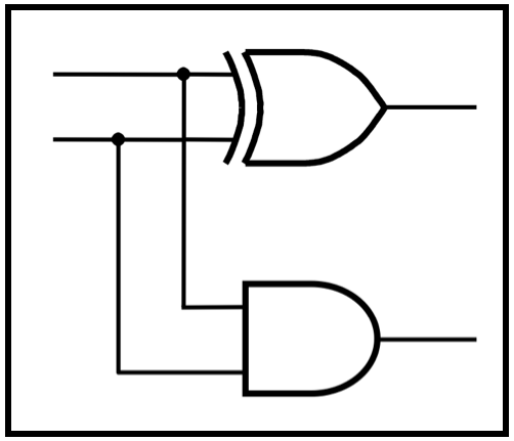


CprE 2810: Digital Logic

Instructor: Alexander Stoytchev

http://www.ece.iastate.edu/~alexs/classes/

Intro to Assembly Language

CprE 2810: Digital Logic
Iowa State University, Ames, IA

Intro to Assembly Language
(for the i281 CPU)

CprE 2810: Digital Logic
Iowa State University, Ames, IA

Intel 8086 Example

```
; _memcpy(dst, src, len)
; Copy a block of memory from one location to another.
;
; Entry stack parameters
;       [BP+6] = len, Number of bytes to copy
;       [BP+4] = src, Address of source data block
;       [BP+2] = dst, Address of target data block
;
; Return registers
;       AX = Zero

0000:1000                       org     1000h       ; Start at 0000:1000h

0000:1000               _memcpy proc
0000:1000 55                    push    bp          ; Set up the call frame
0000:1001 89 E5                 mov     bp,sp
0000:1003 06                    push    es          ; Save ES
0000:1004 8B 4E 06              mov     cx,[bp+6]   ; Set CX = len
0000:1007 E3 11                 jcxz    done        ; If len=0, return
0000:1009 8B 76 04              mov     si,[bp+4]   ; Set SI = src
0000:100C 8B 7E 02              mov     di,[bp+2]   ; Set DI = dst
0000:100F 1E                    push    ds          ; Set ES = DS
0000:1010 07                    pop     es

0000:1011 8A 04         loop    mov     al,[si]     ; Load AL from [src]
0000:1013 88 05                 mov     [di],al     ; Store AL to [dst]
0000:1015 46                    inc     si          ; Increment src
0000:1016 47                    inc     di          ; Increment dst
0000:1017 49                    dec     cx          ; Decrement len
0000:1018 75 F7                 jnz     loop        ; Repeat the loop

0000:101A 07            done    pop     es          ; Restore ES
0000:101B 5D                    pop     bp          ; Restore previous call frame
0000:101C 29 C0                 sub     ax,ax       ; Set AX = 0
0000:101E C3                    ret                 ; Return
0000:101F                       end proc
```

[http://en.wikipedia.org/wiki/Intel_8086]

Intel 8086 Example

Memory Address

```
                        ; _memcpy(dst, src, len)
                        ; Copy a block of memory from one location to another.
                        ;
                        ; Entry stack parameters
                        ;       [BP+6] = len, Number of bytes to copy
                        ;       [BP+4] = src, Address of source data block
                        ;       [BP+2] = dst, Address of target data block
                        ;
                        ; Return registers
                        ;       AX = Zero

0000:1000                               org     1000h           ; Start at 0000:1000h

0000:1000               _memcpy         proc
0000:1000 55                            push    bp              ; Set up the call frame
0000:1001 89 E5                         mov     bp,sp
0000:1003 06                            push    es              ; Save ES
0000:1004 8B 4E 06                      mov     cx,[bp+6]       ; Set CX = len
0000:1007 E3 11                         jcxz    done            ; If len=0, return
0000:1009 8B 76 04                      mov     si,[bp+4]       ; Set SI = src
0000:100C 8B 7E 02                      mov     di,[bp+2]       ; Set DI = dst
0000:100F 1E                            push    ds              ; Set ES = DS
0000:1010 07                            pop     es

0000:1011 8A 04         loop            mov     al,[si]         ; Load AL from [src]
0000:1013 88 05                         mov     [di],al         ; Store AL to [dst]
0000:1015 46                            inc     si              ; Increment src
0000:1016 47                            inc     di              ; Increment dst
0000:1017 49                            dec     cx              ; Decrement len
0000:1018 75 F7                         jnz     loop            ; Repeat the loop

0000:101A 07            done            pop     es              ; Restore ES
0000:101B 5D                            pop     bp              ; Restore previous call frame
0000:101C 29 C0                         sub     ax,ax           ; Set AX = 0
0000:101E C3                            ret                     ; Return
0000:101F                               end proc
```

[http://en.wikipedia.org/wiki/Intel_8086]

Intel 8086 Example

Machine Language

```
                        ; _memcpy(dst, src, len)
                        ; Copy a block of memory from one location to another.
                        ;
                        ; Entry stack parameters
                        ;       [BP+6] = len, Number of bytes to copy
                        ;       [BP+4] = src, Address of source data block
                        ;       [BP+2] = dst, Address of target data block
                        ;
                        ; Return registers
                        ;       AX = Zero

0000:1000                           org     1000h       ; Start at 0000:1000h

0000:1000               _memcpy     proc
0000:1000 55                        push    bp          ; Set up the call frame
0000:1001 89 E5                     mov     bp,sp
0000:1003 06                        push    es          ; Save ES
0000:1004 8B 4E 06                  mov     cx,[bp+6]   ; Set CX = len
0000:1007 E3 11                     jcxz    done        ; If len=0, return
0000:1009 8B 76 04                  mov     si,[bp+4]   ; Set SI = src
0000:100C 8B 7E 02                  mov     di,[bp+2]   ; Set DI = dst
0000:100F 1E                        push    ds          ; Set ES = DS
0000:1010 07                        pop     es

0000:1011 8A 04         loop        mov     al,[si]     ; Load AL from [src]
0000:1013 88 05                     mov     [di],al     ; Store AL to [dst]
0000:1015 46                        inc     si          ; Increment src
0000:1016 47                        inc     di          ; Increment dst
0000:1017 49                        dec     cx          ; Decrement len
0000:1018 75 F7                     jnz     loop        ; Repeat the loop

0000:101A 07            done        pop     es          ; Restore ES
0000:101B 5D                        pop     bp          ; Restore previous call frame
0000:101C 29 C0                     sub     ax,ax       ; Set AX = 0
0000:101E C3                        ret                 ; Return
0000:101F                           end proc
```

[http://en.wikipedia.org/wiki/Intel_8086]

Intel 8086 Example

```
; _memcpy(dst, src, len)
; Copy a block of memory from one location to another.
;
; Entry stack parameters
;       [BP+6] = len, Number of bytes to copy
;       [BP+4] = src, Address of source data block
;       [BP+2] = dst, Address of target data block
;
; Return registers
;       AX = Zero
```

Assembly Language

```
0000:1000                       org     1000h           ; Start at 0000:1000h

0000:1000                       _memcpy proc
0000:1000 55                    push    bp              ; Set up the call frame
0000:1001 89 E5                 mov     bp,sp
0000:1003 06                    push    es              ; Save ES
0000:1004 8B 4E 06              mov     cx,[bp+6]       ; Set CX = len
0000:1007 E3 11                 jcxz    done            ; If len=0, return
0000:1009 8B 76 04              mov     si,[bp+4]       ; Set SI = src
0000:100C 8B 7E 02              mov     di,[bp+2]       ; Set DI = dst
0000:100F 1E                    push    ds              ; Set ES = DS
0000:1010 07                    pop     es

0000:1011 8A 04                 loop    mov     al,[si] ; Load AL from [src]
0000:1013 88 05                 mov     [di],al         ; Store AL to [dst]
0000:1015 46                    inc     si              ; Increment src
0000:1016 47                    inc     di              ; Increment dst
0000:1017 49                    dec     cx              ; Decrement len
0000:1018 75 F7                 jnz     loop            ; Repeat the loop

0000:101A 07                    done    pop     es      ; Restore ES
0000:101B 5D                    pop     bp              ; Restore previous call frame
0000:101C 29 C0                 sub     ax,ax           ; Set AX = 0
0000:101E C3                    ret                     ; Return
0000:101F                       end proc
```

[http://en.wikipedia.org/wiki/Intel_8086]

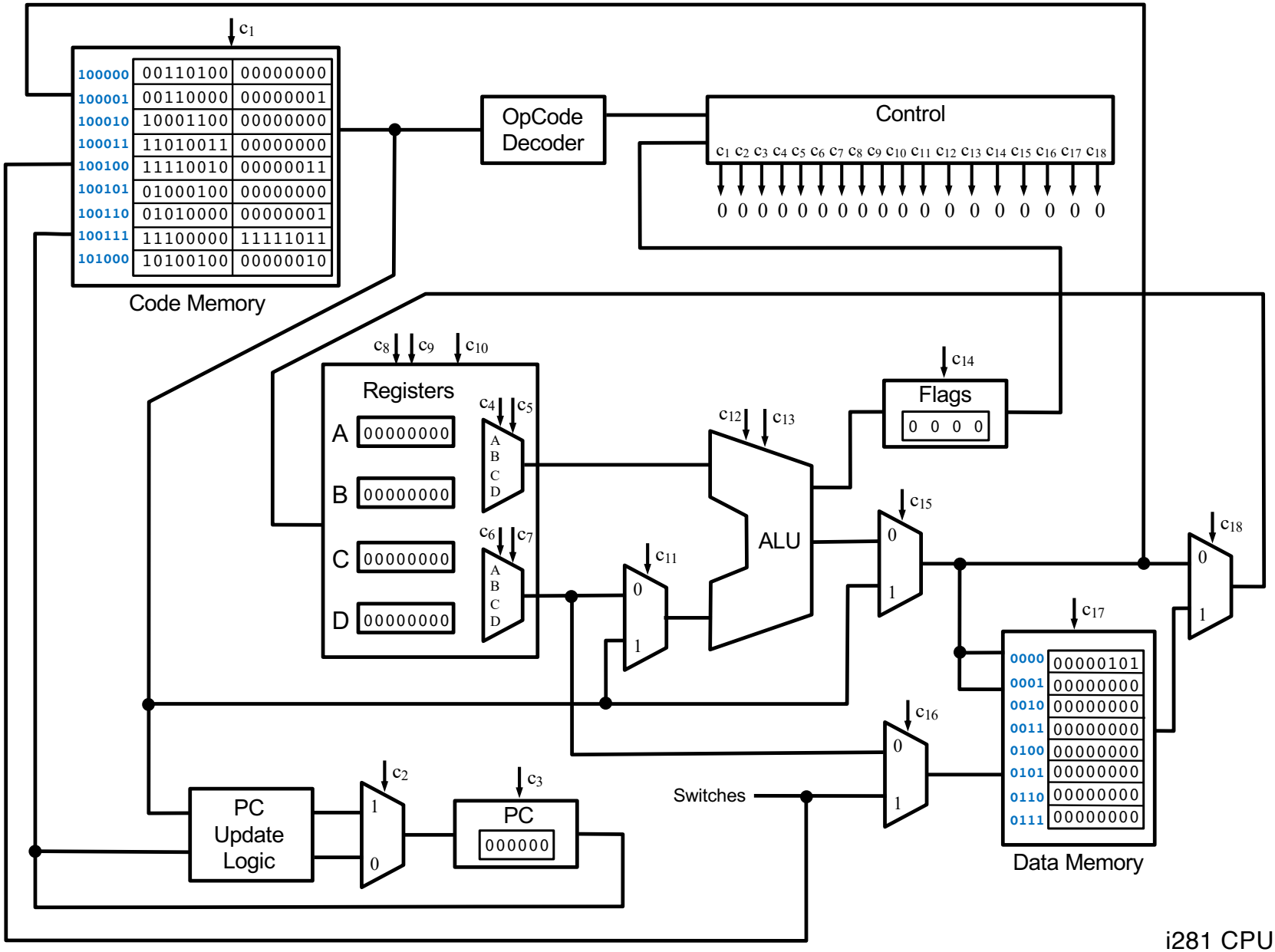
Intel 8086 Example

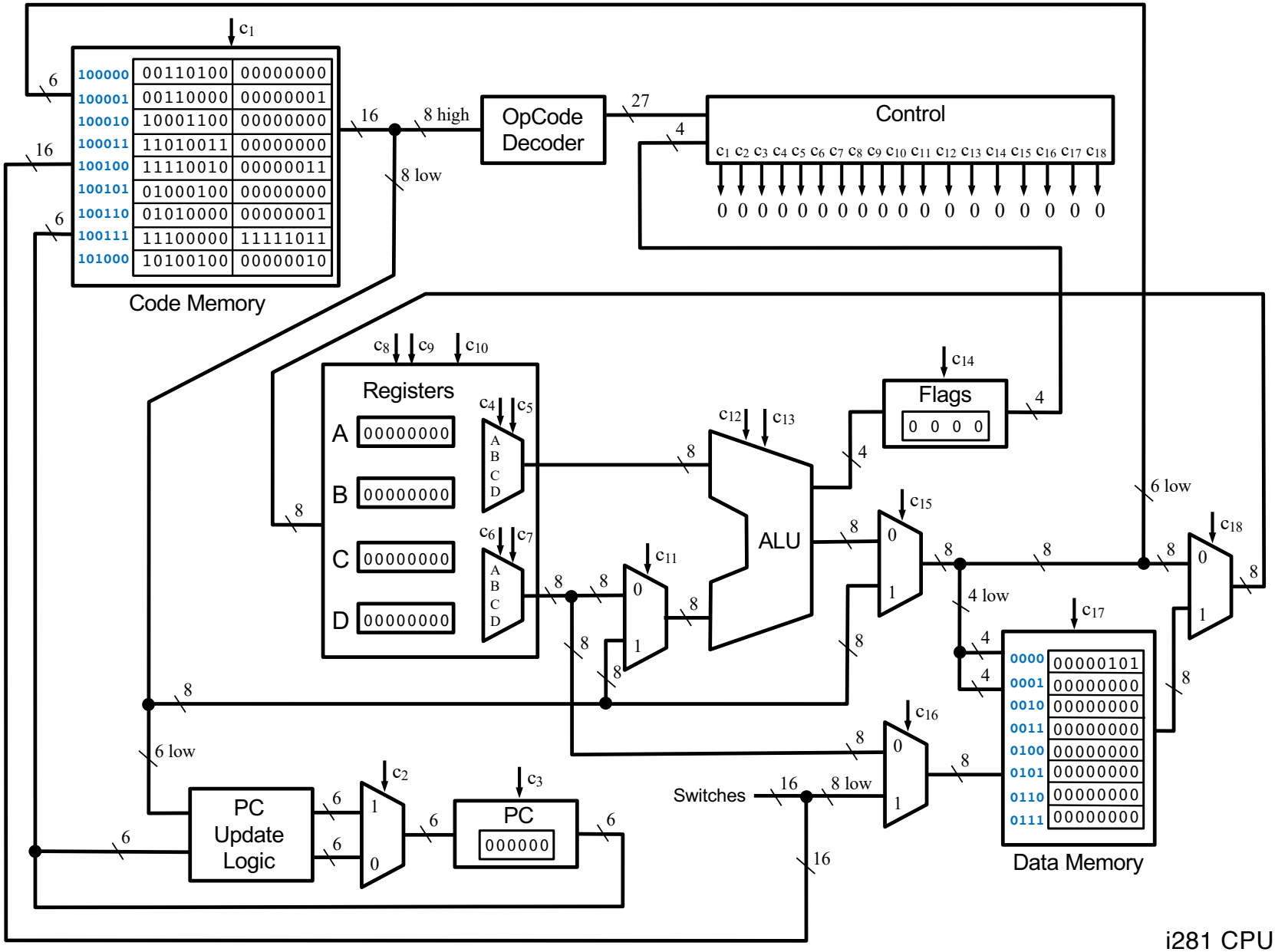
```
; _memcpy(dst, src, len)
; Copy a block of memory from one location to another.
;
; Entry stack parameters
;       [BP+6] = len, Number of bytes to copy
;       [BP+4] = src, Address of source data block
;       [BP+2] = dst, Address of target data block
;
; Return registers
;       AX = Zero

0000:1000                       org     1000h       ; Start at 0000:1000h

0000:1000               _memcpy proc
0000:1000 55                    push    bp          ; Set up the call frame
0000:1001 89 E5                 mov     bp,sp
0000:1003 06                    push    es          ; Save ES
0000:1004 8B 4E 06              mov     cx,[bp+6]   ; Set CX = len
0000:1007 E3 11                 jcxz    done        ; If len=0, return
0000:1009 8B 76 04              mov     si,[bp+4]   ; Set SI = src
0000:100C 8B 7E 02              mov     di,[bp+2]   ; Set DI = dst
0000:100F 1E                    push    ds          ; Set ES = DS
0000:1010 07                    pop     es

0000:1011 8A 04         loop    mov     al,[si]     ; Load AL from [src]
0000:1013 88 05                 mov     [di],al     ; Store AL to [dst]
0000:1015 46                    inc     si          ; Increment src
0000:1016 47                    inc     di          ; Increment dst
0000:1017 49                    dec     cx          ; Decrement len
0000:1018 75 F7                 jnz     loop        ; Repeat the loop

0000:101A 07            done    pop     es          ; Restore ES
0000:101B 5D                    pop     bp          ; Restore previous call frame
0000:101C 29 C0                 sub     ax,ax       ; Set AX = 0
0000:101E C3                    ret                 ; Return
0000:101F                       end proc
```

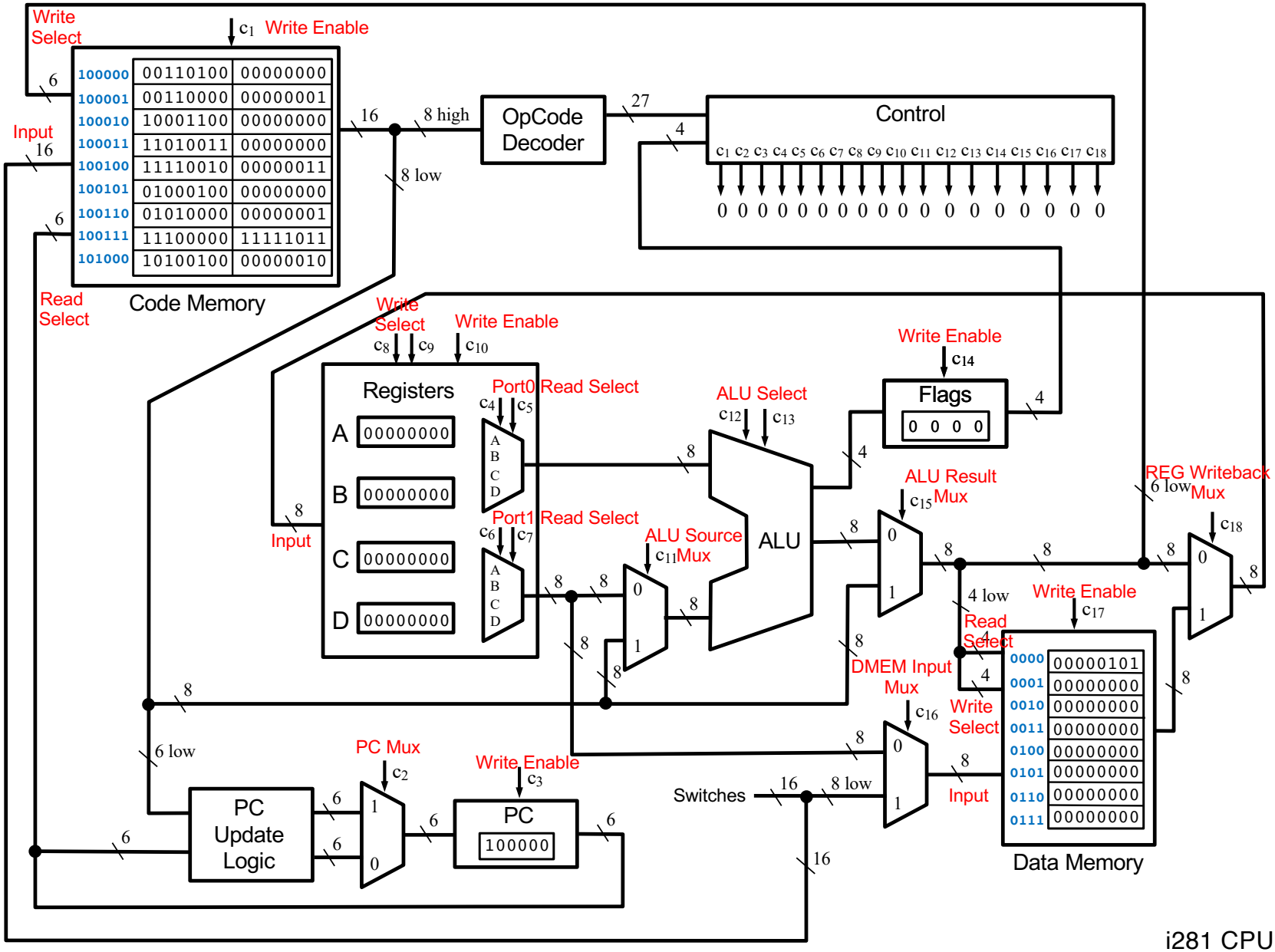
Comments

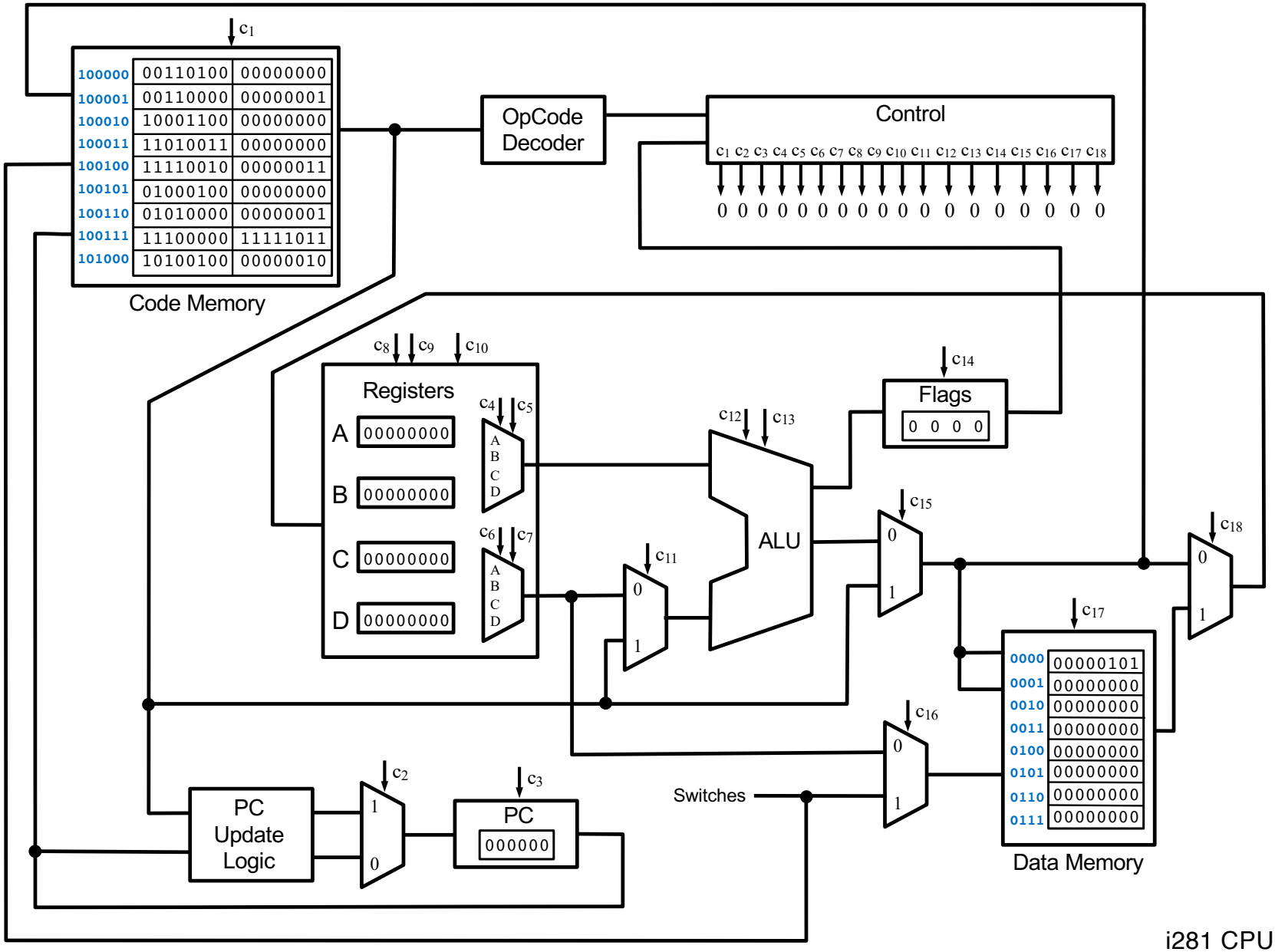
[http://en.wikipedia.org/wiki/Intel_8086]

i281 CPU Architecture


c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
OpCode Decoder
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
c11
c12 c13
ALU
c14
Flags
0 0 0 0
c15
c16
c17
c18
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
Switches
c2
c3
PC Update Logic
PC
000000
i281 CPU

