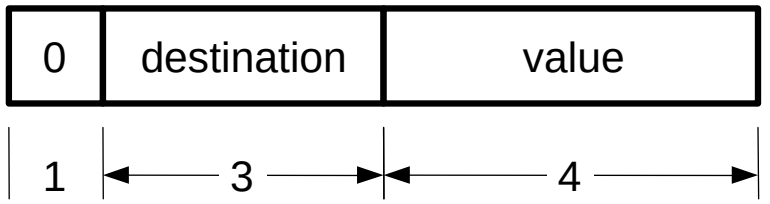
Instruction Format Of Microcode With 2 Instruction Types (一次用一種)

- Branch Instructions:
  - Operates branch mux as before
  - No write to output registers
- Load Instructions:
  - Sets branch mux to +1
  - Update selected output register:
    - Can include other datapath components – e.g., timer in place of an output register

Branch Instruction



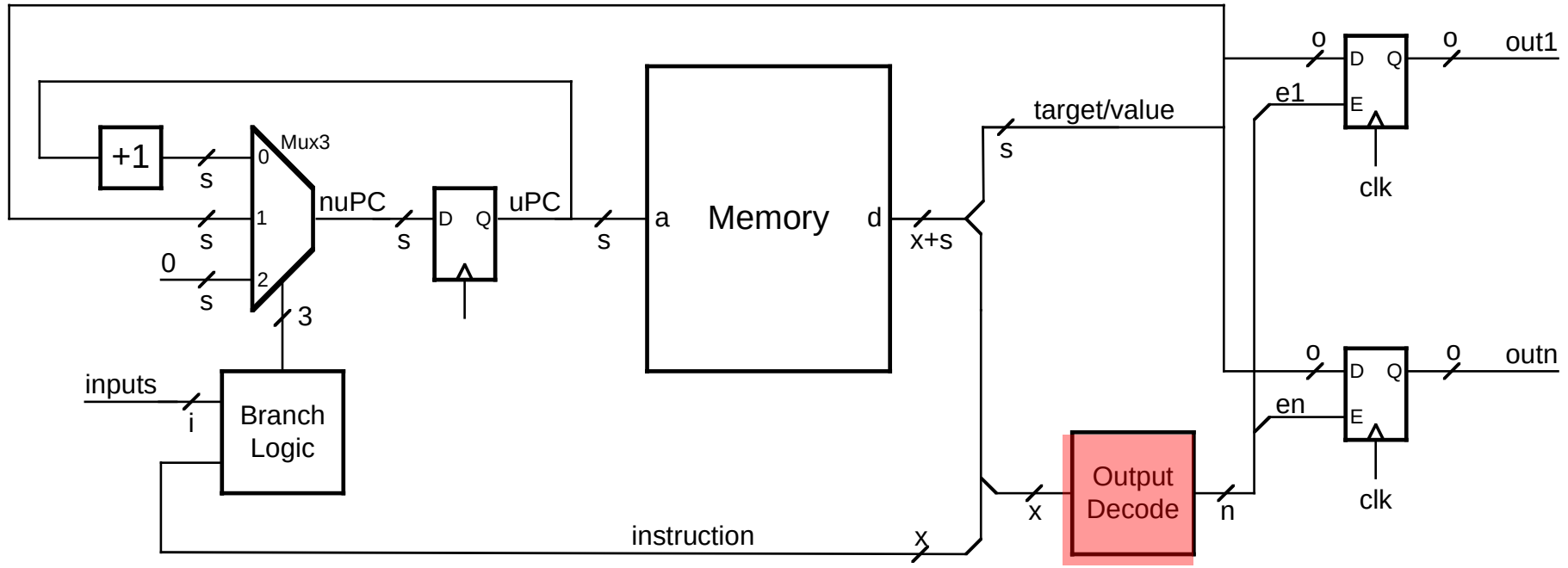
Load Instruction



One step on the path to a processor
邁向CPU設計之路

State table → Branch Inst → Branch and Load Insts → Full Instruction Set

Block Diagram Of Microcode With Output Instructions



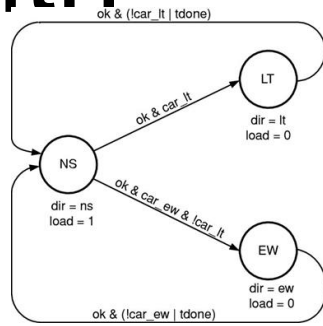

```
//reduce instruction space with brx or ldx
```

Microcode For Traffic-Light Controller With brx & ldx Instructions

- Registers to load:
  - ldns – load north/south light with value
  - ldlt – load left-turn light with value
  - ldew – load east/west light with value
  - ltim1 – load timer 1 with value – starts timer

??: 有沒有覺得哪裡怪怪的?每個register load 值都要一個指令

Microcode For Traffic-Light Controller With brx & ldx Instructions (Cont)