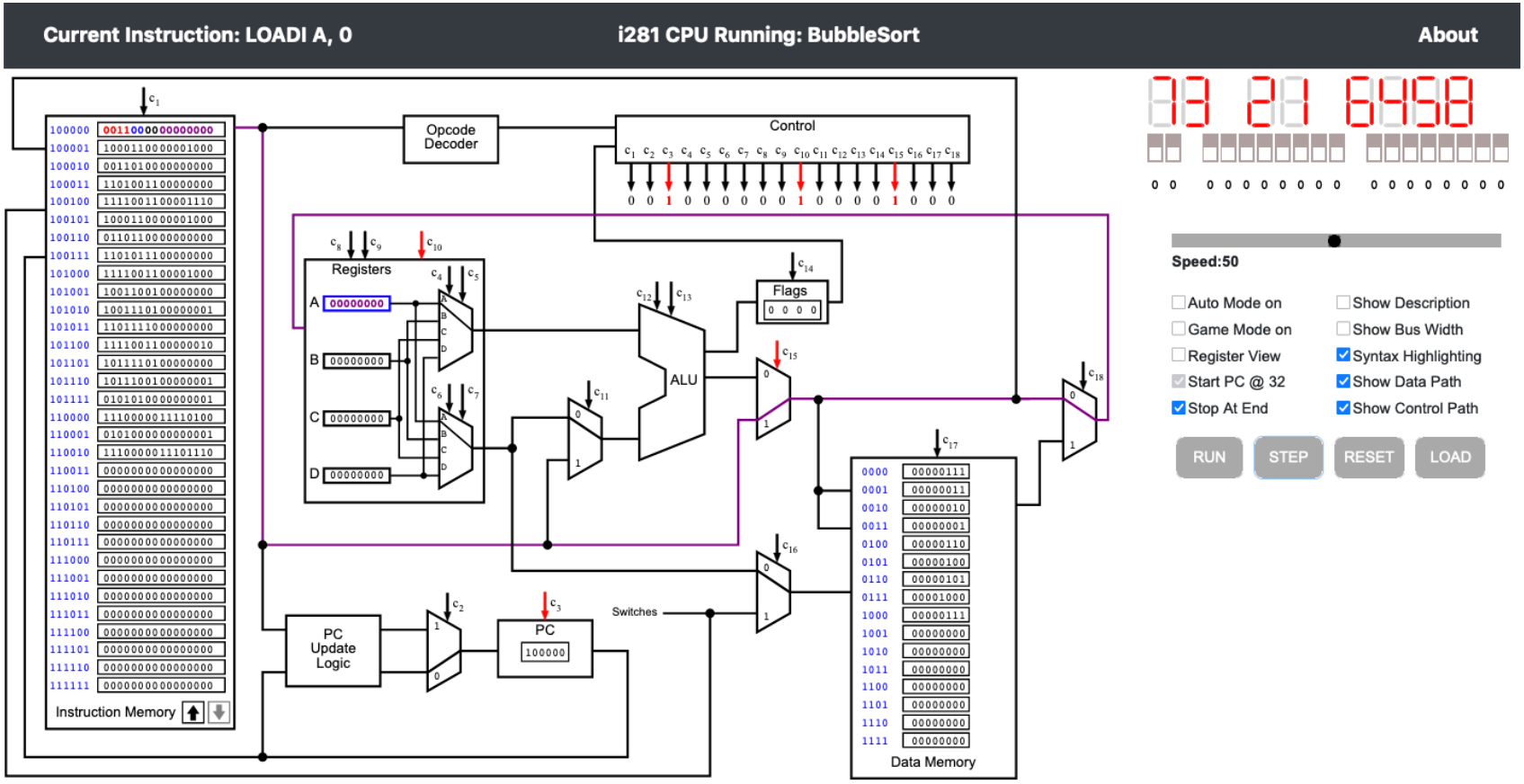

c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
6
16
6
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
8
c14
Flags
0 0 0 0
4
c12 c13
ALU
c15
c11
c16
c17
c18
6 low
4 low
8 low
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
Switches
16
c2
c3
PC Update Logic
PC
000000
6
i281 CPU


Write Select
c1 Write Enable
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Input
16
6
Read Select
Code Memory
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Write Select
c8 c9
Write Enable
c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
Port0 Read Select
c4 c5
Port1 Read Select
c6 c7
A B C D
8
Input
ALU Select
c12 c13
ALU
ALU Source
c11 Mux
Write Enable
c14
Flags
0 0 0 0
4
ALU Result
c15 Mux
REG Writeback
6 low Mux
c18
4 low
Read Select
Write Enable
c17
DMEM Input
Mux
c16
Write Select
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
6 low
PC Mux
c2
Write Enable
c3
PC Update Logic
PC
100000
Switches
16
8 low
Input
i281 CPU


c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
OpCode Decoder
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
c11
c12 c13
ALU
c14
Flags
0 0 0 0
c15
c16
c17
c18
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
Switches
c2
c3
PC Update Logic
PC
000000
i281 CPU

i281 Simulator

i281 Simulator



To try the simulator, go to the class web page and follow the link.

i281 Example:
Add the numbers from 1 to 5

i281 Example:
Add the numbers from 1 to 5

C Language v.s. Assembly Language

C Version

```
// C Version
//
// Add the numbers from 1 to 5 using a for loop.

int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++)
           sum+=i;

        // printf("%d\n", sum);
}
```

i281 Assembly Version

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0          ; sum=0
          LOADI   A, 1          ; i=1
          LOAD    D, [N]        ; register_D=N
Loop:     CMP     A, D          ; i<=N ?
          BRG     End           ; exit if i>N
Add:      ADD     B, A          ; sum+=i
          ADDI    A, 1          ; i++
          JUMP    Loop          ; next iteration
End:      STORE   [sum], B      ; update the memory for sum

; Register allocation:
; A: i
; B: sum
; C: <not used>
; D: N
```

i281 Assembly Version

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0          ; sum=0
          LOADI   A, 1          ; i=1
          LOAD    D, [N]        ; register_D=N
Loop:     CMP     A, D          ; i<=N ?
          BRG     End           ; exit if i>N
Add:      ADD     B, A          ; sum+=i
          ADDI    A, 1          ; i++
          JUMP    Loop          ; next iteration
End:      STORE   [sum], B      ; update the memory for sum

; Register allocation:
; A: i
; B: sum
; C: <not used>
; D: N
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0       ; sum=0
          LOADI   A, 1       ; i=1
          LOAD    D, [N]     ; register_D=N
Loop:     CMP     A, D       ; i<=N ?
          BRG     End        ; exit if i>N
Add:      ADD     B, A       ; sum+=i
          ADDI    A, 1       ; i++
          JUMP    Loop       ; next iteration
End:      STORE   [sum], B   ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

i=1

```
; Assembly Version

.data
N       BYTE    5
i       BYTE    ?
sum     BYTE    ?

.code
        LOADI   B, 0        ; sum=0
        LOADI   A, 1        ; i=1
        LOAD    D, [N]      ; register_D=N
Loop:   CMP     A, D        ; i<=N ?
        BRG     End         ; exit if i>N
Add:    ADD     B, A        ; sum+=i
        ADDI    A, 1        ; i++
        JUMP    Loop        ; next iteration
End:    STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

This has no analog in the C version, which is written in a high-level language.

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Load the value of N into register D.

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0       ; sum=0
          LOADI   A, 1       ; i=1
          LOAD    D, [N]     ; register_D=N
Loop:     CMP     A, D       ; i<=N ?
          BRG     End        ; exit if i>N
Add:      ADD     B, A       ; sum+=i
          ADDI    A, 1       ; i++
          JUMP    Loop       ; next iteration
End:      STORE   [sum], B   ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

i=2

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0       ; sum=0
          LOADI   A, 1       ; i=1
          LOAD    D, [N]     ; register_D=N
Loop:     CMP     A, D       ; i<=N ?
          BRG     End        ; exit if i>N
Add:      ADD     B, A       ; sum+=i
          ADDI    A, 1       ; i++
          JUMP    Loop       ; next iteration
End:      STORE   [sum], B   ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

i=3

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0       ; sum=0
          LOADI   A, 1       ; i=1
          LOAD    D, [N]     ; register_D=N
Loop:     CMP     A, D       ; i<=N ?
          BRG     End        ; exit if i>N
Add:      ADD     B, A       ; sum+=i
          ADDI    A, 1       ; i++
          JUMP    Loop       ; next iteration
End:      STORE   [sum], B   ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

i=4

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0       ; sum=0
          LOADI   A, 1       ; i=1
          LOAD    D, [N]     ; register_D=N
Loop:     CMP     A, D       ; i<=N ?
          BRG     End        ; exit if i>N
Add:      ADD     B, A       ; sum+=i
          ADDI    A, 1       ; i++
          JUMP    Loop       ; next iteration
End:      STORE   [sum], B   ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

i=5

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0       ; sum=0
          LOADI   A, 1       ; i=1
          LOAD    D, [N]     ; register_D=N
Loop:     CMP     A, D       ; i<=N ?
          BRG     End        ; exit if i>N
Add:      ADD     B, A       ; sum+=i
          ADDI    A, 1       ; i++
          JUMP    Loop       ; next iteration
End:      STORE   [sum], B   ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0       ; sum=0
          LOADI   A, 1       ; i=1
          LOAD    D, [N]     ; register_D=N
Loop:     CMP     A, D       ; i<=N ?
          BRG     End        ; exit if i>N
Add:      ADD     B, A       ; sum+=i
          ADDI    A, 1       ; i++
          JUMP    Loop       ; next iteration
End:      STORE   [sum], B   ; write B to sum
```

Add the numbers from 1 to 5

```
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

i=6

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0       ; sum=0
          LOADI   A, 1       ; i=1
          LOAD    D, [N]     ; register_D=N
Loop:     CMP     A, D       ; i<=N ?
          BRG     End        ; exit if i>N
Add:      ADD     B, A       ; sum+=i
          ADDI    A, 1       ; i++
          JUMP    Loop       ; next iteration
End:      STORE   [sum], B   ; write B to sum
```

Add the numbers from 1 to 5

```c
// C Version
// using a for loop




int main()
{
        int N=5;
        int i, sum;

        sum=0;
        for(i=1; i<=N; i++) {
           sum+=i;
        }

        // printf("%d\n", sum);
}
```

```
; Assembly Version

.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0        ; sum=0
          LOADI   A, 1        ; i=1
          LOAD    D, [N]      ; register_D=N
Loop:     CMP     A, D        ; i<=N ?
          BRG     End         ; exit if i>N
Add:      ADD     B, A        ; sum+=i
          ADDI    A, 1        ; i++
          JUMP    Loop        ; next iteration
End:      STORE   [sum], B    ; write B to sum
```

i281 Example:
Add the numbers from 1 to 5

Assembly Language v.s. Machine Language

i281 Assembly Code

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0          ; sum=0
          LOADI   A, 1          ; i=1
          LOAD    D, [N]        ; register_D=N
Loop:     CMP     A, D          ; i<=N ?
          BRG     End           ; exit if i>N
Add:      ADD     B, A          ; sum+=i
          ADDI    A, 1          ; i++
          JUMP    Loop          ; next iteration
End:      STORE   [sum], B      ; update the memory for sum
```

i281 Assembly Code

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

Mapping Assembly to Machine Code

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

Assembly Language

```
Data Memory:
00000101
00000000
00000000

Code Memory:
0011010000000000
0011000000000001
1000110000000000
1101001100000000
1111001000000011
0100010000000000
0101000000000001
1110000011111011
1010010000000010
```

Machine Language

Mapping Assembly to Machine Code

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

Assembly Language

```
Data Memory:
0000 0101
0000 0000
0000 0000

Code Memory:
0011 0100 0000 0000
0011 0000 0000 0001
1000 1100 0000 0000
1101 0011 0000 0000
1111 0010 0000 0011
0100 0100 0000 0000
0101 0000 0000 0001
1110 0000 1111 1011
1010 0100 0000 0010
```

Machine Language
in Binary

Mapping Assembly to Machine Code

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

Assembly Language

```
Data Memory:
 0    5
 0    0
 0    0

Code Memory:
 3    4    0    0
 3    0    0    1
 8    C    0    0
 D    3    0    0
 F    2    0    3
 4    4    0    0
 5    0    0    1
 E    0    F    B
 A    4    0    2
```

Machine Language
in Binary

Mapping Assembly to Machine Code

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

Assembly Language

```
Data Memory:
05
00
00

Code Memory:
34 00
30 01
8C 00
D3 00
F2 03
44 00
50 01
E0 FB
A4 02
```

Machine Language
in Hexadecimal

i281 Example:
Add the numbers from 1 to 5

Preview of OPCODEs

Mapping Assembly to Machine Code

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

Assembly Language

```
Data Memory:
00000101
00000000
00000000

Code Memory:
0011010000000000
0011000000000001
1000110000000000
1101001100000000
1111001000000011
0100010000000000
0101000000000001
1110000011111011
1010010000000010
```

Machine Language

Mapping Assembly to Machine Code

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

Assembly Language

```
Data Memory:
00000101
00000000
00000000

Code Memory:
00110100_00000000
00110000_00000001
10001100_00000000
11010011_00000000
11110010_00000011
01000100_00000000
01010000_00000001
11100000_11111011
10100100_00000010
```

Machine Language

Mapping Assembly to Machine Code

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

Assembly Language

```
Data Memory:
00000101
00000000
00000000

Code Memory:
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
```

Machine Language

Mapping Assembly to Machine Code

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

Data Memory:

```
00000101
00000000
00000000
```

Code Memory:

```
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
```

Mapping Assembly to Machine Code

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

```
Data Memory:
00000101
00000000
00000000

Code Memory:
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
```

Mapping Assembly to Machine Code

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

Data Memory:

```
00000101
00000000
00000000
```

Code Memory:

```
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
```

Mapping Assembly to Machine Code

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

Data Memory:

```
00000101
00000000
00000000
```

Code Memory:

```
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
```

OPCODE Mapping

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

Data Memory:

```
00000101
00000000
00000000
```

Code Memory:

```
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
```

OPCODE Mapping

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

Data Memory:

```
00000101
00000000
00000000
```

Code Memory:

```
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
```

Register Parameter Mapping

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

```
Data Memory:
00000101
00000000
00000000

Code Memory:
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
```

Register Parameter Mapping

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

```
Data Memory:
00000101
00000000
00000000

Code Memory:
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
```

Second Register Parameter Mapping

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

```
Data Memory:
00000101
00000000
00000000

Code Memory:
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
```

Second Register Parameter Mapping

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

```
Data Memory:
00000101
00000000
00000000

Code Memory:
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
```

Value / Address / Offset Mapping

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

```
Data Memory:
00000101
00000000
00000000

Code Memory:
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
```

Value / Address / Offset Mapping

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

```
Data Memory:
00000101
00000000
00000000

Code Memory:
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
```

“Don’t care” bits …

```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

```
Data Memory:
00000101
00000000
00000000

Code Memory:
0011_01_dd_00000000
0011_00_dd_00000001
1000_11_dd_00000000
1101_00_11_dddddddd
1111_dd_10_00000011
0100_01_00_dddddddd
0101_00_dd_00000001
1110_dd_dd_11111011
1010_01_dd_00000010
```

… are mapped to 0 by the Assembler

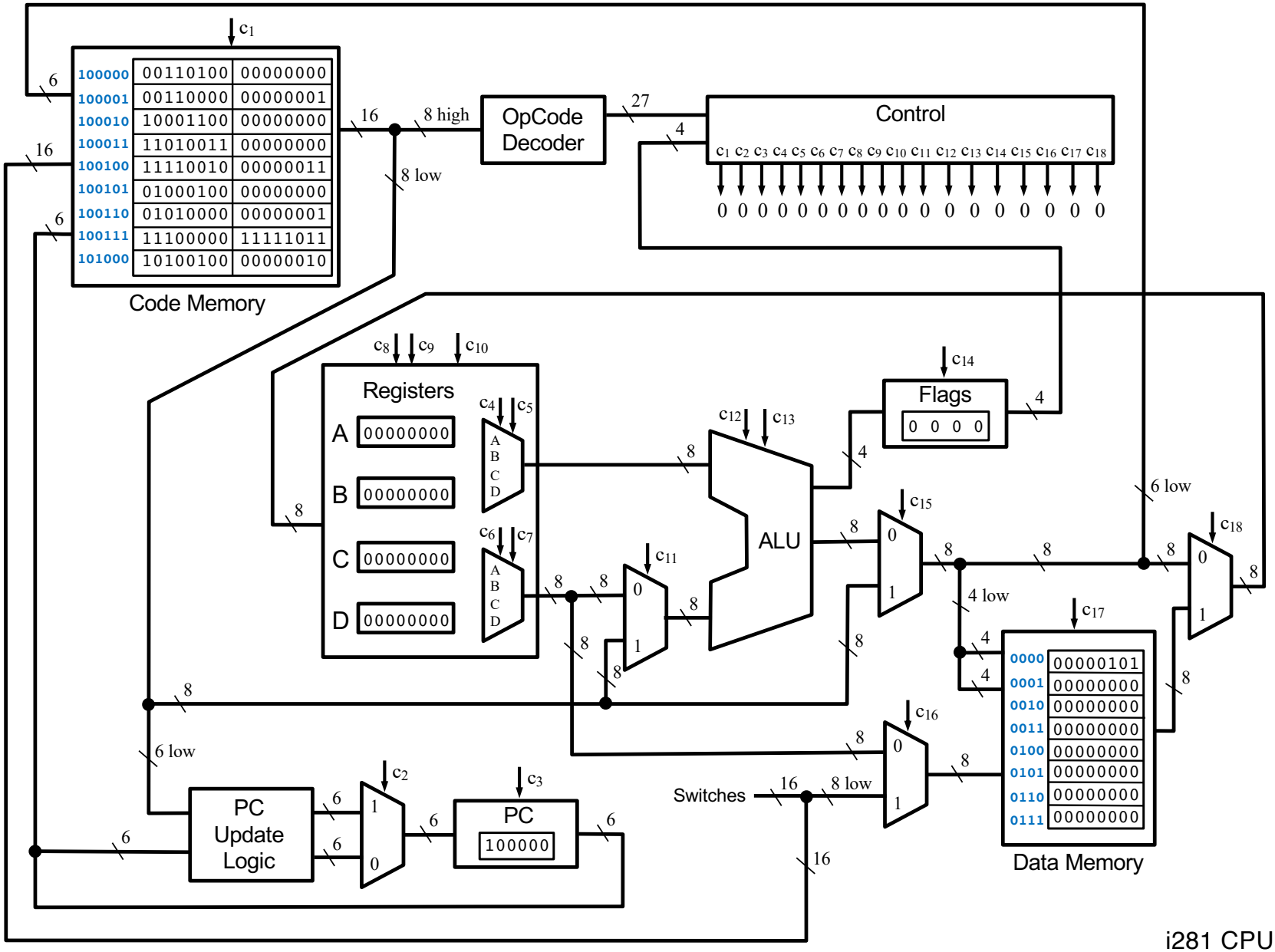
```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

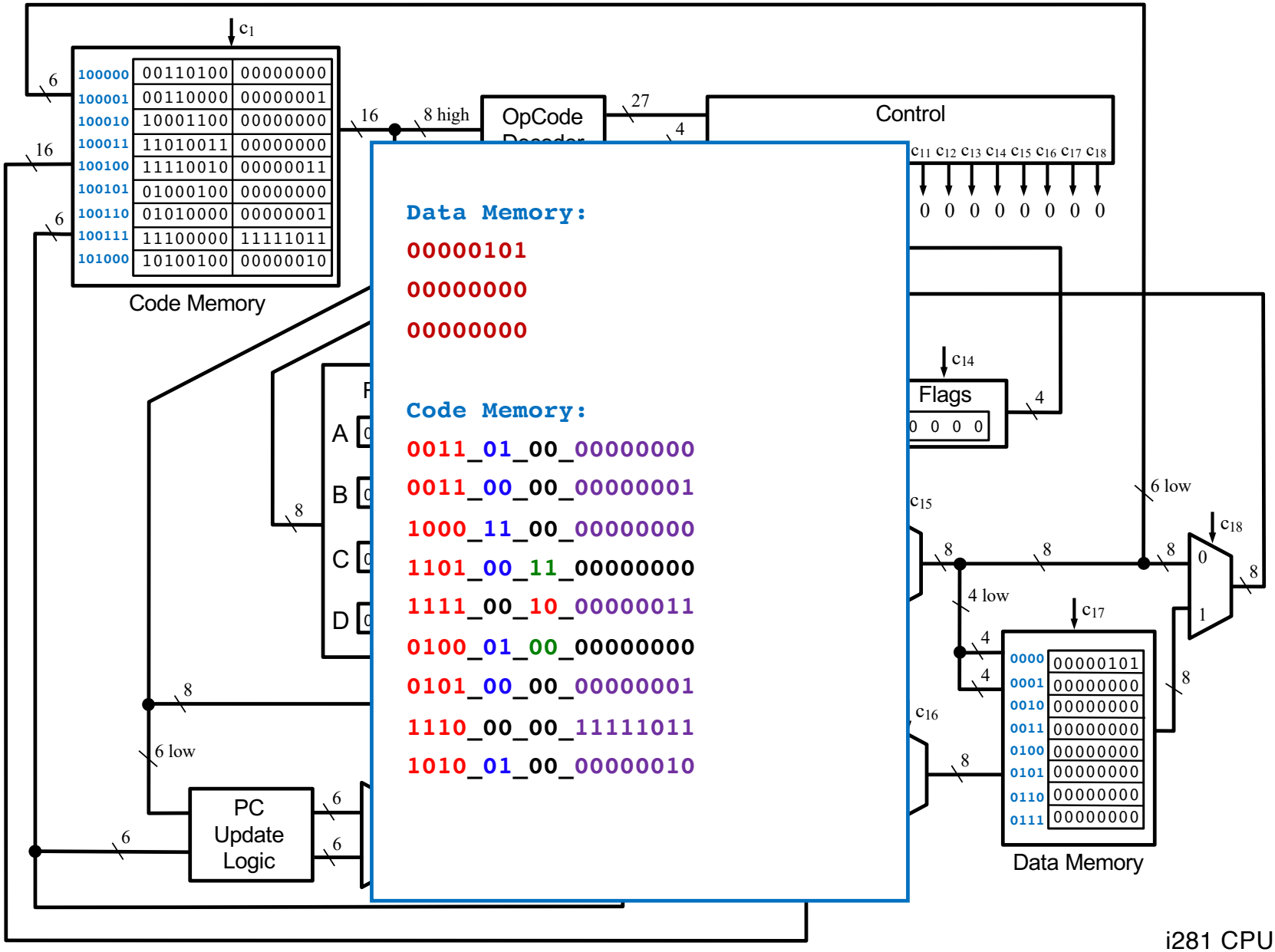
```
Data Memory:
00000101
00000000
00000000

Code Memory:
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
```

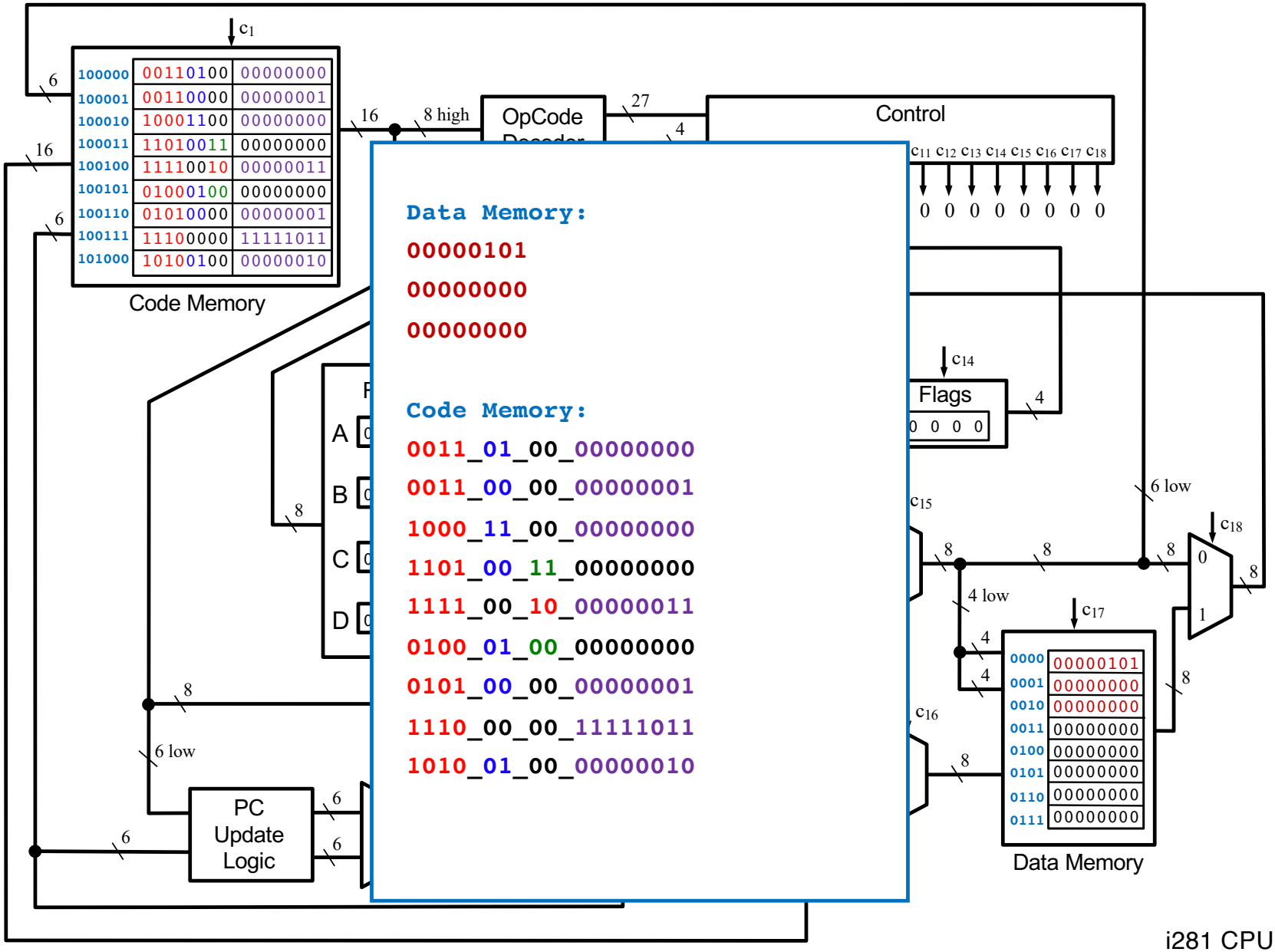
Mapping Assembly to Machine Code

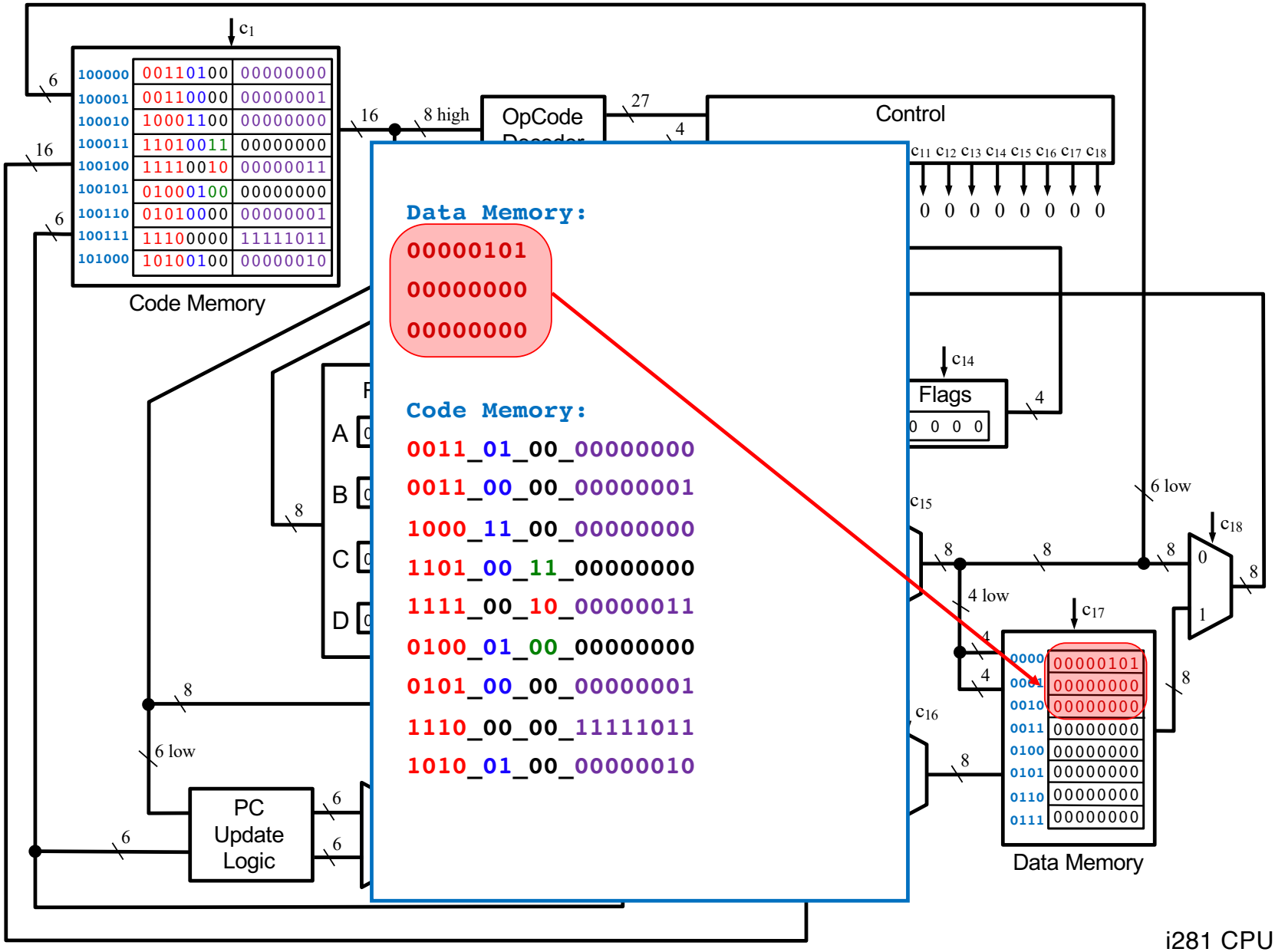
```
.data
N         BYTE    5
i         BYTE    ?
sum       BYTE    ?

.code
          LOADI   B, 0
          LOADI   A, 1
          LOAD    D, [N]
Loop:     CMP     A, D
          BRG     End
Add:      ADD     B, A
          ADDI    A, 1
          JUMP    Loop
End:      STORE   [sum], B
```

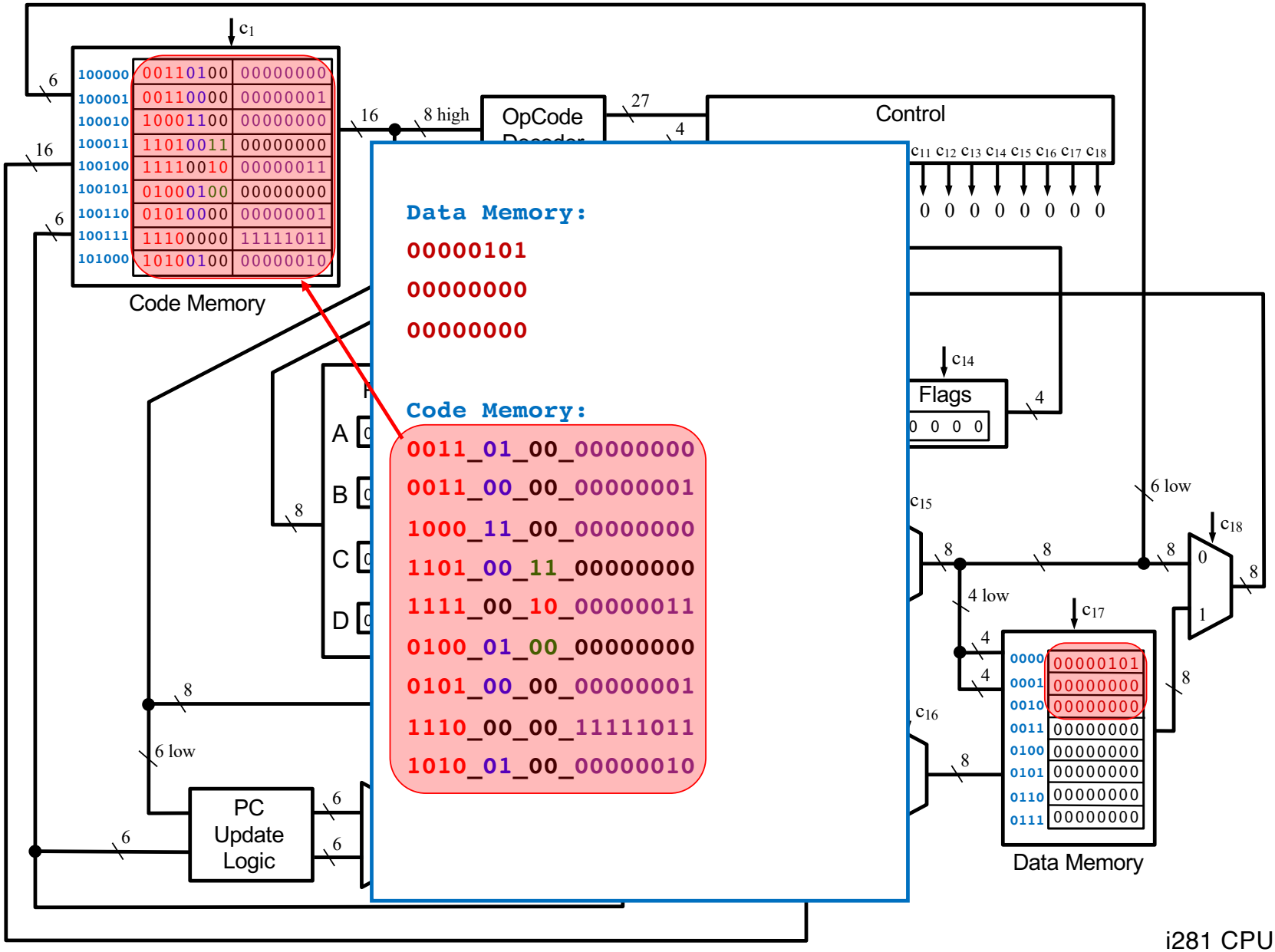
```
Data Memory:
00000101
00000000
00000000

Code Memory:
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
```

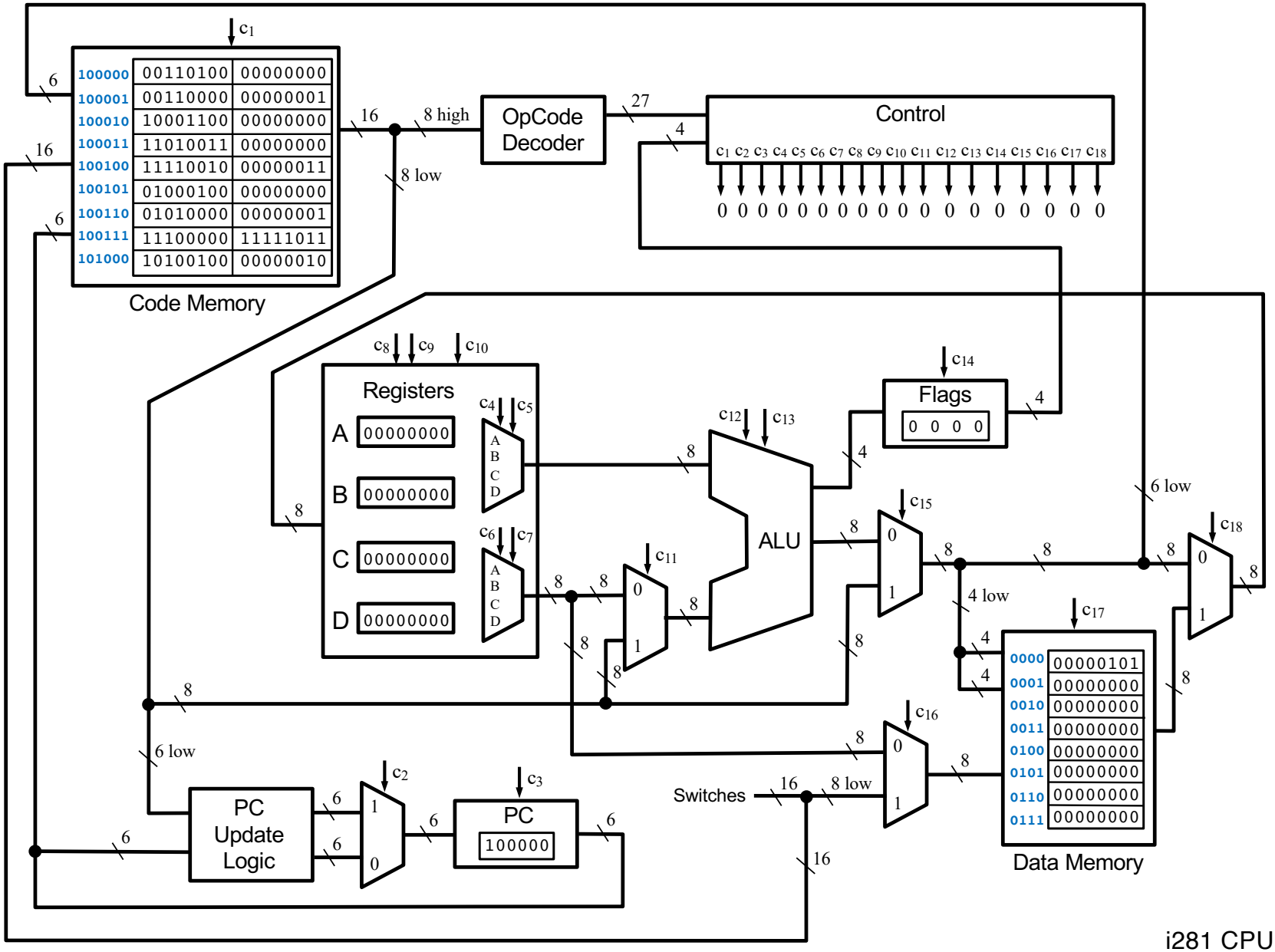

c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
6
16
6
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
8
c14
Flags
0 0 0 0
4
c12 c13
ALU
c15
c11
c16
c17
c18
6 low
4 low
8 low
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
Switches
16
c2
c3
PC Update Logic
PC
100000
i281 CPU


c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
6
16
8 high
OpCode
27
4
Control
c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0
Data Memory:
00000101
00000000
00000000
Code Memory:
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
c14
Flags
0 0 0 0
4
A
B
C
D
8
c15
6 low
c18
4 low
c17
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
c16
PC
Update
Logic
i281 CPU


c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
6
16
8 high
OpCode
27
4
Control
c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0
Data Memory:
00000101
00000000
00000000
Code Memory:
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
c14
Flags
0 0 0 0
4
A
B
C
D
8
c15
6 low
c18
4 low
c17
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
c16
PC
Update
Logic
i281 CPU


c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
6
16
16
8 high
OpCode
27
4
Control
c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0
Data Memory:
00000101
00000000
00000000
Code Memory:
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
c14
Flags
4
0 0 0 0
A
B
C
D
8
c15
8
8
8
6 low
c18
0
1
8
4 low
c17
4
4
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
c16
8
8
6 low
PC
Update
Logic
6
6
6
6
i281 CPU


c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
6
16
8 high
OpCode
27
4
Control
c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0
Data Memory:
00000101
00000000
00000000
Code Memory:
0011_01_00_00000000
0011_00_00_00000001
1000_11_00_00000000
1101_00_11_00000000
1111_00_10_00000011
0100_01_00_00000000
0101_00_00_00000001
1110_00_00_11111011
1010_01_00_00000010
A
B
C
D
8
c14
Flags
0 0 0 0
4
c15
6 low
c18
0
1
4 low
c17
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
c16
PC
Update
Logic
i281 CPU


c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
6
16
6
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
8
c14
Flags
0 0 0 0
4
c12 c13
ALU
c15
c11
c16
c17
c18
6 low
4 low
8 low
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
Switches
16
c2
c3
PC Update Logic
PC
100000
i281 CPU

The Assembly Language Instructions

The i281 Assembly Instructions

NOOP	NO OPeration
INPUTC	INPUT into Code memory
INPUTCF	INPUT into Code memory with oFfset
INPUTD	INPUT into Data memory
INPUTDF	INPUT into Data memory with oFfset
MOVE	MOVE the contents of one register into another
LOADI	LOAD Immediate value
LOADP	LOAD Pointer address
ADD	ADD two registers
ADDI	ADD an Immediate value to a register
SUB	SUBtract two registers
SUBI	SUBtract an Immediate value from a register
LOAD	LOAD from a data memory address into a register
LOADF	LOAD with an oFfset specified by another register
STORE	STORE a register into a data memory address
STOREF	STORE with an oFfset specified by another register
SHIFTL	SHIFT Left all bits in a register
SHIFTR	SHIFT Right all bits in a register
CMP	CoMPare the values in two registers
JUMP	JUMP unconditionally to a specified address
BRE	BRanch if Equal
BRZ	BRanch if Zero
BRNE	BRanch if Not Equal
BRNZ	BRanch if Not Zero
BRG	BRanch if Greater
BRGE	BRanch if Greater than or Equal

The i281 Assembly Instructions

NOOP	NO OPeration
INPUTC	INPUT into Code memory
INPUTCF	INPUT into Code memory with oFfset
INPUTD	INPUT into Data memory
INPUTDF	INPUT into Data memory with oFfset
MOVE	MOVE the contents of one register into another
LOADI	LOAD Immediate value
LOADP	LOAD Pointer address
ADD	ADD two registers
ADDI	ADD an Immediate value to a register
SUB	SUBtract two registers
SUBI	SUBtract an Immediate value from a register
LOAD	LOAD from a data memory address into a register
LOADF	LOAD with an oFfset specified by another register
STORE	STORE a register into a data memory address
STOREF	STORE with an oFfset specified by another register
SHIFTL	SHIFT Left all bits in a register
SHIFTR	SHIFT Right all bits in a register
CMP	CoMPare the values in two registers
JUMP	JUMP unconditionally to a specified address
BRE	BRanch if Equal
BRZ	BRanch if Zero
BRNE	BRanch if Not Equal
BRNZ	BRanch if Not Zero
BRG	BRanch if Greater
BRGE	BRanch if Greater than or Equal

There are only 26 Instructions

```
NOOP
INPUTC
INPUTCF
INPUTD
INPUTDF
MOVE
LOADI
LOADP
```

```
ADD
ADDI
SUB
SUBI
LOAD
LOADF
STORE
STOREF
```

```
SHIFTL
SHIFTR
CMP
JUMP
BRE
BRZ
BRNE
BRNZ
BRG
BRGE
```

There are only 26 Instructions

```
NOOP
INPUTC
INPUTCF
INPUTD
INPUTDF
MOVE
LOADI
LOADP
```

```
ADD
ADDI
SUB
SUBI
LOAD
LOADF
STORE
STOREF
```

```
SHIFTL
SHIFTR
CMP
JUMP
BRE
BRZ
BRNE
BRNZ
BRG
BRGE
```

All of these are available in the assembly language for this processor.
However, three pairs are aliased at the machine language level.

There are only ~~26~~ 25 Instructions

```
NOOP
INPUTC
INPUTCF
INPUTD
INPUTDF
MOVE
LOADI
LOADP
```

these two are aliased

```
ADD
ADDI
SUB
SUBI
LOAD
LOADF
STORE
STOREF
```

```
SHIFTL
SHIFTR
CMP
JUMP
BRE
BRZ
BRNE
BRNZ
BRG
BRGE
```

They have a different meaning in the assembly language, but the assembler maps them to the same machine language OPCODE.

There are only ~~26~~ 25 Instructions

```
NOOP
INPUTC
INPUTCF
INPUTD
INPUTDF
MOVE
LOADI/LOADP
```

```
ADD
ADDI
SUB
SUBI
LOAD
LOADF
STORE
STOREF
```

```
SHIFTL
SHIFTR
CMP
JUMP
BRE
BRZ
BRNE
BRNZ
BRG
BRGE
```

There are only ~~26~~ 24 Instructions

```
NOOP
INPUTC
INPUTCF
INPUTD
INPUTDF
MOVE
LOADI/LOADP
```

```
ADD
ADDI
SUB
SUBI
LOAD
LOADF
STORE
STOREF
```

```
SHIFTL
SHIFTR
CMP
JUMP
BRE
BRZ
BRNE
BRNZ
BRG
BRGE
```

these two are aliased

There are only ~~26~~ 23 Instructions

```
NOOP
INPUTC
INPUTCF
INPUTD
INPUTDF
MOVE
LOADI/LOADP
```

```
ADD
ADDI
SUB
SUBI
LOAD
LOADF
STORE
STOREF
```

```
SHIFTL
SHIFTR
CMP
JUMP
BRE
BRZ
BRNE
BRNZ
BRG
BRGE
```

these two are aliased

these two are aliased

There are only 23 Instructions

```
NOOP
INPUTC
INPUTCF
INPUTD
INPUTDF
MOVE
LOADI/LOADP
```

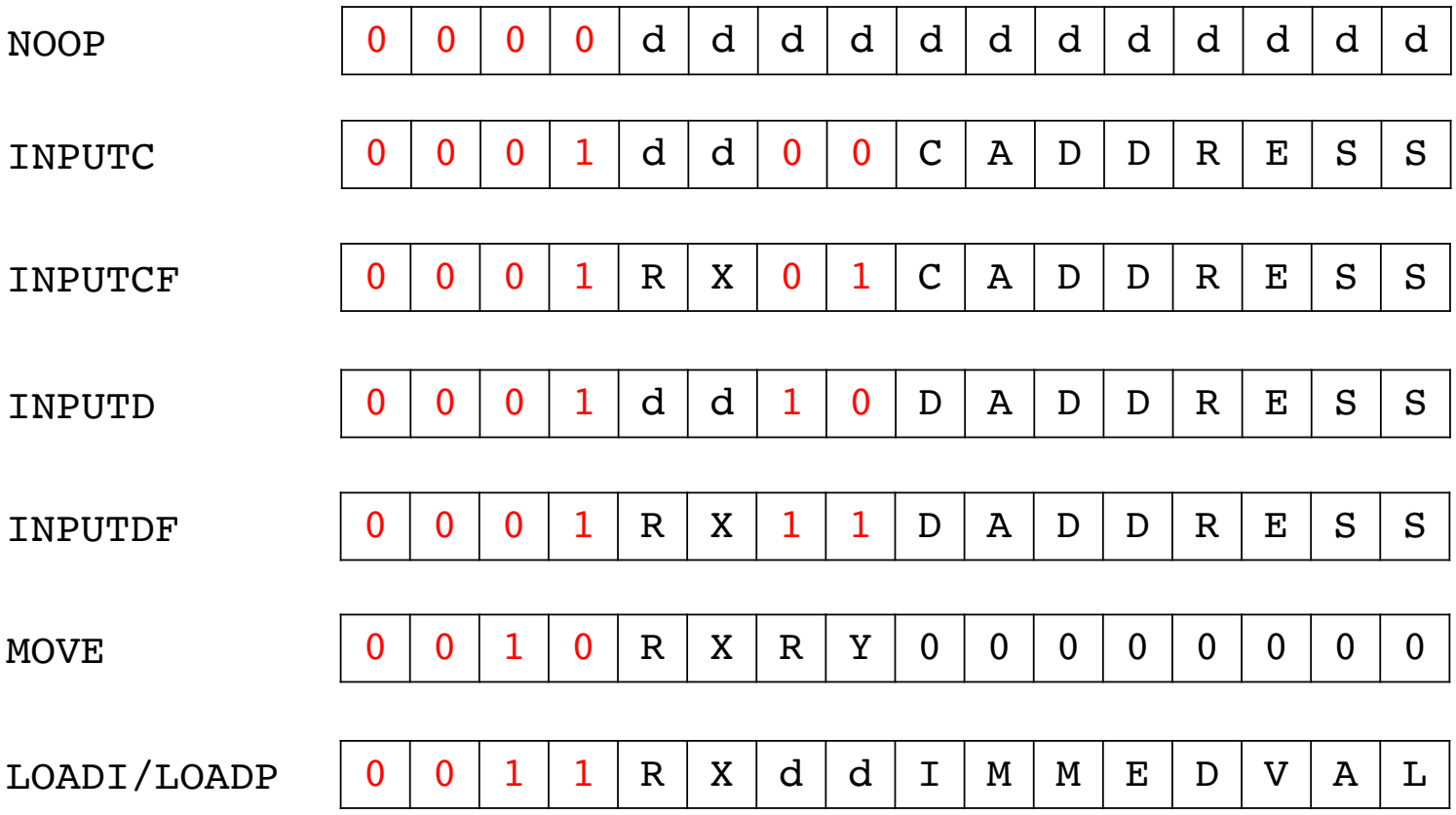
```
ADD
ADDI
SUB
SUBI
LOAD
LOADF
STORE
STOREF
```

```
SHIFTL
SHIFTR
CMP
JUMP
BRE/BRZ
BRNE/BRNZ
BRG
BRGE
```