	State	Addr	Inst	Value
	rst1	00000	ldlt	RED
	rst2	00001	ldew	RED
NS Green	ns1	00010	ldns	GREEN
	ns3	00011	ltim	T_GREEN
	ns4	00100	bntz	ns4
Until input	ns5	00101	brnle	ns5
	ns6	00110	ldns	YELLOW
NS Yellow to Red	ns7	00111	ltim	T_YELLOW
	ns8	01000	bntz	ns8
	ns9	01001	ldns	RED
EW or LT?	ns10	01010	blt	lt1
	ew1	01011	ltim	T_RED
Wait for NS Red, make EW Green	ew2	01100	bntz	ew2
	ew3	01101	ldew	GREEN
	ew4	01110	ltim	T_GREEN
	ew5	01111	bntz	ew5
	ew6	10000	ldew	YELLOW
EW to Red	ew7	10001	ltim	T_YELLOW
	ew8	10010	bntz	ew8
	ew9	10011	ldew	RED
Back to NS	ew10	10100	br	ns1
	lt1	10101	ltim	T_RED
Wait for NS Red, make LT Green	lt2	10110	bntz	lt2
	lt3	10111	ldlt	GREEN
	lt4	11000	ltim	T_GREEN
	lt5	11001	bntz	lt5
	lt6	11010	ldlt	YELLOW
LT to Red	lt7	11011	ltim	T_YELLOW
	lt8	11100	bntz	lt8
	lt9	11101	ldlt	RED
Back to NS	lt10	11110	br	ns1

bntz: branch if not timer zero
brnle: branch if no left or EW input

Implementing Traffic-Light Controller Microcode With brx & ldx Instructions In Verilog

```
//-----------------------------------------------------------------
module ucodeMI(clk,rst,in,out) ;
  parameter n = 2 ; // input width
  parameter m = 9 ; // output width
  parameter o = 3 ; // output sub-width
  parameter k = 5 ; // bits of state
  parameter j = 4 ; // bits of instruction

  input  clk, rst ;
  input  [n-1:0] in ;
  output [m-1:0] out ;

  wire   [k-1:0] nupc, upc ; // microprogram counter
  wire [j+k-1:0] uinst ;   // microinstruction word
  wire done ; // timer done signal

  // split off fields of microinstruction
  wire opcode ; // opcode bit
  wire [j-2:0] inst ;  // condition for branch, dest for store
  wire [k-1:0] value ; // target for branch, value for store
  assign {opcode, inst, value} = uinst ;
```

To be continued on next page…

Implementing Traffic-Light Controller Microcode With brx & ldx Instructions In Verilog (Cont)

```
  DFF #(k) upc_reg(clk, nupc, upc) ;  // microprogram counter
  ROM #(k,k+j) uc(upc, uinst) ; // microcode store

  // output registers and timer
  DFFE #(o) or0(clk, e[0], value[o-1:0], out[o-1:0]) ;     // NS
  DFFE #(o) or1(clk, e[1], value[o-1:0], out[2*o-1:o]) ;   // EW
  DFFE #(o) or2(clk, e[2], value[o-1:0], out[3*o-1:2*o]) ; // LT
  Timer #(k) tim(clk, rst, e[3], value, done) ;            // timer

  // enable for output registers and timer
  wire [3:0] e = opcode ? 4'b0 : 1<<inst ;

  // branch instruction decode
  wire branch = opcode ? (inst[2] ^ (((inst[1:0] == 0) & in[0]) | // BLT
                                     ((inst[1:0] == 1) & in[1]) | // BEW
                                    ((inst[1:0] == 2) & (in[0]|in[1])) | //BLE
                                    ((inst[1:0] == 3) & done))) // BTD
                       : 1'b0 ; // for a store opcode

  // microprogram counter
  assign nupc = rst ? {k{1'b0}} : branch ? value : upc + 1'b1 ;
endmodule
```

Extending this to a processor

- Add three new instructions //like RISC
  - ADD //add data and keep the result in registers
    - R2 <- R1+R0
  - LOAD //load data from memory
    - R2 <- M[R1+R0]
  - STORE //store data back to memory
    - M[R1+R0] <- R2
- And add registers R0, R1, R2 – can also target these with LDR
- Opcode 000/Branch, 001/LDR, 010/ADD, 011/LOAD, 100/STORE
- Branch and LDR take condition/register and value fields
- ADD, LOAD, STORE ignore these fields

- Can tweak the instruction set to make this more efficient.
  - -> more details about RISC design, refer computer organization and architecture.

Summary

- Microcode is just FSM implemented with a ROM or RAM
  - One address for each state x input combination
  - Address contains next state and output
- **Adding a *sequencer* reduces size of ROM/RAM**
  - **One entry per state rather than 2^i**
  - **uPC, incrementer, branch address, and branch control**
- **Adding instruction types reduces width of ROM/RAM**
  - **Branch *or* output in each instruction – rather than both**
  - **Type field specifies which one**
- One step away from a full processor
  - Just add more instructions