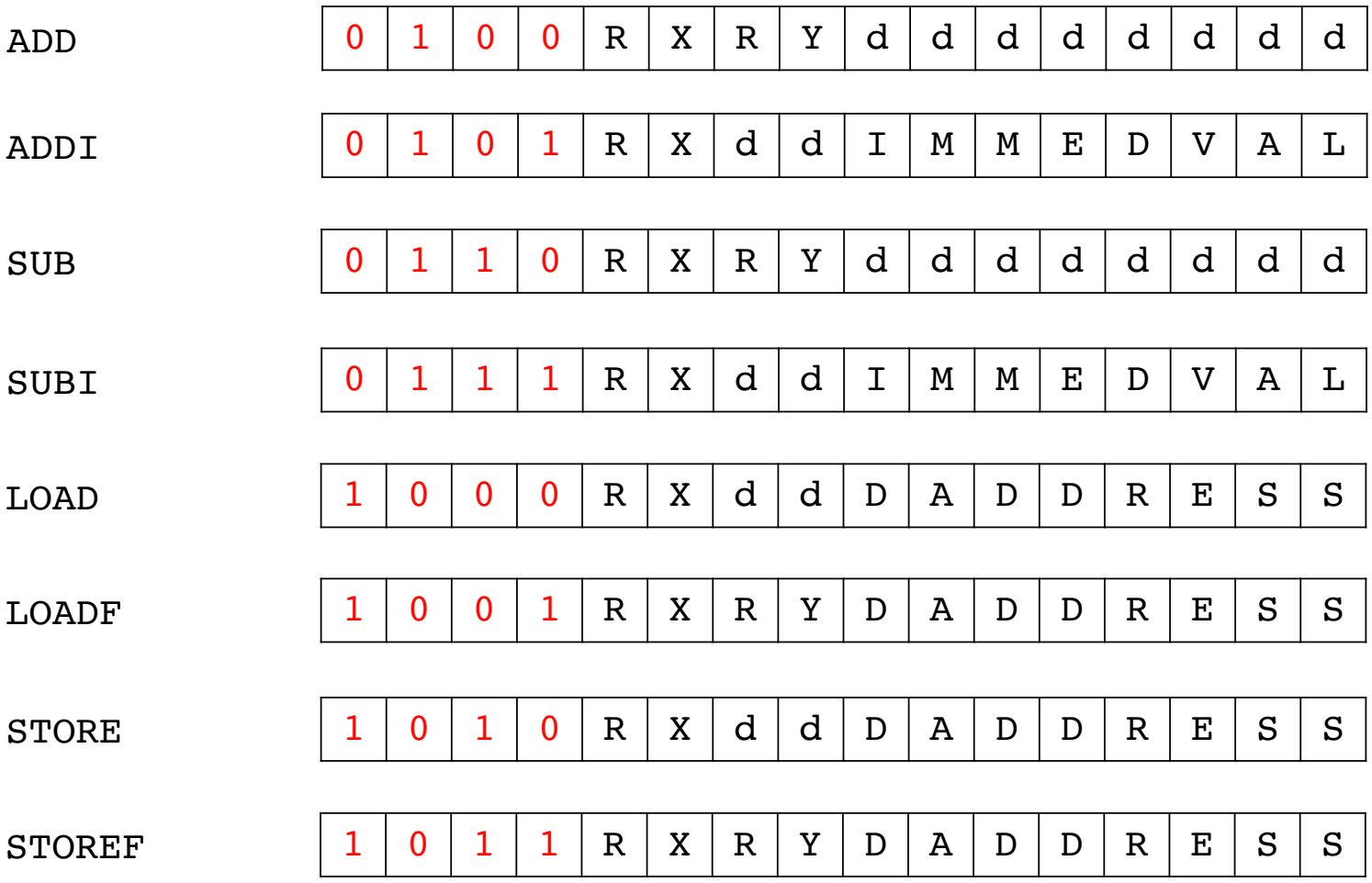
The OPCODEs

(Mapped to Machine Language)

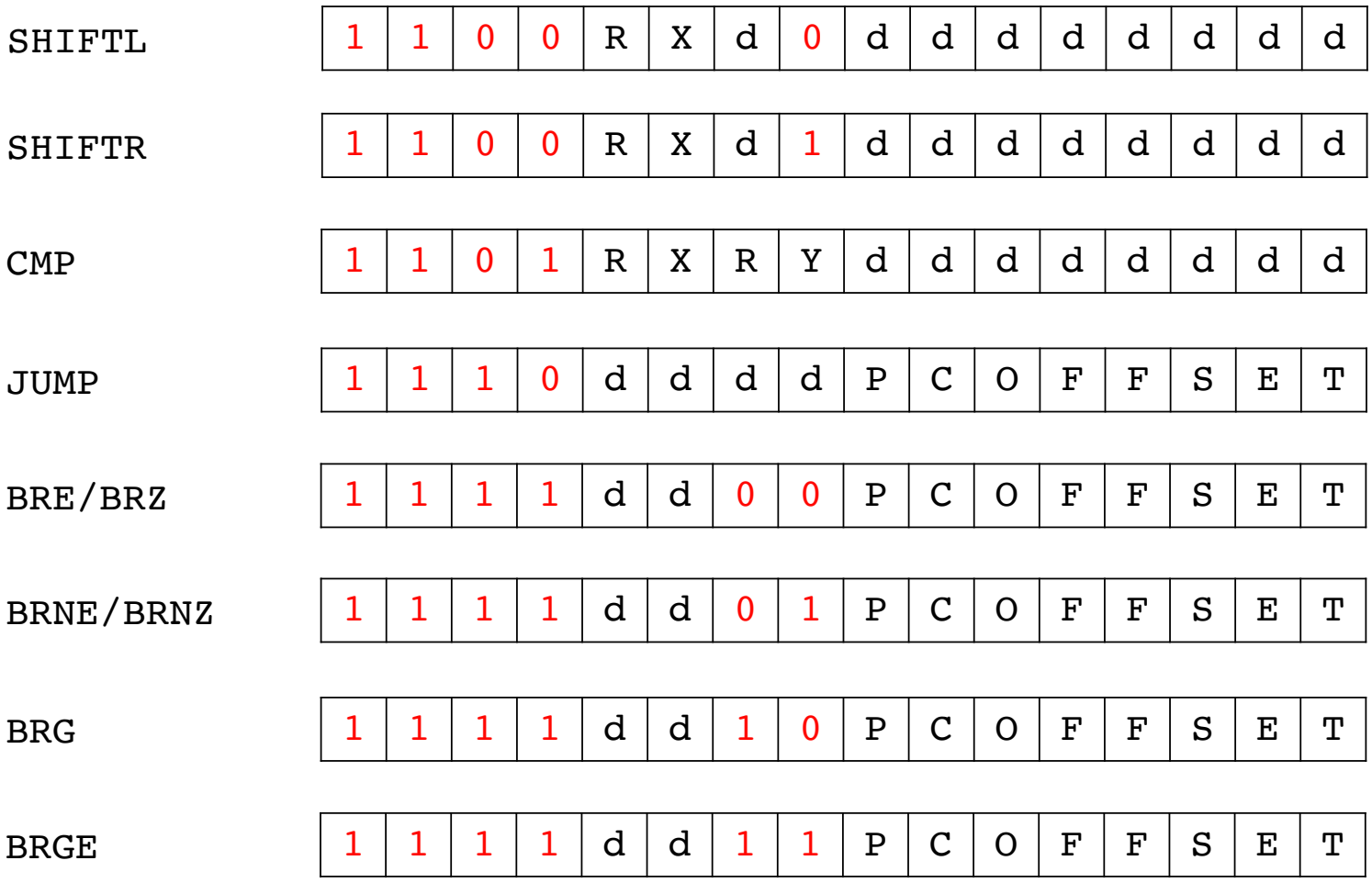
The OPCODEs


NOOP
0 0 0 0 d d d d d d d d d d d d
INPUTC
0 0 0 1 d d 0 0 C A D D R E S S
INPUTCF
0 0 0 1 R X 0 1 C A D D R E S S
INPUTD
0 0 0 1 d d 1 0 D A D D R E S S
INPUTDF
0 0 0 1 R X 1 1 D A D D R E S S
MOVE
0 0 1 0 R X R Y 0 0 0 0 0 0 0 0
LOADI/LOADP
0 0 1 1 R X d d I M M E D V A L

The OPCODEs

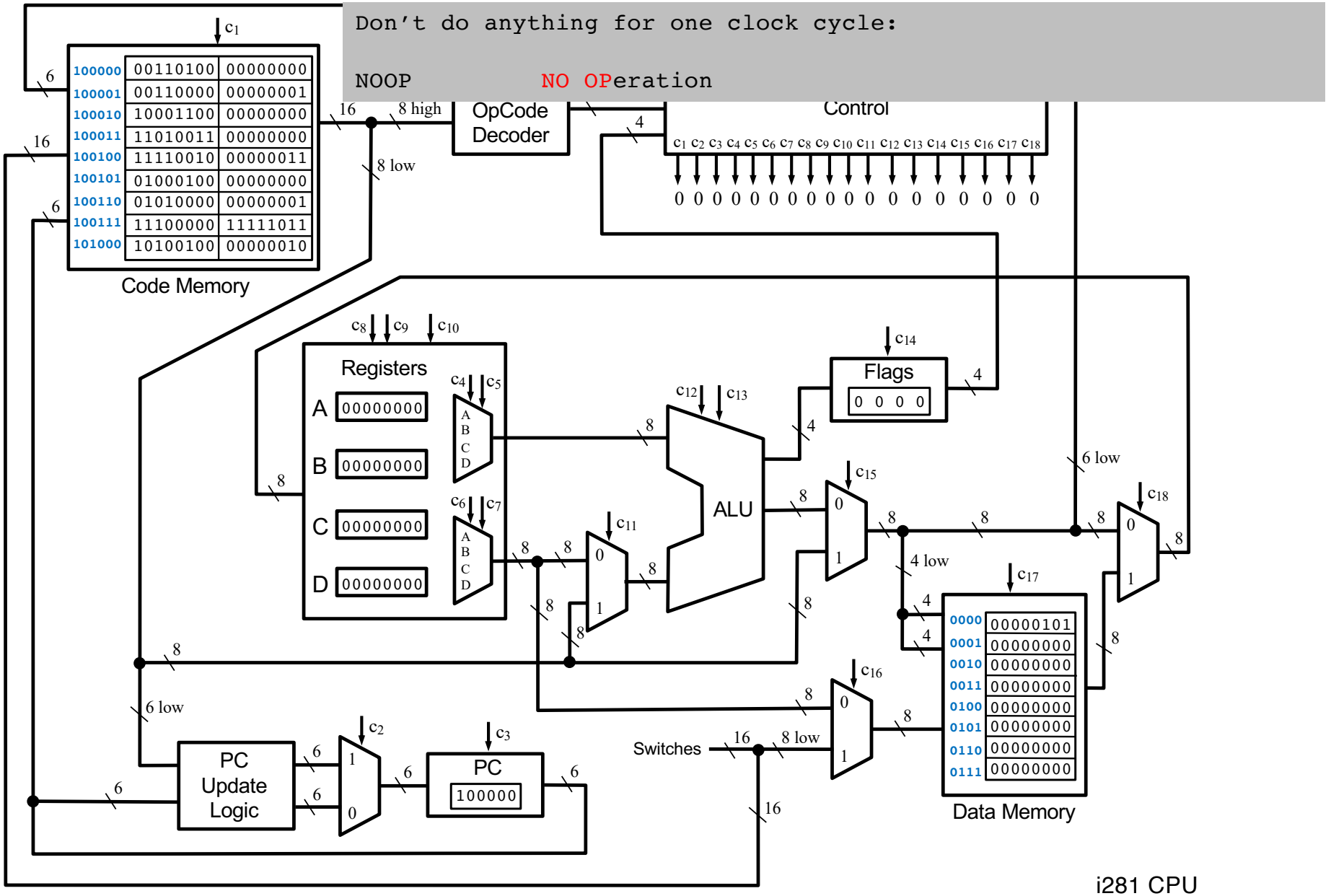

ADD
0 1 0 0 R X R Y d d d d d d d d
ADDI
0 1 0 1 R X d d I M M E D V A L
SUB
0 1 1 0 R X R Y d d d d d d d d
SUBI
0 1 1 1 R X d d I M M E D V A L
LOAD
1 0 0 0 R X d d D A D D R E S S
LOADF
1 0 0 1 R X R Y D A D D R E S S
STORE
1 0 1 0 R X d d D A D D R E S S
STOREF
1 0 1 1 R X R Y D A D D R E S S

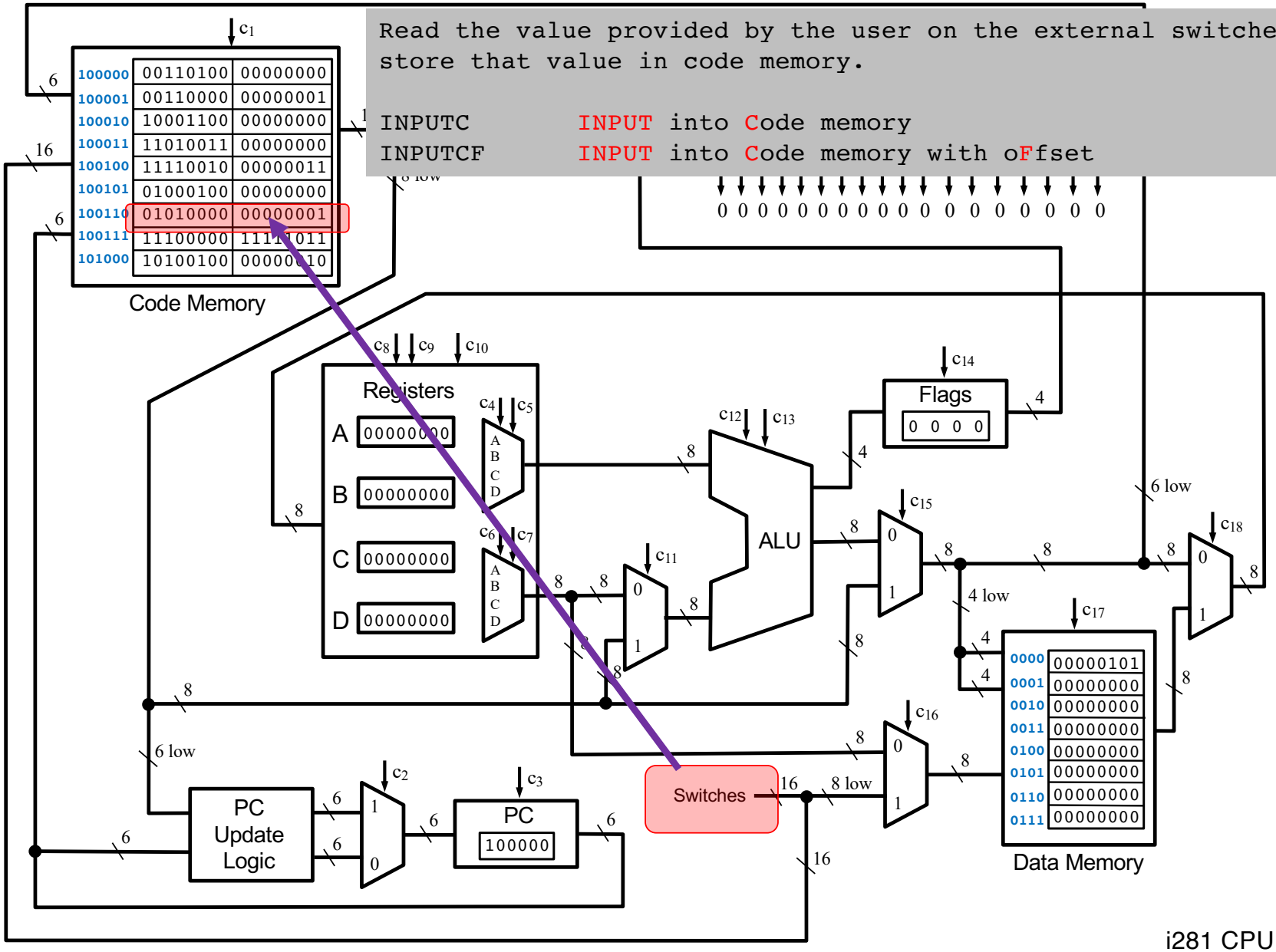
The OPCODEs

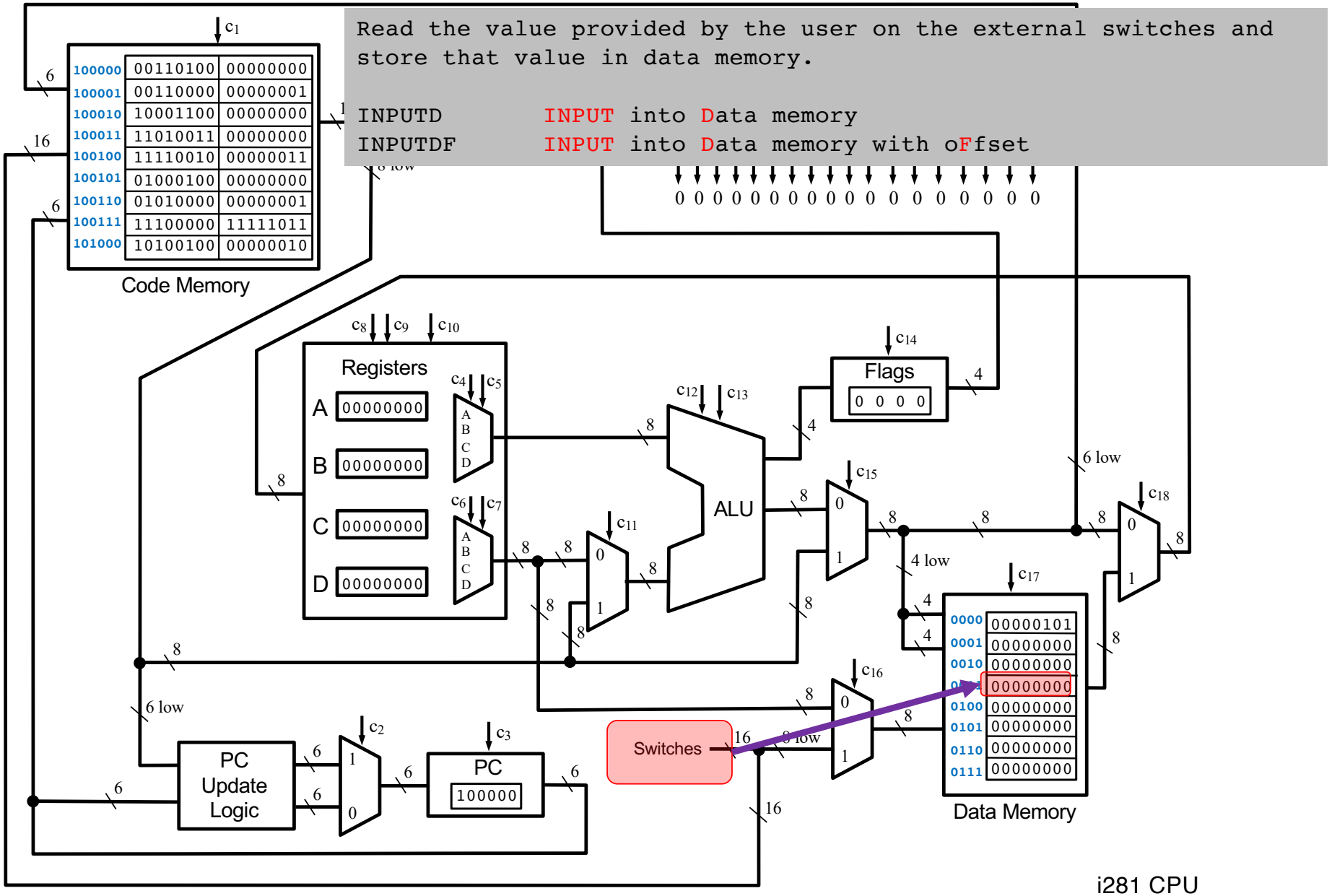

SHIFTL
1 1 0 0 R X d 0 d d d d d d d d
SHIFTR
1 1 0 0 R X d 1 d d d d d d d d
CMP
1 1 0 1 R X R Y d d d d d d d d
JUMP
1 1 1 0 d d d d P C O F F S E T
BRE/BRZ
1 1 1 1 d d 0 0 P C O F F S E T
BRNE/BRNZ
1 1 1 1 d d 0 1 P C O F F S E T
BRG
1 1 1 1 d d 1 0 P C O F F S E T
BRGE
1 1 1 1 d d 1 1 P C O F F S E T

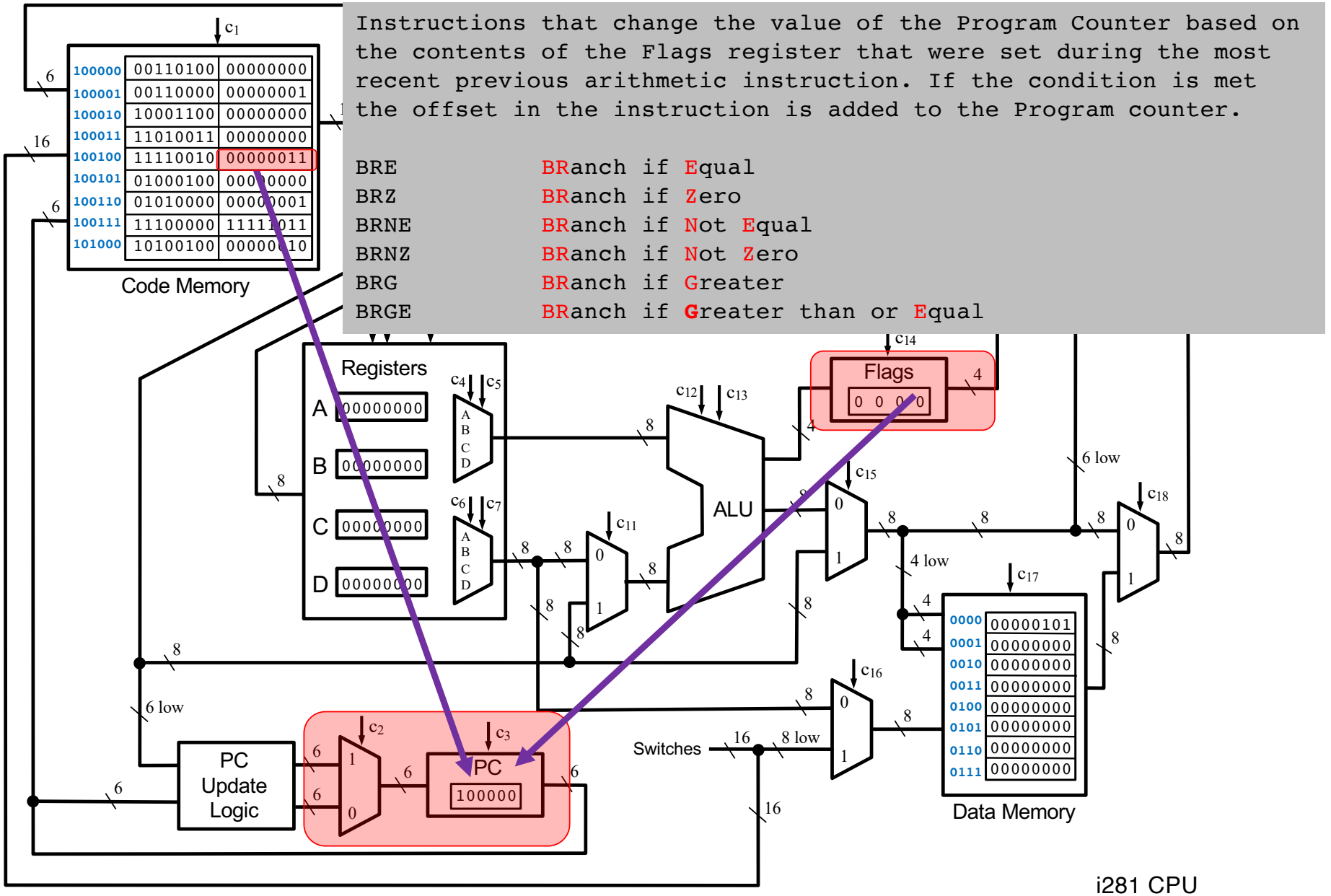
The i281 Assembly Instructions

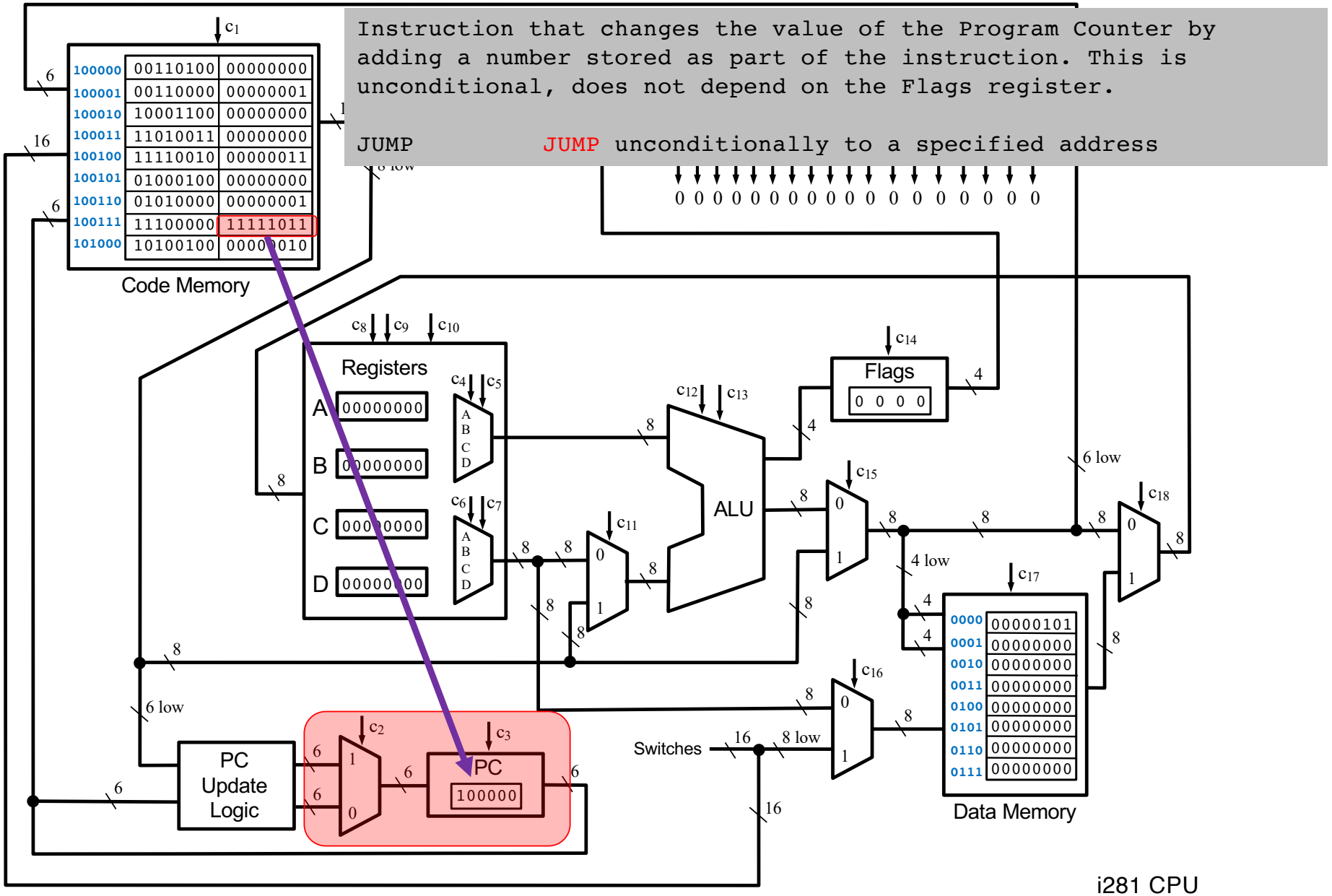
NOOP	NO OPeration
INPUTC	INPUT into Code memory
INPUTCF	INPUT into Code memory with oFfset
INPUTD	INPUT into Data memory
INPUTDF	INPUT into Data memory with oFfset
MOVE	MOVE the contents of one register into another
LOADI	LOAD Immediate value
LOADP	LOAD Pointer address
ADD	ADD two registers
ADDI	ADD an Immediate value to a register
SUB	SUBtract two registers
SUBI	SUBtract an Immediate value from a register
LOAD	LOAD from a data memory address into a register
LOADF	LOAD with an oFfset specified by another register
STORE	STORE a register into a data memory address
STOREF	STORE with an oFfset specified by another register
SHIFTL	SHIFT Left all bits in a register
SHIFTR	SHIFT Right all bits in a register
CMP	CoMPare the values in two registers
JUMP	JUMP unconditionally to a specified address
BRE	BRanch if Equal
BRZ	BRanch if Zero
BRNE	BRanch if Not Equal
BRNZ	BRanch if Not Zero
BRG	BRanch if Greater
BRGE	BRanch if Greater than or Equal

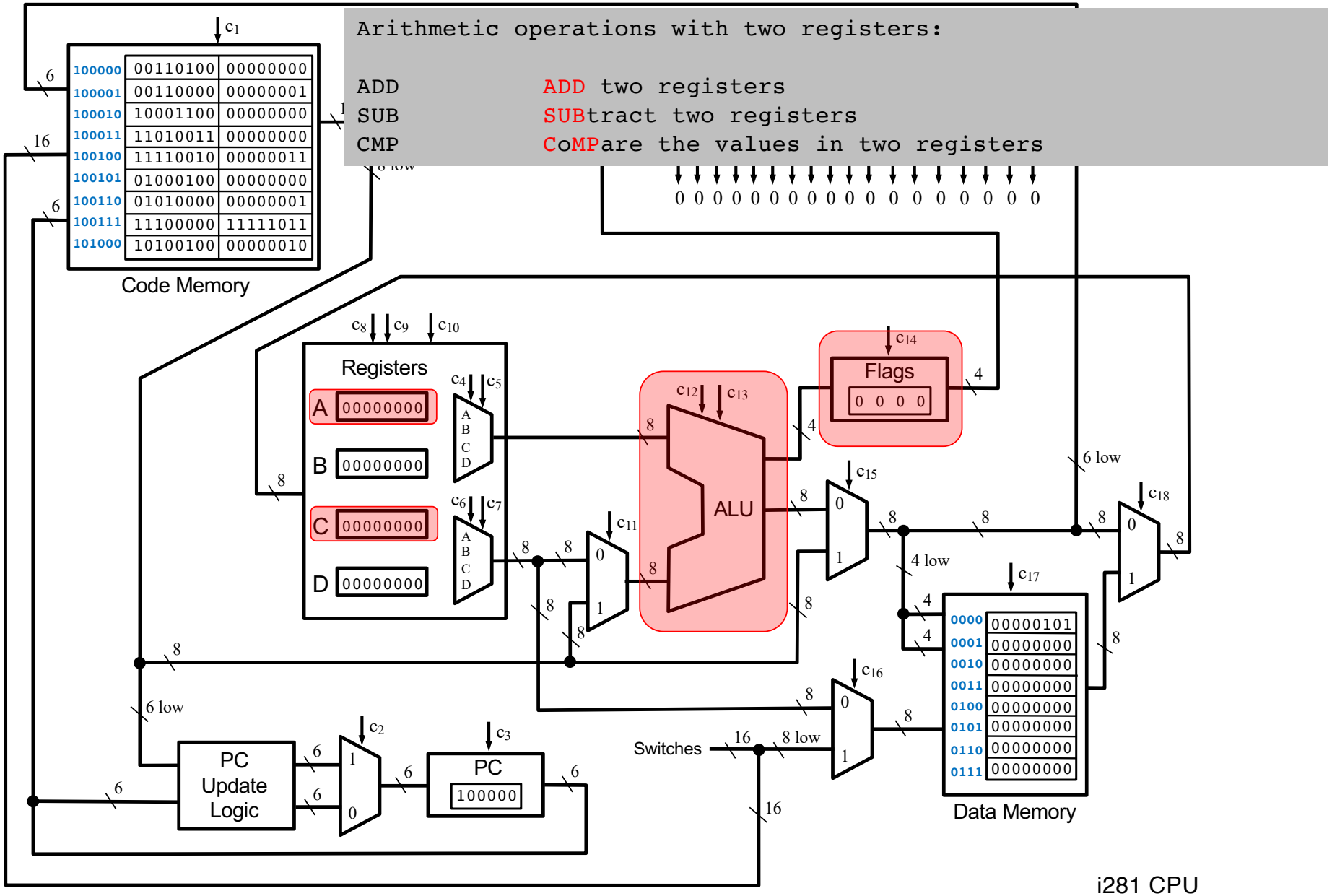

Don't do anything for one clock cycle:
NOOP NO OPeration
c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
16
8 high
8 low
OpCode Decoder
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
c12 c13
ALU
c14
Flags
0 0 0 0
c15
c11
c16
c17
c18
6 low
4 low
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
Switches
8 low
PC Update Logic
c2
c3
PC
100000
i281 CPU

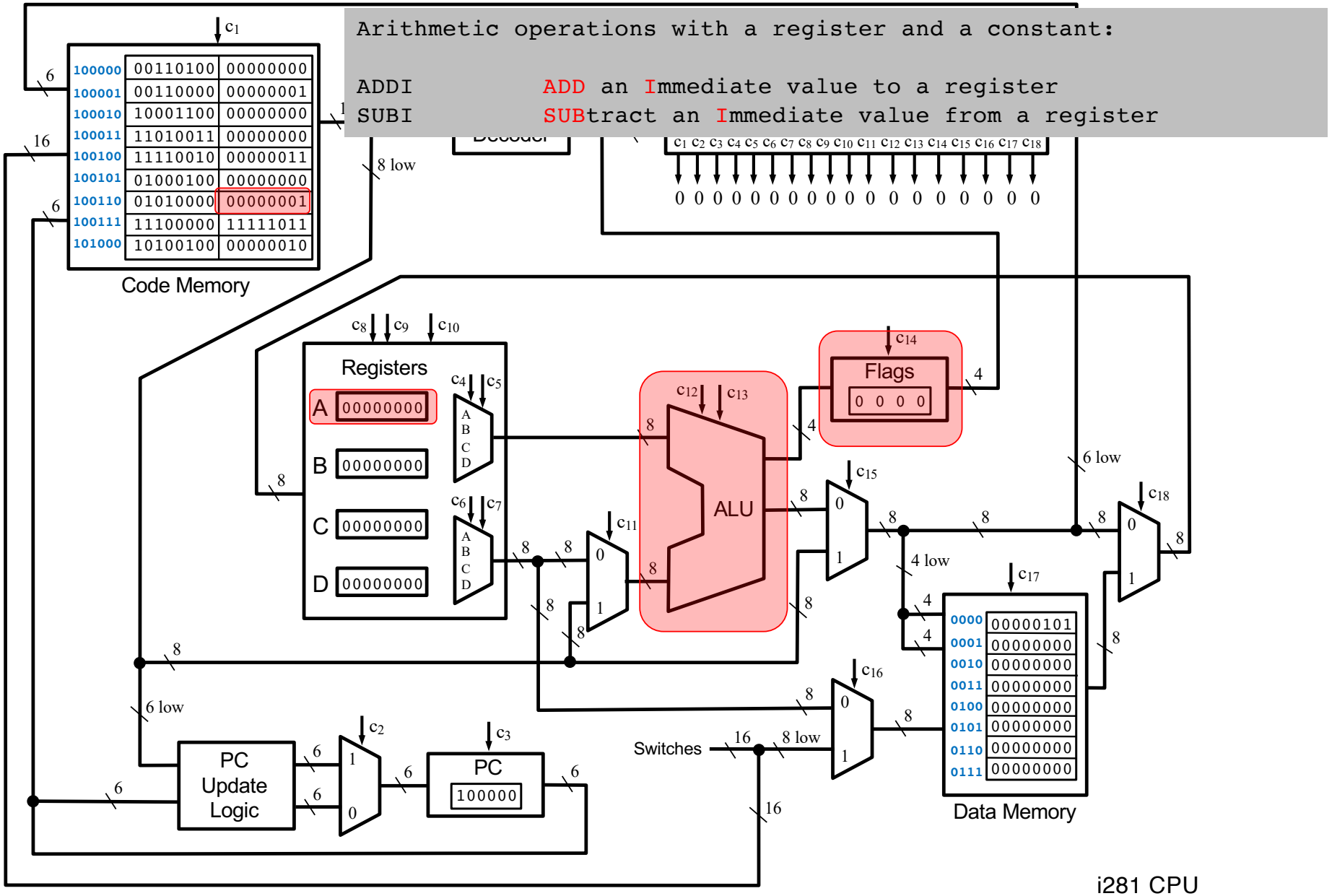

Read the value provided by the user on the external switches and store that value in code memory.
INPUTC INPUT into Code memory
INPUTCF INPUT into Code memory with oFfset
Code Memory
Registers
A
B
C
D
ALU
Flags
PC
PC Update Logic
Switches
Data Memory
i281 CPU

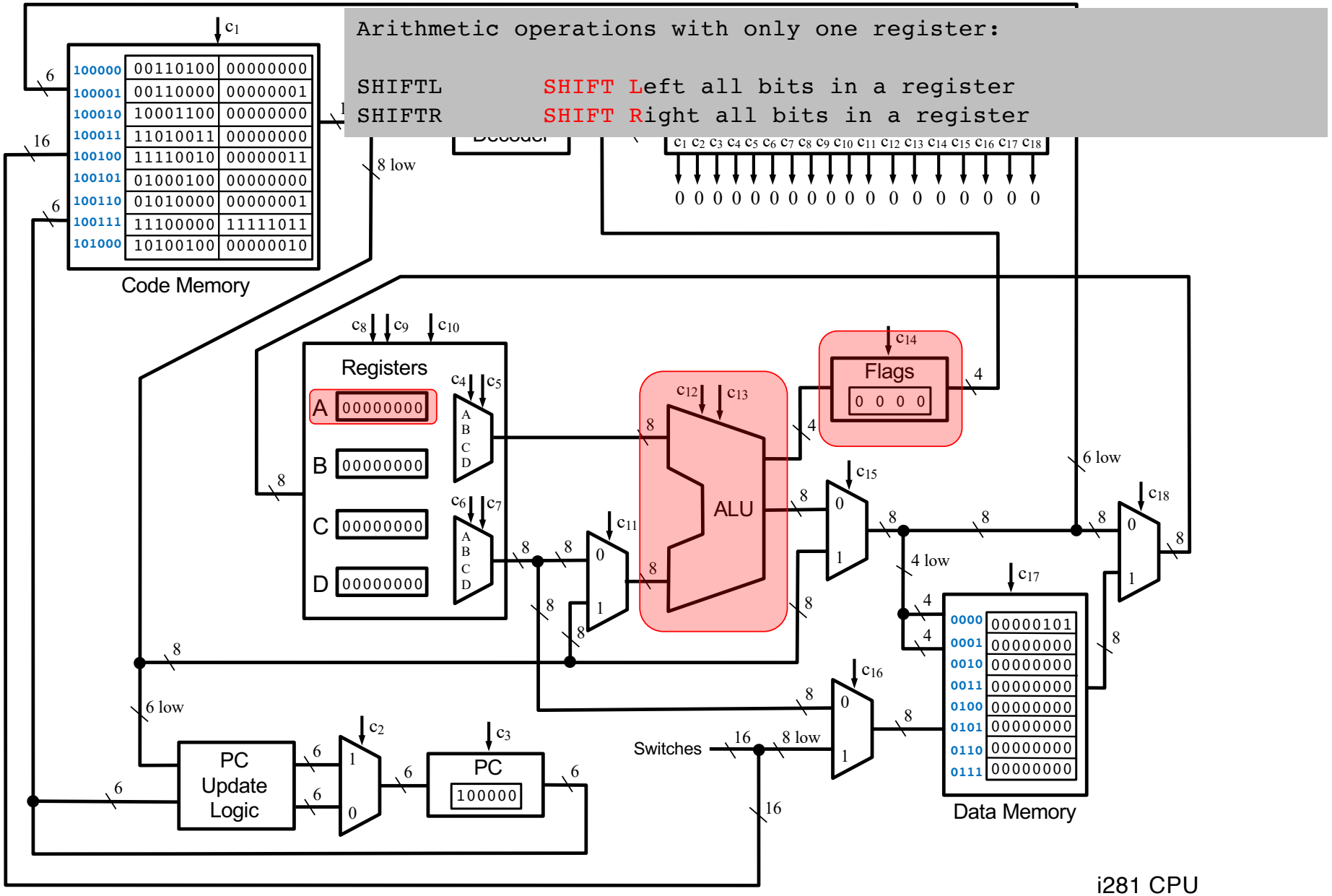

Read the value provided by the user on the external switches and store that value in data memory.
INPUTD INPUT into Data memory
INPUTDF INPUT into Data memory with oFfset
Code Memory
Registers
A
B
C
D
ALU
Flags
PC
Update
Logic
PC
Switches
Data Memory
i281 CPU

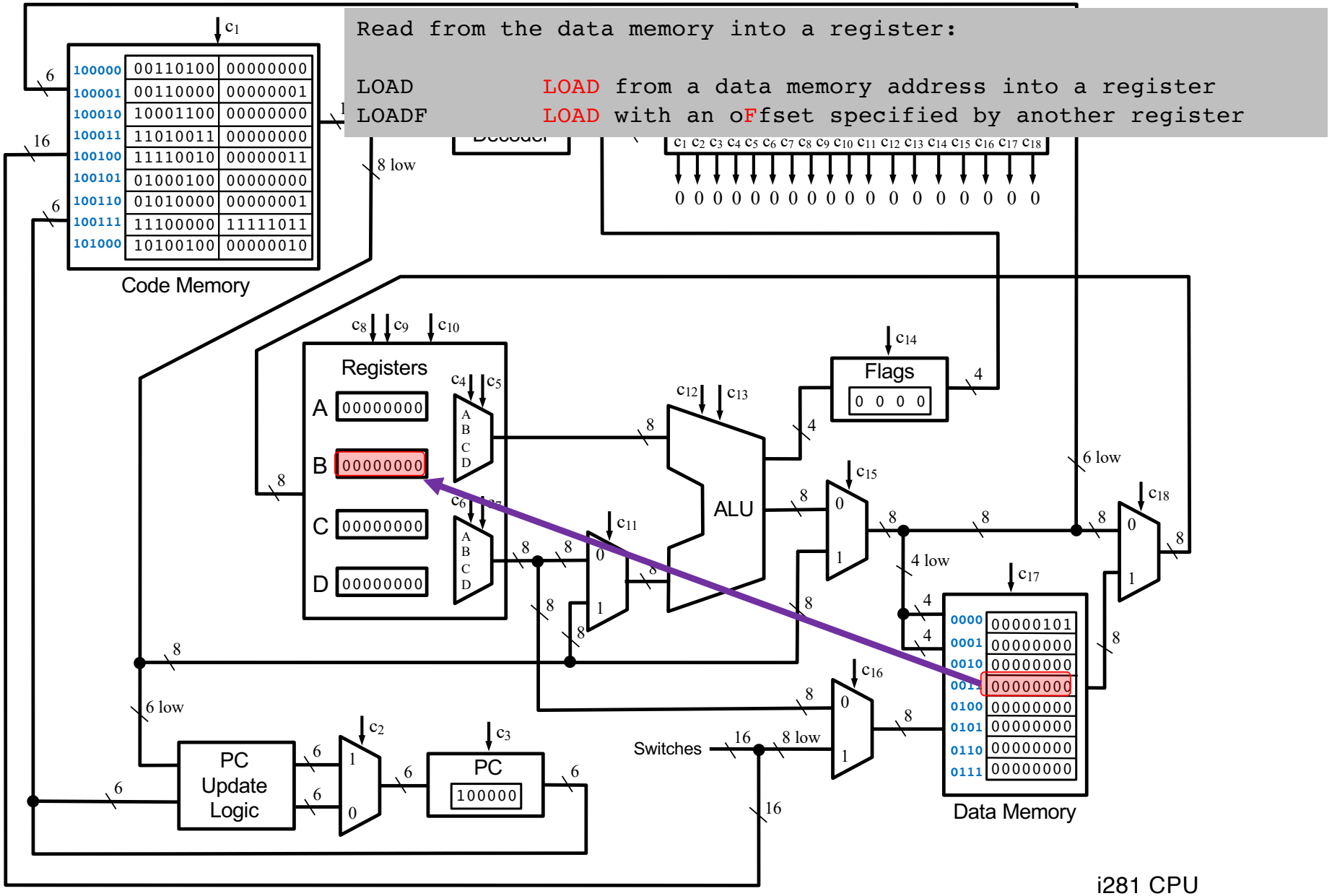

Instructions that change the value of the Program Counter based on the contents of the Flags register that were set during the most recent previous arithmetic instruction. If the condition is met the offset in the instruction is added to the Program counter.
BRE BRanch if Equal
BRZ BRanch if Zero
BRNE BRanch if Not Equal
BRNZ BRanch if Not Zero
BRG BRanch if Greater
BRGE BRanch if Greater than or Equal
Code Memory
Registers
Flags
ALU
PC
PC Update Logic
Switches
Data Memory
i281 CPU

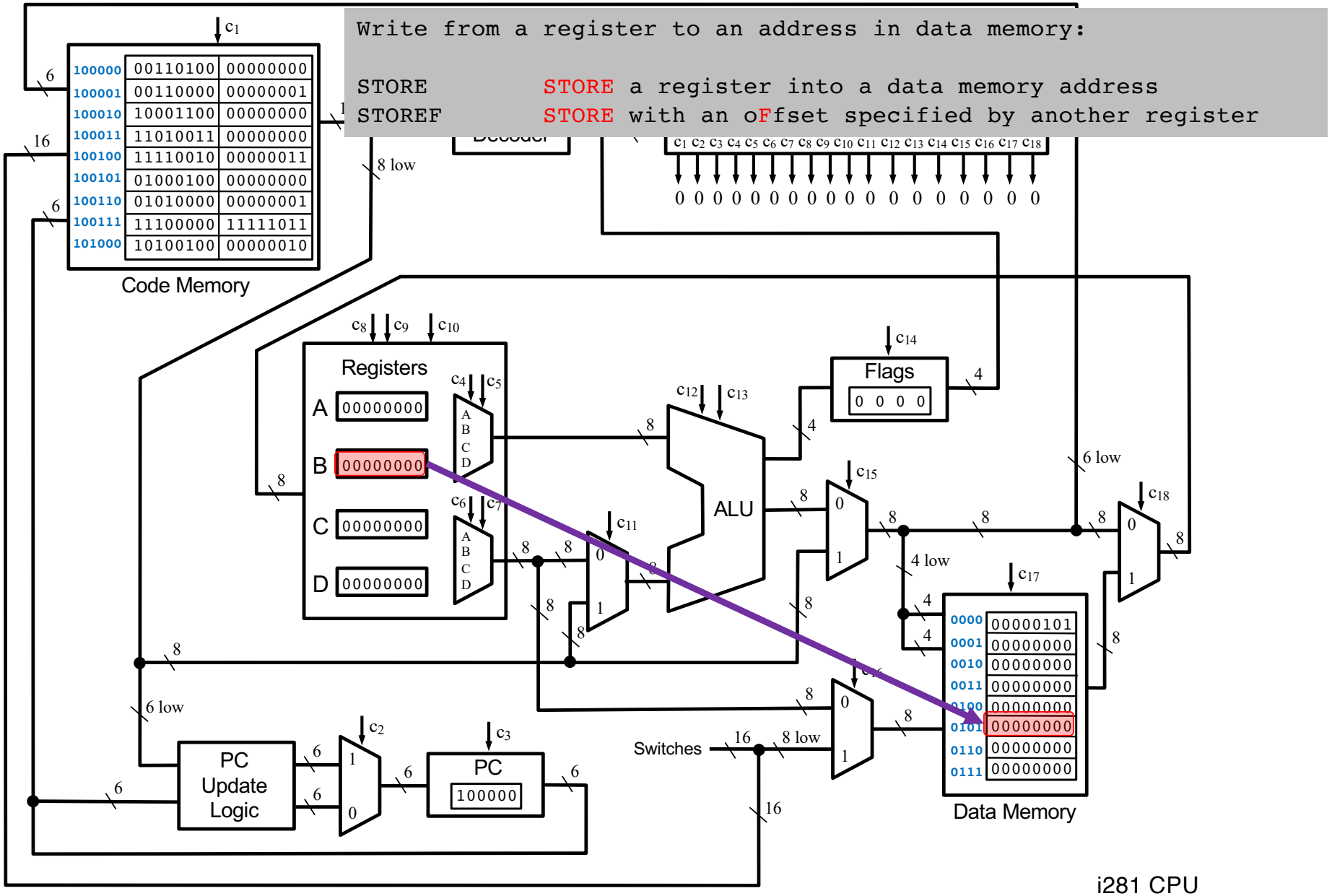

Instruction that changes the value of the Program Counter by adding a number stored as part of the instruction. This is unconditional, does not depend on the Flags register.
JUMP JUMP unconditionally to a specified address
Code Memory
Registers
Flags
ALU
PC
PC Update Logic
Switches
Data Memory
i281 CPU

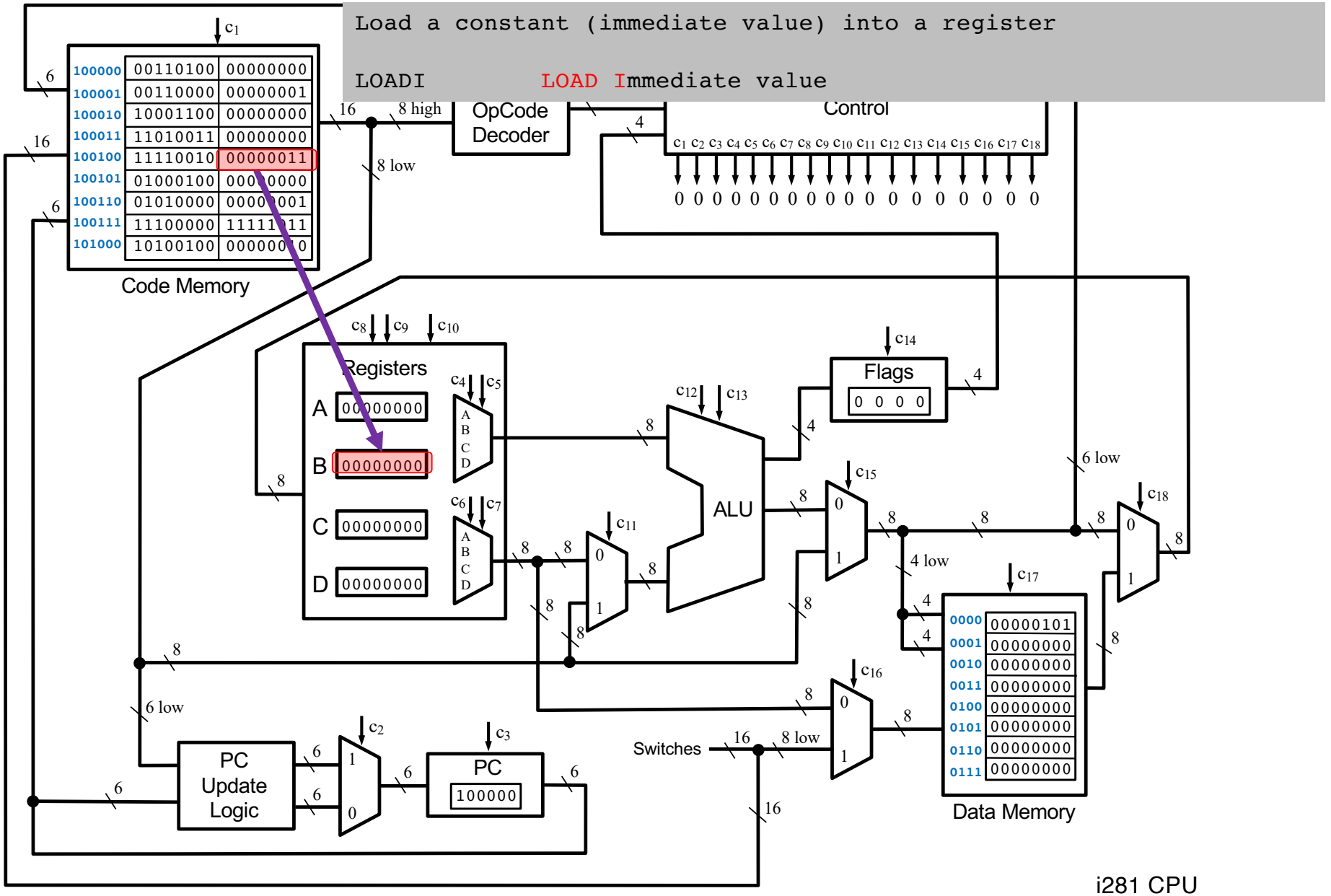

Arithmetic operations with two registers:
ADD ADD two registers
SUB SUBtract two registers
CMP CoMPare the values in two registers
Code Memory
Registers
A 00000000
B 00000000
C 00000000
D 00000000
ALU
Flags
0 0 0 0
PC
100000
PC Update Logic
Switches
Data Memory
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
i281 CPU

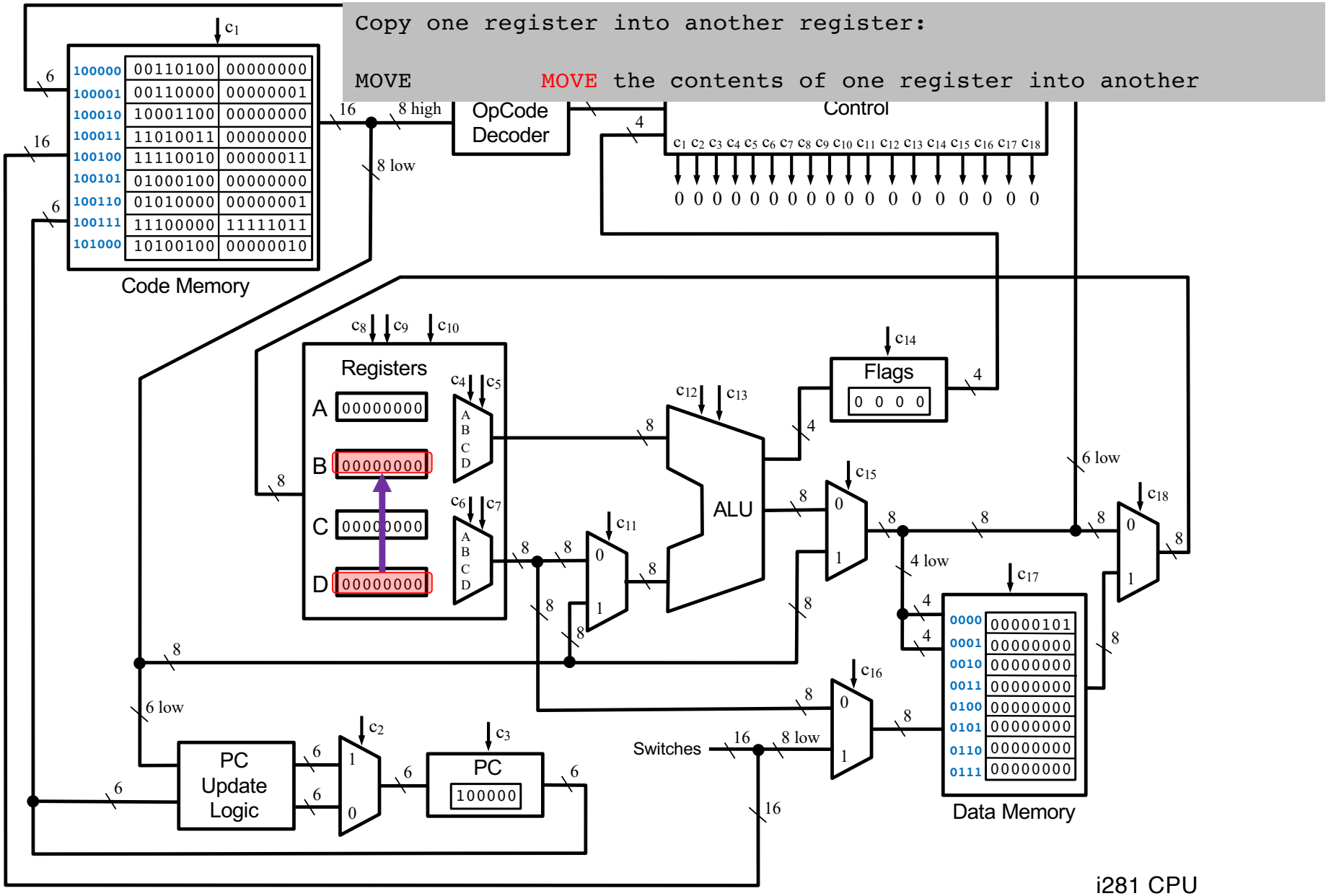

Arithmetic operations with a register and a constant:
ADDI ADD an Immediate value to a register
SUBI SUBtract an Immediate value from a register
Code Memory
Registers
Flags
ALU
PC
PC Update Logic
Data Memory
Switches
i281 CPU


Arithmetic operations with only one register:
SHIFTL SHIFT Left all bits in a register
SHIFTR SHIFT Right all bits in a register
Code Memory
Registers
A
B
C
D
ALU
Flags
PC
PC Update Logic
Data Memory
Switches
i281 CPU


Read from the data memory into a register:
LOAD LOAD from a data memory address into a register
LOADF LOAD with an oFfset specified by another register
Code Memory
Registers
A
B
C
D
ALU
Flags
PC
Update
Logic
PC
Switches
Data Memory
i281 CPU


Write from a register to an address in data memory:
STORE STORE a register into a data memory address
STOREF STORE with an oFfset specified by another register
Code Memory
Registers
A
B
C
D
ALU
Flags
PC
Update
Logic
PC
Switches
Data Memory
i281 CPU


Load a constant (immediate value) into a register
LOADI LOAD Immediate value
c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000000
Code Memory
16
8 high
8 low
OpCode Decoder
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
c12 c13
ALU
c14
Flags
0 0 0 0
c15
c11
c16
c17
c18
6 low
4 low
Switches
8 low
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
PC Update Logic
c2
c3
PC
100000
i281 CPU


Copy one register into another register:
MOVE MOVE the contents of one register into another
c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
16
8 high
8 low
OpCode Decoder
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
c12 c13
ALU
c14
Flags
0 0 0 0
c15
c11
c16
c17
c18
6 low
4 low
8 low
Switches
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
PC Update Logic
c2
c3
PC
100000
i281 CPU

The OPCODEs

(With More Details)

NOOP

Full name:

NO OPeration

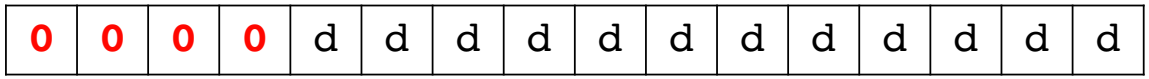
Description:

Do nothing for one clock cycle, i.e., idle.

Assembly Example:

NOOP

Instruction Layout:



ADD

Full name:

ADD two registers

Description:

Add the values stored in two registers. The result of the addition is stored in the first register, overwriting the previous value. The flags are updated based on the result of the addition.

Assembly Example:

```
ADD  A, C
```

Instruction Layout:

0	1	0	0	0	0	1	0	d	d	d	d	d	d	d	d

ADD

		Register
0	0	A
0	1	B
1	0	C
1	1	D

Full name:

ADD two registers

Description:

Add the values stored in two registers. The result of the addition is stored in the first register, overwriting the previous value. The flags are updated based on the result of the addition.

Assembly Example:

```
ADD  A, C
```

Instruction Layout:

0	1	0	0	0	0	1	0	d	d	d	d	d	d	d	d

MOVE

Full name:

MOVE

Description:

Move (i.e., copy) the contents of the second register into the first register, overwriting the first register.

This is implemented as A=B+0. Thus, the mandatory zeros in the last 8 bits, which must go through the ALU.

The flags are <u>not</u> updated.

Assembly Example:

MOVE A, B

Instruction Layout:

SUB

Full name:

SUBtract two registers

Description:

Subtract the values stored in two registers. This is done by subtracting the second register from the first register. The result of the subtraction is stored in the first register. The flags are updated.

Assembly Example:

```
SUB B, C
```

Instruction Layout:

0	1	1	0	0	1	1	0	d	d	d	d	d	d	d	d

CMP

Full name:

CoMPare the values stored in two registers

Description:

Compare two registers by subtracting the second register from the first register. The result of the subtraction is not stored. Only the flags are updated.

Assembly Example:

```
CMP  D, C
```

Instruction Layout:

1	1	0	1	1	1	1	0	d	d	d	d	d	d	d	d

CMP

		Register
0	0	A
0	1	B
1	0	C
1	1	D

Full name:

CoMPare the values stored in two registers

Description:

Compare two registers by subtracting the second register from the first register. The result of the subtraction is not stored. Only the flags are updated.

Assembly Example:

```
CMP  D, C
```

Instruction Layout:

1	1	0	1	1	1	1	0	d	d	d	d	d	d	d	d

SHIFTL

Full name:

SHIFT Left

Description:

Shift left all bits in a register. The bit that is shifted out is stored in the carry flag. The LSB is set 0.

Update the flags based on the final value.

Assembly Example:

```
SHIFTL  B
```

Instruction Layout:

1	1	0	0	0	1	d	0	d	d	d	d	d	d	d	d

SHIFTR

Full name:

SHIFT Right

Description:

Shift right all bits in a register. The bit that is shifted out is stored in the carry flag. The MSB is set 0.

Update the flags based on the final value.

Assembly Example:

```
SHIFTR  C
```

Instruction Layout:

1	1	0	0	1	0	d	1	d	d	d	d	d	d	d	d

JUMP

Full name:

JUMP

Description:

Unconditional jump to the specified address, which is given as a label for some row of the assembly program but converted to a PC offset in the machine code.

Assembly Example:

```
JUMP End
```

Instruction Layout:

1	1	1	0	d	d	d	d	P	C	O	F	F	S	E	T

BRE

Full name:

BRanch if Equal (identical to BRE)

Description:

Conditional jump to the specified address, which is given as a label for some row of the assembly program but converted to a PC offset in the machine code.

Assembly Example:

```
BRE  Start
```

Instruction Layout:

1	1	1	1	d	d	0	0	P	C	O	F	F	S	E	T

BRZ

Full name:

BRanch if Zero (identical to BRE)

Description:

Conditional jump to the specified address, which is given as a label for some row of the assembly program but converted to a PC offset in the machine code.

Assembly Example:

```
BRZ  Start
```

Instruction Layout:

1	1	1	1	d	d	0	0	P	C	O	F	F	S	E	T

BRNE

Full name:

BRanch if Not Equal (identical to BRNZ)

Description:

Conditional jump to the specified address, which is given as a label for some row of the assembly program but converted to a PC offset in the machine code.

Assembly Example:

```
BRNE  Loop
```

Instruction Layout:

1	1	1	1	d	d	0	1	P	C	O	F	F	S	E	T

BRNZ

Full name:

BRanch if Not Zero (identical to BRNE)

Description:

Conditional jump to the specified address, which is given as a label for some row of the assembly program but converted to a PC offset in the machine code.

Assembly Example:

```
BRNZ  Loop
```

Instruction Layout:

1	1	1	1	d	d	0	1	P	C	O	F	F	S	E	T

BRG

Full name:

BRanch if Greater

Description:

Conditional jump to the specified address, which is given as a label for some row of the assembly program but converted to a PC offset in the machine code.

Assembly Example:

```
BRG  InnerLoop
```

Instruction Layout:

1	1	1	1	d	d	1	0	P	C	O	F	F	S	E	T

BRGE

Full name:

BRanch if Greater

Description:

Conditional jump to the specified address, which is given as a label for some row of the assembly program but converted to a PC offset in the machine code.

Assembly Example:

```
BRGE  OuterLoop
```

Instruction Layout:

ADDI

Full name:

ADD an Immediate value to a register

Description:

Add the immediate value, which is stored in the last 8 bits of the instruction in the code memory, to the register.

The result of the addition is stored in the same register.

The flags are updated.

Assembly Example:

```
ADDI  A, 3
```

Instruction Layout:

0	1	0	1	0	0	d	d	0	0	0	0	0	0	1	1

SUBI

Full name:

SUBtract an Immediate value from a register

Description:

Subtract the immediate value, which is stored in the last 8 bits of the instruction in the code memory, from the register.

The result of the subtraction is stored in the same register.

The flags are updated.

Assembly Example:

```
SUBI  C, 5
```

Instruction Layout:

0	1	1	1	1	0	d	d	0	0	0	0	0	1	0	1

LOADI

Full name:

LOAD an Immediate value into a register

Description:

Load the immediate value, which is stored in the last 8 bits of the instruction in the code memory, into the register.

The old value of the register is overwritten.

Assembly Example:

```
LOADI  B, 6
```

Instruction Layout:

0	0	1	1	0	1	d	d	0	0	0	0	0	1	1	0

LOADP

Full name:

LOAD a Pointer address into a register

Description:

Load the address of a data memory location into a register. The address is specified by the label for that memory cell, surrounded by curly brackets. The compiler computes the address and stores it in the last 8 bits of the instruction.

Assembly Example:

```
.data
        M  BYTE  3   ; this is stored at address 0
        N  BYTE  5   ; this is stored at address 1
.code
        LOADP  A, {M}
```

Instruction Layout:

0	0	1	1	0	0	d	d	0	0	0	0	0	0	0	0

LOADP

Full name:

LOAD a Pointer address into a register

Description:

Load the address of a data memory location into a register. The address is specified by the label for that memory cell, surrounded by curly brackets. The compiler computes the address and stores it in the last 8 bits of the instruction.

Assembly Example:

```
.data
        M  BYTE  3   ; this is stored at address 0
        N  BYTE  5   ; this is stored at address 1
.code
        LOADP  A, {M+1}     ; +1 is an optional additive constant
```

Instruction Layout:

0	0	1	1	0	0	d	d	0	0	0	0	0	0	0	1

LOAD

Full name:

LOAD the value from a data memory address into a register

Description:

Load the contents of a data memory cell into a register. The address is specified by the label for that memory location, surrounded by square brackets. The compiler computes the address and stores it in the last 8 bits of the instruction.

Assembly Example:

```
.data
        N       BYTE  3          ; this is stored at address 0
        array   BYTE  4, 7, 1    ; this is stored at addresses 1, 2, 3
.code
        LOAD  C, [array]
```

Instruction Layout:

1	0	0	0	1	0	d	d	0	0	0	0	0	0	0	1

LOAD

Full name:

LOAD the value from a data memory address into a register

Description:

Load the contents of a data memory cell into a register. The address is specified by the label for that memory location, surrounded by square brackets. The compiler computes the address and stores it in the last 8 bits of the instruction.

Assembly Example:

```
.data
        N       BYTE  3          ; this is stored at address 0
        array   BYTE  4, 7, 1    ; this is stored at addresses 1, 2, 3
.code
        LOAD  C, [array+2]     ; +2 is an optional offset constant
```

Instruction Layout:

1	0	0	0	1	0	d	d	0	0	0	0	0	0	1	1

LOADF

Full name:

LOAD but with an oFfset specified by another register

Description:

Load the contents of a data memory cell into a register. The address of the cell is specified by a label plus an offset value stored in a register, surrounded by square brackets. The compiler computes the address of the label and stores it in the last 8 bits of the instruction. The offset is added at runtime to compute the effective address from which the value is loaded.

Assembly Example:

```
.data
        N       BYTE  3          ; this is stored at address 0
        array   BYTE  4, 7, 1    ; this is stored at addresses 1, 2, 3
.code
        LOADI  B, 1
        LOADF  C, [array + B]     ; this will load into C the value 7
```

Instruction Layout:

1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1

LOADF

Full name:

LOAD but with an oFfset specified by another register

Description:

Load the contents of a data memory cell into a register. The address of the cell is specified by a label plus an offset value stored in a register, surrounded by square brackets. The compiler computes the address of the label and stores it in the last 8 bits of the instruction. The offset is added at runtime to compute the effective address from which the value is loaded.

Assembly Example:

```
.data
        N       BYTE  3          ; this is stored at address 0
        array   BYTE  4, 7, 1    ; this is stored at addresses 1, 2, 3
.code
        LOADI  B, 1
        LOADF  C, [array + B + 1]     ; this will load into C the value 1
```

Instruction Layout:

1	0	0	1	1	0	0	1	0	0	0	0	0	0	1	0

STORE

Full name:

STORE a register value into a data memory location

Description:

Store the value of a register into a data memory cell. The address is specified by the label for that memory location, surrounded by square brackets. The compiler computes the address and stores it in the last 8 bits of the instruction.

Assembly Example:

```
.data
        N       BYTE  3          ; this is stored at address 0
        array   BYTE  4, 7, 1    ; this is stored at addresses 1, 2, 3
.code
        LOADI  D, 5              ; set register D to 5
        STORE  [array], D        ; this will replace the 4 with 5 in array
```

Instruction Layout:

1	0	1	0	1	1	d	d	0	0	0	0	0	0	0	1

STORE

Full name:

STORE a register value into a data memory location

Description:

Store the value of a register into a data memory cell. The address is specified by the label for that memory location, surrounded by square brackets. The compiler computes the address and stores it in the last 8 bits of the instruction.

Assembly Example:

```
.data
        N       BYTE  3          ; this is stored at address 0
        array   BYTE  4, 7, 1    ; this is stored at addresses 1, 2, 3
.code
        LOADI  D, 5              ; set register D to 5
        STORE  [array+2], D      ; this will replace the 1 with 5 in array
```

Instruction Layout:

1	0	1	0	1	1	d	d	0	0	0	0	0	0	1	1

STOREF

Full name:

STORE but with an oFfset specified by another register

Description:

Store the value of a register into a data memory cell. The address of the cell is specified by a label plus an offset value stored in a register, surrounded by square brackets. The compiler computes the address of the label and stores it in the last 8 bits of the instruction. The offset is added at runtime to compute the effective address at which the value is stored.

Assembly Example:

```
.data
        array  BYTE  4, 7, 8   ; this is stored at addresses 0, 1, 2
.code
        LOADI  B, 1                 ; register B is set to 1
        LOADI  D, 6                 ; register D is set to 6
        STOREF  [array + B], D      ; this will replace the 7 with 6 in array
```

Instruction Layout:

1	0	1	1	1	1	0	1	0	0	0	0	0	0	0	0

STOREF

Full name:

STORE but with an oFfset specified by another register

Description:

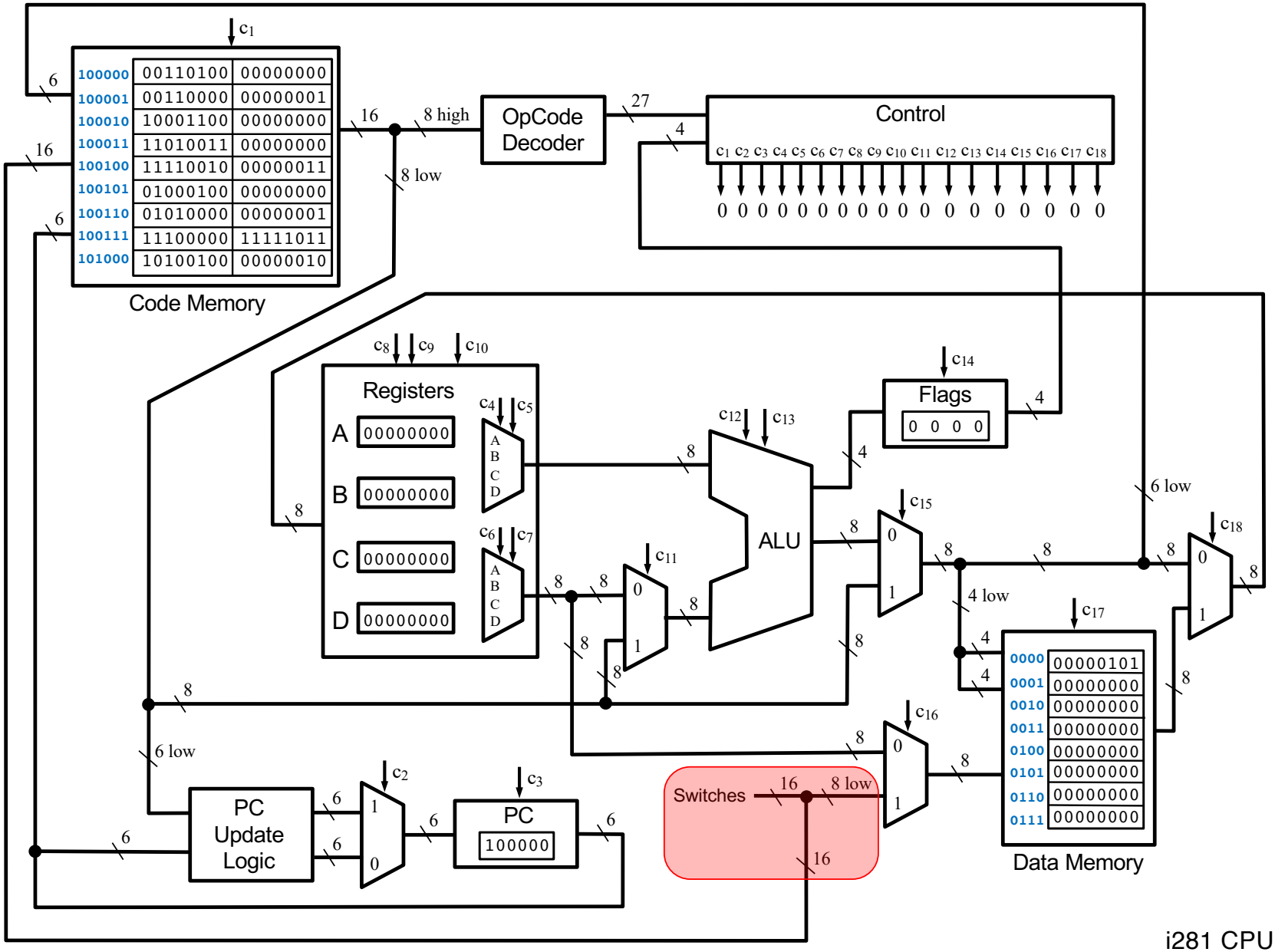
Store the value of a register into a data memory cell. The address of the cell is specified by a label plus an offset value stored in a register, surrounded by square brackets. The compiler computes the address of the label and stores it in the last 8 bits of the instruction. The offset is added at runtime to compute the effective address at which the value is stored.

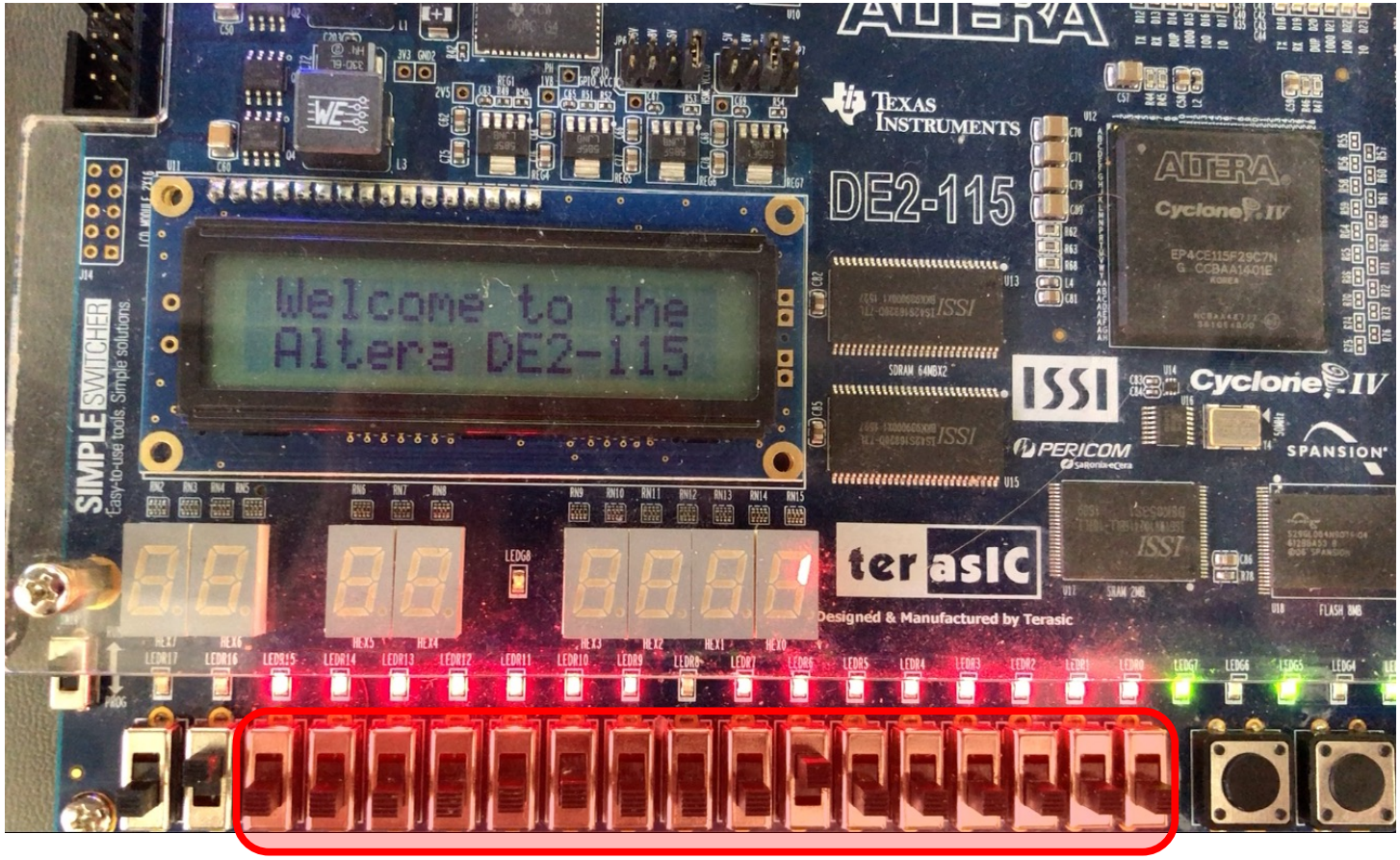
Assembly Example:

```
.data
        array  BYTE  4, 7, 8   ; this is stored at addresses 0, 1, 2
.code
        LOADI  B, 1                    ; register B is set to 1
        LOADI  D, 6                    ; register D is set to 6
        STOREF  [array + B +1], D      ; this will replace the 8 with 6 in array
```

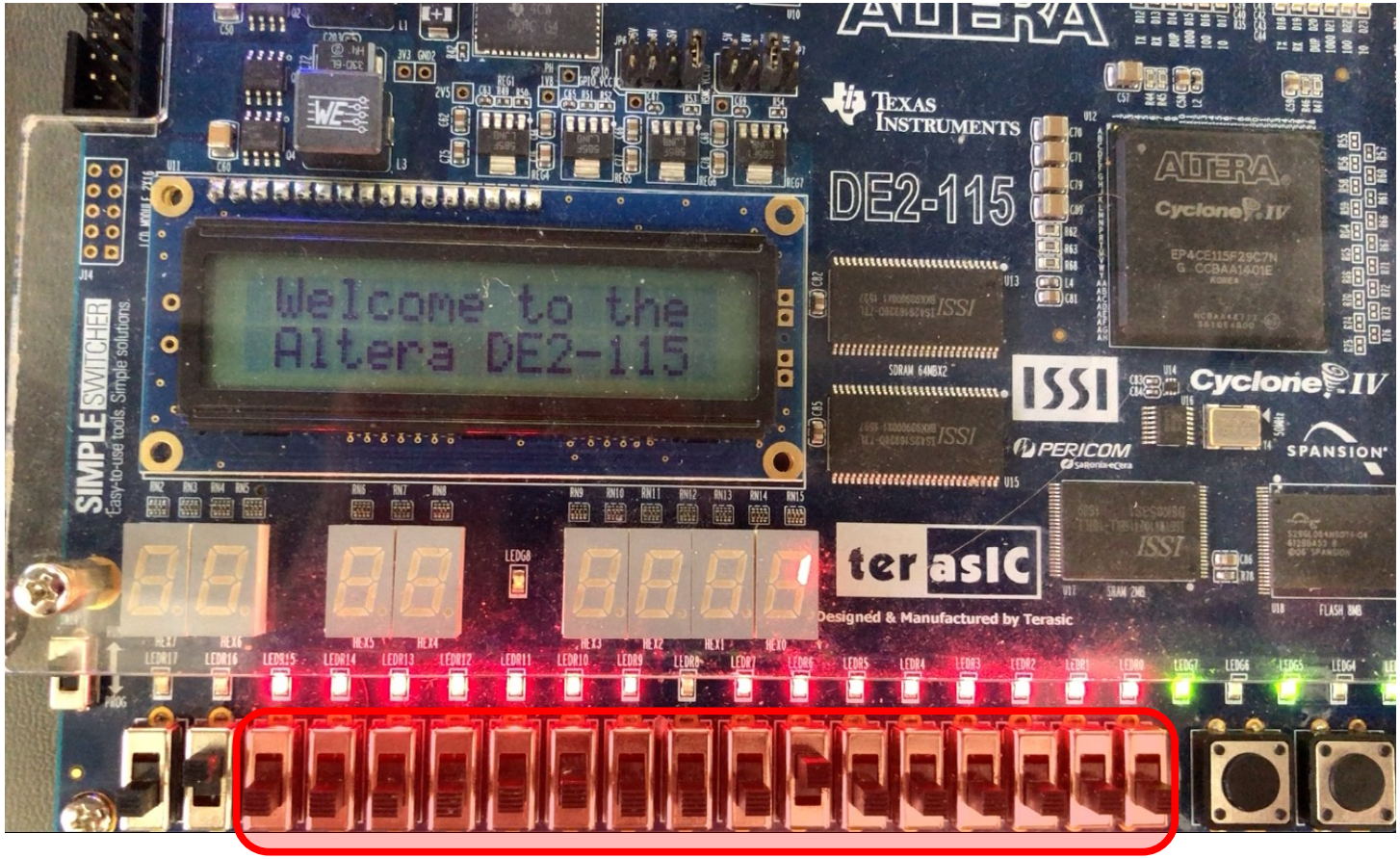
Instruction Layout:

1	0	1	1	1	1	0	1	0	0	0	0	0	0	0	1

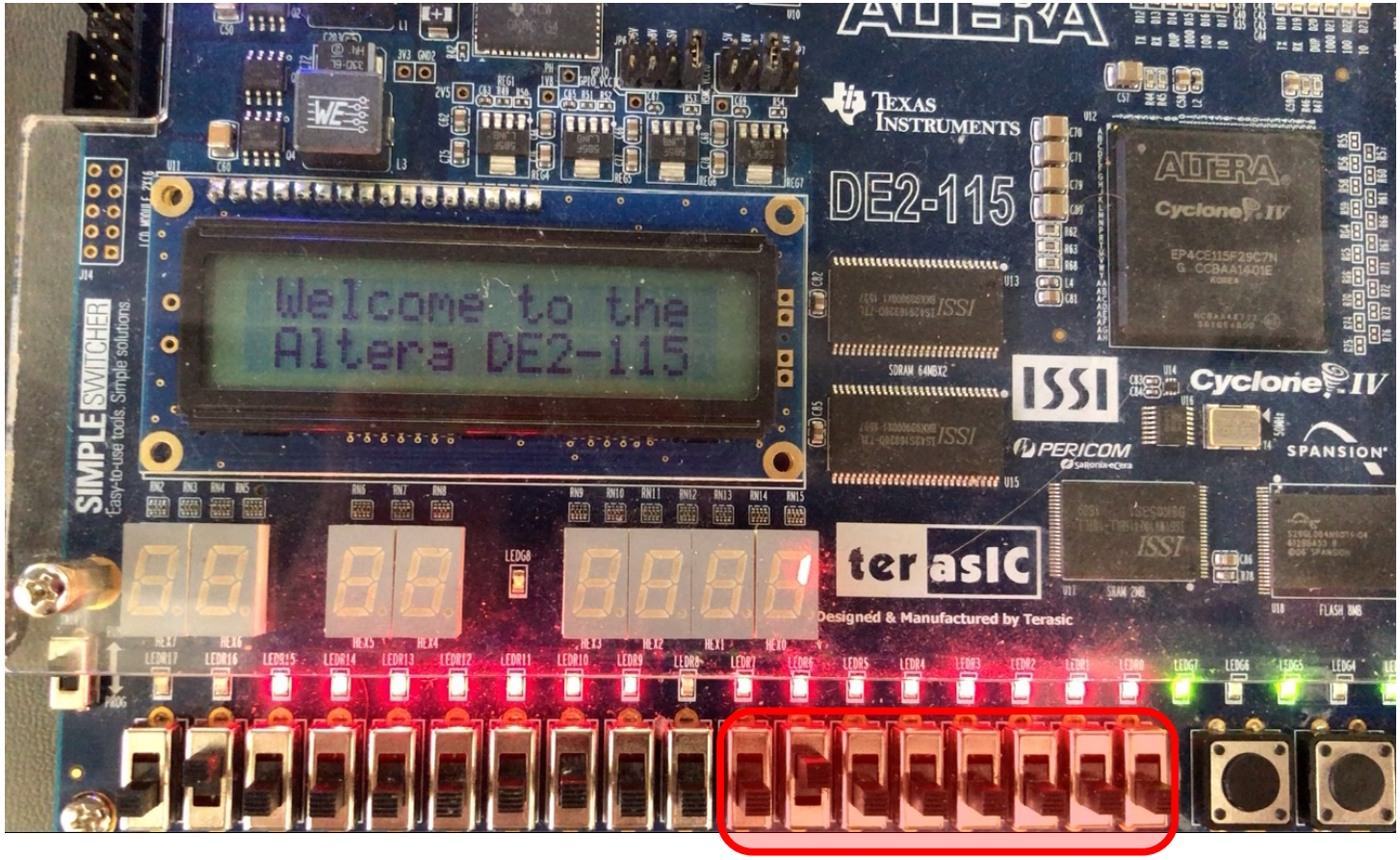

c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
c12 c13
ALU
c14
Flags
0 0 0 0
c15
c11
c16
c17
c18
6 low
4 low
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
PC Update Logic
c2
c3
PC
100000
Switches
16
8 low
i281 CPU



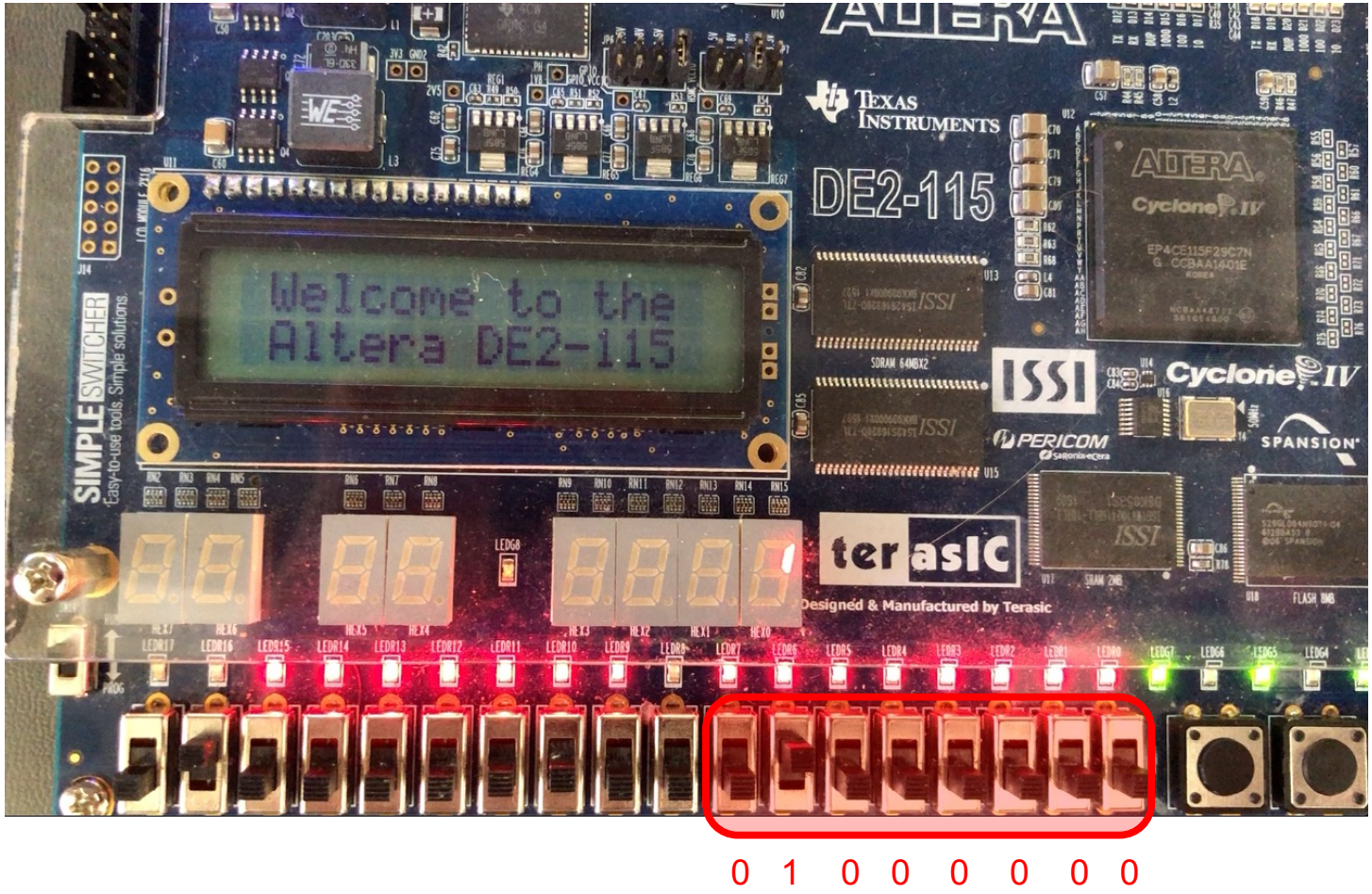
These 16 switches are used for input into the Code Memory.



0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0



These 8 switches are used for input into the Data Memory.


Welcome to the
Altera DE2-115
DE2-115
ALTERA
Cyclone IV
EP4CE115F29C7N
Texas Instruments
ISSI
PERICOM
SPANSION
terasIC
Designed & Manufactured by Terasic
SIMPLE SWITCHER
Easy-to-use tools. Simple solutions.
0 1 0 0 0 0 0 0

INPUTC

Full name:

INPUT into Code memory

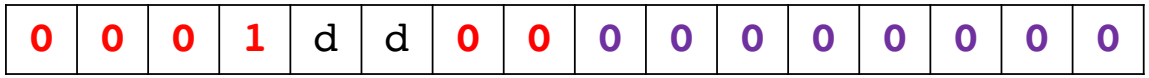
Description:

Read a 16-bit value (from switches SW15-SW0) and store it in the code memory at the given address. The address is specified by the label for that memory location, surrounded by square brackets. The compiler computes the address and stores it in the last 8 bits of the instruction.

Assembly Example:

```
 .data
        zero   BYTE  ?          ; this is stored at address 0 in DMEM
.code
        INPUTC  [zero]          ; store the value @ address 0 in the CMEM
                                ; Hack to get around  a compiler issue.
                                ; Currently the CMEM address can only be
                                ; given as a label to a DMEM location.
```

Instruction Layout:



INPUTC

Full name:

INPUT into Code memory

Description:

Read a 16-bit value (from switches SW15-SW0) and store it in the code memory at the given address. The address is specified by the label for that memory location, surrounded by square brackets. The compiler computes the address and stores it in the last 8 bits of the instruction.

Assembly Example:

```
 .data
        zero   BYTE  ?          ; this is stored at address 0 in DMEM
.code
        INPUTC  [zero+32]   ; store the value @ address 32 in the CMEM
                            ; Hack to get around  a compiler issue.
                            ; Currently the CMEM address can only be
                            ; given as a label to a DMEM location.
```

Instruction Layout:

0	0	0	1	d	d	0	0	0	0	1	0	0	0	0	0

INPUTCF

Full name:

INPUT into Code memory with oFfset

Description:

Read a 16-bit value (from switches SW15-SW0) and store it in the code memory at the given address. The address is specified by a label plus an offset value stored in a register, surrounded by square brackets. The compiler computes the address of the label and stores it in the last 8 bits of the instruction. The offset is added at runtime to compute the effective address at which the value is stored.

Assembly Example:

```
.data
        zero   BYTE  ?          ; this is stored at address 0 in DMEM
.code
        LOADI   D, 5
        INPUTCF  [zero + D]  ; store the value @ address 5 in the CMEM
```

Instruction Layout:

0	0	0	1	d	d	0	1	0	0	0	0	0	0	0	0

INPUTCF

Full name:

INPUT into Code memory with oFfset

Description:

Read a 16-bit value (from switches SW15-SW0) and store it in the code memory at the given address. The address is specified by a label plus an offset value stored in a register, surrounded by square brackets. The compiler computes the address of the label and stores it in the last 8 bits of the instruction. The offset is added at runtime to compute the effective address at which the value is stored.

Assembly Example:

```
.data
        zero   BYTE  ?          ; this is stored at address 0 in DMEM
.code
        LOADI   D, 5
        INPUTCF  [zero + D + 32]  ; store @ address 37 in the CMEM
```

Instruction Layout:

0	0	0	1	d	d	0	1	0	0	1	0	0	0	0	0

INPUTD

Full name:

INPUT into Data memory

Description:

Read an 8-bit value (from switches SW7-SW0) and store it in the data memory at the given address. The address is specified by the label for that memory location, surrounded by square brackets. The compiler computes the address and stores it in the last 8 bits of the instruction.

Assembly Example:

```
 .data
        M   BYTE   1          ; this is stored at address 0 in DMEM
        N    BYTE  ?          ; this is at address 1 in DMEM
.code
        INPUTD  [N]        ; store the value @ address 1 in the DMEM
```

Instruction Layout:

0	0	0	1	d	d	1	0	0	0	0	0	0	0	0	1

INPUTD

Full name:

INPUT into Data memory

Description:

Read an 8-bit value (from switches SW7-SW0) and store it in the data memory at the given address. The address is specified by the label for that memory location, surrounded by square brackets. The compiler computes the address and stores it in the last 8 bits of the instruction.

Assembly Example:

```
 .data
        M   BYTE   1          ; this is stored at address 0 in DMEM
        X   BYTE   ?,?,?      ; this is at addresses 1, 2, 3 in DMEM
.code
        INPUTD  [X+2]         ; store the value @ address 3 in the DMEM
```

Instruction Layout:

0	0	0	1	d	d	1	0	0	0	0	0	0	0	1	1

INPUTDF

Full name:

INPUT into Data memory with oFfset

Description:

Read an 8-bit value (from switches SW7-SW0) and store it in the data memory at the given address. The address is specified by the label for that memory location plus an offset value stored in a register, surrounded by square brackets. The compiler computes the address of the label and stores it in the last 8 bits of the instruction. The offset is added at runtime to compute the effective address at which the value from the switches is stored.

Assembly Example:

```
.data
        M   BYTE   1            ; this is stored at address 0 in DMEM
        X   BYTE  ?,?,?,?       ; this is at addresses 1, 2, 3, 4 in DMEM
.code
        LOADI   B, 2
        INPUTDF  [X + B]    ; store the value @ address 3 in the DMEM
```

Instruction Layout:

0	0	0	1	0	1	1	1	0	0	0	0	0	0	0	1

INPUTDF

Full name:

INPUT into Data memory with oFfset

Description:

Read an 8-bit value (from switches SW7-SW0) and store it in the data memory at the given address. The address is specified by the label for that memory location plus an offset value stored in a register, surrounded by square brackets. The compiler computes the address of the label and stores it in the last 8 bits of the instruction. The offset is added at runtime to compute the effective address at which the value from the switches is stored.

Assembly Example:

```
.data
        M   BYTE   1            ; this is stored at address 0 in DMEM
        X   BYTE  ?,?,?,?       ; this is at addresses 1, 2, 3, 4 in DMEM
.code
        LOADI   B, 2
        INPUTDF  [X + B + 1]    ; store the value @ address 4 in the DMEM
```

Instruction Layout:

0	0	0	1	0	1	1	1	0	0	0	0	0	0	1	0

How many unique assembly instructions are there?

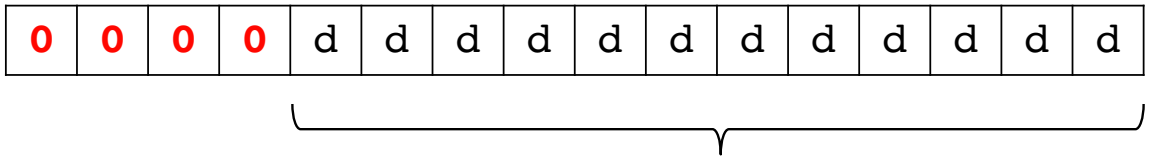
(examples for some of the instructions)

There is only one NOOP instruction

NOOP

There is only one NOOP instruction

NOOP

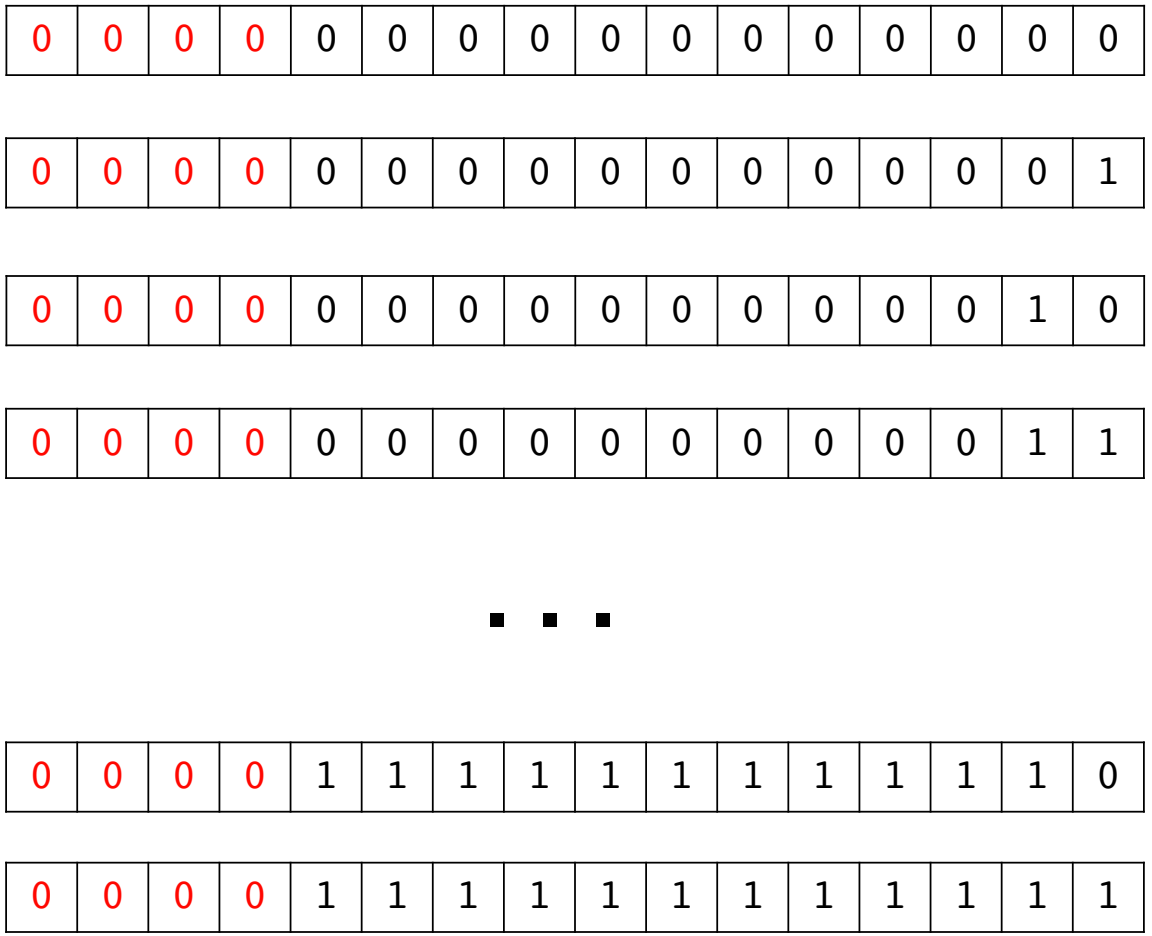


Because these 12 bits are don't cares, however, there are $2^{12} = 4096$ possible ways to map the assembly instruction NOOP to machine language.

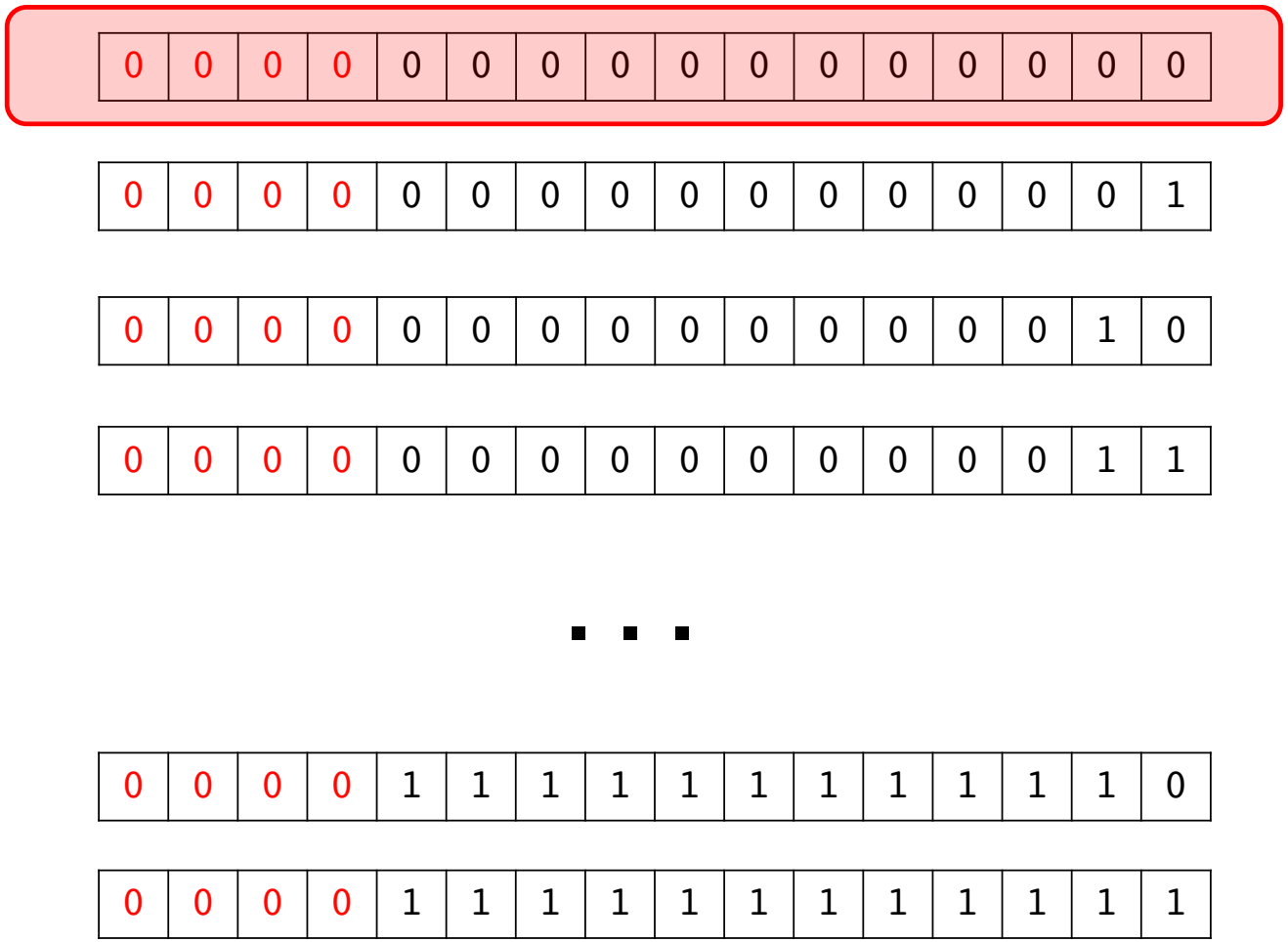
The hardware ignores the don't care bits. So, any one of these 4096 mappings leads to a valid machine language command.

The compiler, however, maps all d bits to zero. Thus, mapping NOOP to 16 zeros.

There are 4096 ways to map the assembly NOOP instruction to machine language



The compiler from assembly to machine language uses only this one by default



There are 16 possible ADD instructions

ADD A, A	ADD B, A	ADD C, A	ADD D, A
ADD A, B	ADD B, B	ADD C, B	ADD D, B
ADD A, C	ADD B, C	ADD C, C	ADD D, C
ADD A, D	ADD B, D	ADD C, D	ADD D, D

There are 16 possible ADD instructions

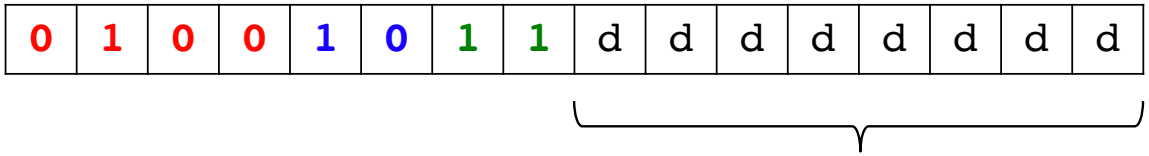
ADD A, A	0	1	0	0	0	0	0	0	d	d	d	d	d	d	d	d
ADD A, B	0	1	0	0	0	0	0	1	d	d	d	d	d	d	d	d
ADD A, C	0	1	0	0	0	0	1	0	d	d	d	d	d	d	d	d
ADD A, D	0	1	0	0	0	0	1	1	d	d	d	d	d	d	d	d
ADD B, A	0	1	0	0	0	1	0	0	d	d	d	d	d	d	d	d
ADD B, B	0	1	0	0	0	1	0	1	d	d	d	d	d	d	d	d
ADD B, C	0	1	0	0	0	1	1	0	d	d	d	d	d	d	d	d
ADD B, D	0	1	0	0	0	1	1	1	d	d	d	d	d	d	d	d

There are 16 possible ADD instructions

ADD C, A	0	1	0	0	1	0	0	0	d	d	d	d	d	d	d	d
ADD C, B	0	1	0	0	1	0	0	1	d	d	d	d	d	d	d	d
ADD C, C	0	1	0	0	1	0	1	0	d	d	d	d	d	d	d	d
ADD C, D	0	1	0	0	1	0	1	1	d	d	d	d	d	d	d	d
ADD D, A	0	1	0	0	1	1	0	0	d	d	d	d	d	d	d	d
ADD D, B	0	1	0	0	1	1	0	1	d	d	d	d	d	d	d	d
ADD D, C	0	1	0	0	1	1	1	0	d	d	d	d	d	d	d	d
ADD D, D	0	1	0	0	1	1	1	1	d	d	d	d	d	d	d	d

The last 8 bits could be set to anything

ADD C, D



Because these 8 bits are don't cares, however, there are $2^8 = 256$ possible ways to map the assembly instruction ADD C,D to machine language.

The hardware ignores the don't care bits. So any one of these 256 mappings leads to a valid machine language command.

The compiler, however, maps all d bits to zero.

There are 16 possible SUB instructions

SUB A, A	SUB B, A	SUB C, A	SUB D, A
SUB A, B	SUB B, B	SUB C, B	SUB D, B
SUB A, C	SUB B, C	SUB C, C	SUB D, C
SUB A, D	SUB B, D	SUB C, D	SUB D, D

There are 16 possible SUB instructions

SUB A, A	0	1	1	0	0	0	0	0	d	d	d	d	d	d	d	d
SUB A, B	0	1	1	0	0	0	0	1	d	d	d	d	d	d	d	d
SUB A, C	0	1	1	0	0	0	1	0	d	d	d	d	d	d	d	d
SUB A, D	0	1	1	0	0	0	1	1	d	d	d	d	d	d	d	d
SUB B, A	0	1	1	0	0	1	0	0	d	d	d	d	d	d	d	d
SUB B, B	0	1	1	0	0	1	0	1	d	d	d	d	d	d	d	d
SUB B, C	0	1	1	0	0	1	1	0	d	d	d	d	d	d	d	d
SUB B, D	0	1	1	0	0	1	1	1	d	d	d	d	d	d	d	d

There are 16 possible SUB instructions

SUB C, A	0	1	1	0	1	0	0	0	d	d	d	d	d	d	d	d
SUB C, B	0	1	1	0	1	0	0	1	d	d	d	d	d	d	d	d
SUB C, C	0	1	1	0	1	0	1	0	d	d	d	d	d	d	d	d
SUB C, D	0	1	1	0	1	0	1	1	d	d	d	d	d	d	d	d
SUB D, A	0	1	1	0	1	1	0	0	d	d	d	d	d	d	d	d
SUB D, B	0	1	1	0	1	1	0	1	d	d	d	d	d	d	d	d
SUB D, C	0	1	1	0	1	1	1	0	d	d	d	d	d	d	d	d
SUB D, D	0	1	1	0	1	1	1	1	d	d	d	d	d	d	d	d

There are 4 possible SHIFTL instructions

SHIFTL A

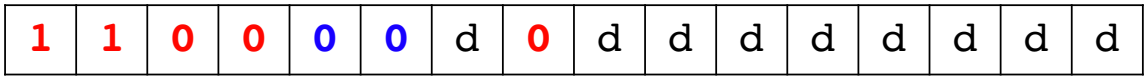
SHIFTL B

SHIFTL C

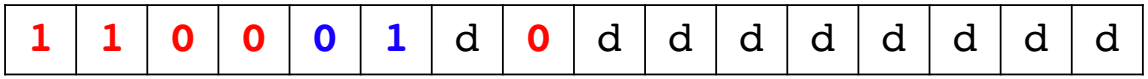
SHIFTL D

There are 4 possible SHIFTL instructions

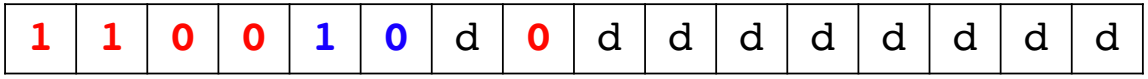
SHIFTL A



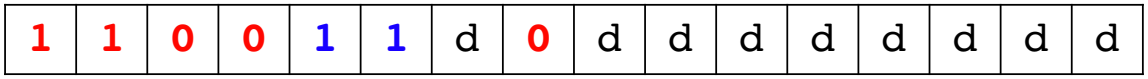
SHIFTL B



SHIFTL C

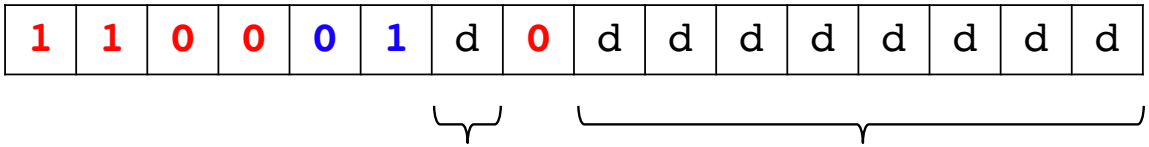


SHIFTL D



There are 4 possible SHIFTL instructions

SHIFTL B



Because these 9 bits are don't cares, however, there are $2^9 = 512$ possible ways to map the assembly instruction SHIFTL B to machine language.

The hardware ignores the don't care bits. So, any one of these 512 mappings leads to a valid machine language command.

The compiler, however, maps all d bits to zero.

There are 4 possible SHIFTR instructions

SHIFTR A

SHIFTR B

SHIFTR C

SHIFTR D

There are 4 possible SHIFTR instructions

SHIFTR A	1	1	0	0	0	0	d	1	d	d	d	d	d	d	d	d
SHIFTR B	1	1	0	0	0	1	d	1	d	d	d	d	d	d	d	d
SHIFTR C	1	1	0	0	1	0	d	1	d	d	d	d	d	d	d	d
SHIFTR D	1	1	0	0	1	1	d	1	d	d	d	d	d	d	d	d

There are many possible JUMP instructions (at the assembly language level)

JUMP Start

JUMP InnerLoop

JUMP OuterLoop

JUMP End

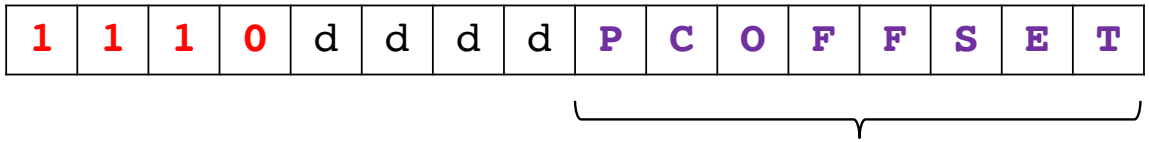
The second word must be an unique label. The number of possible commands depends on the maximum length of the label supported by the compiler.

There are many possible JUMP instructions

JUMP Label

There are many possible JUMP instructions

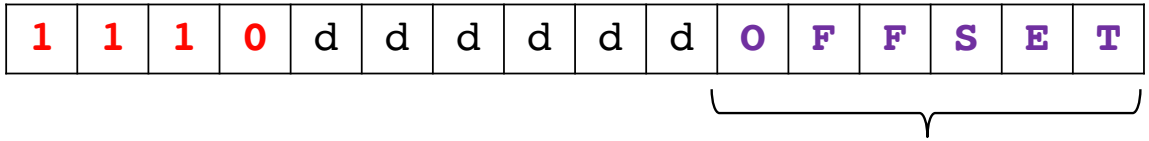
JUMP Label



These 8 bits specify an offset for the program counter (PC).
The offset is encoded in 2's complement representation
and can be either positive or negative.

There are many possible JUMP instructions

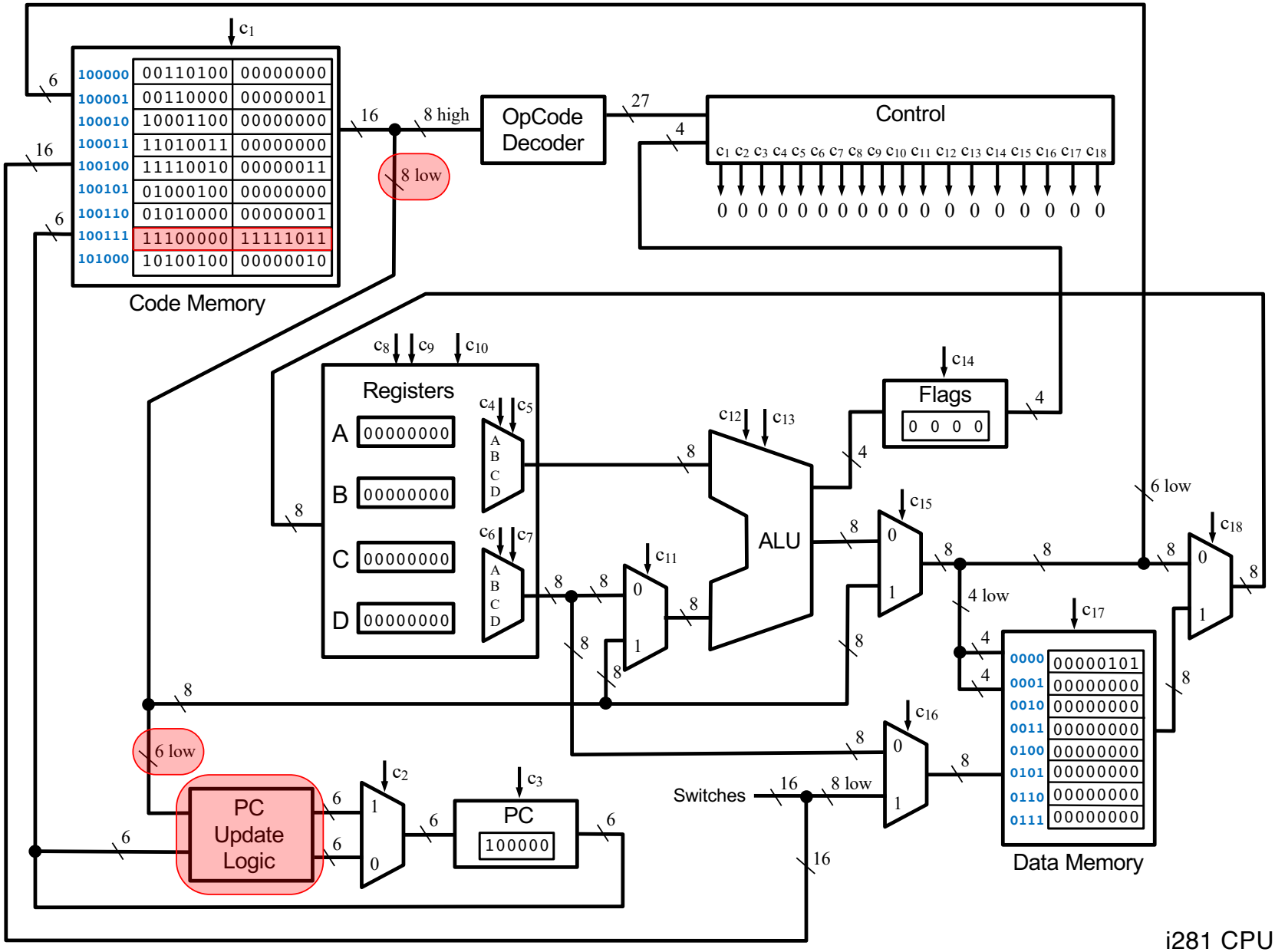
JUMP Label

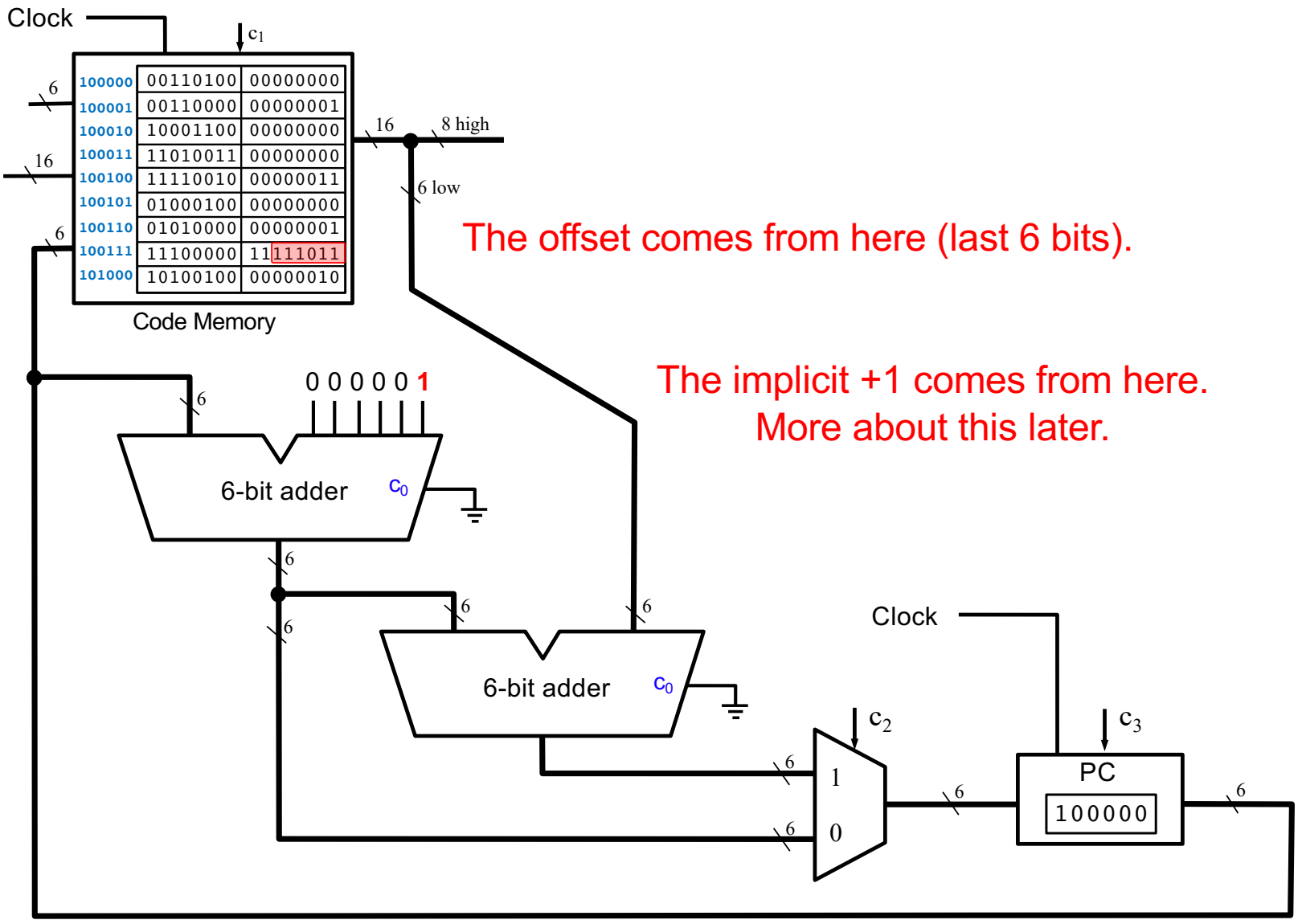


Because the code memory of the i281 CPU has space for only 64 instructions, however, the hardware uses only the last 6 bits. That is, it uses 6-bit adders to compute addresses.

Therefore, the possible range of offset values ranges from -32 to +31. However, due to an implicit +1 offset implemented by the hardware in the PC update logic, the actual effective range is -31 to +32.

The compiler emits 6-bit numbers that are then sign extended to 8-bit.


c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
6
16
6
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
8
c12 c13
ALU
c14
Flags
0 0 0 0
4
c15
c11
c16
c17
c18
6 low
4 low
8 low
Switches
16
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
c2
c3
PC Update Logic
PC
100000
i281 CPU


Clock
c1
6
16
6
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
16
8 high
6 low
Code Memory
The offset comes from here (last 6 bits).
The implicit +1 comes from here.
More about this later.
0 0 0 0 0 1
6-bit adder
c0
6-bit adder
c0
Clock
c2
c3
1
0
PC
100000

Possible Offsets with +1 Correction

0	11111111
1	00000000
2	00000001
3	00000010
4	00000011
5	00000100
6	00000101
7	00000110
8	00000111
9	00001000
10	00001001
11	00001010
12	00001011
13	00001100
14	00001101
15	00001110
16	00001111
17	00010000
18	00010001
19	00010010
20	00010011
21	00010100
22	00010101
23	00010110
24	00010111
25	00011000
26	00011001
27	00011010
28	00011011
29	00011100
30	00011101
31	00011110
32	00011111

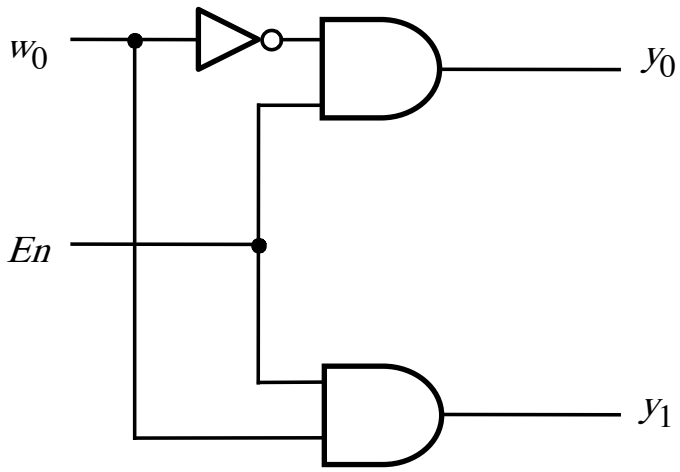
-1	11111110
-2	11111101
-3	11111100
-4	11111011
-5	11111010
-6	11111001
-7	11111000
-8	11110111
-9	11110110
-10	11110101
-11	11110100
-12	11110011
-13	11110010
-14	11110001
-15	11110000
-16	11101111
-17	11101110
-18	11101101
-19	11101100
-20	11101011
-21	11101010
-22	11101001
-23	11101000
-24	11100111
-25	11100110
-26	11100101
-27	11100100
-28	11100011
-29	11100010
-30	11100001
-31	11100000
-32	N/A

The i281 Assembly Instructions

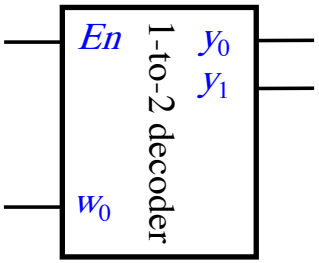
NOOP	NO OPeration
INPUTC	INPUT into Code memory
INPUTCF	INPUT into Code memory with oFfset
INPUTD	INPUT into Data memory
INPUTDF	INPUT into Data memory with oFfset
MOVE	MOVE the contents of one register into another
LOADI	LOAD Immediate value
LOADP	LOAD Pointer address
ADD	ADD two registers
ADDI	ADD an Immediate value to a register
SUB	SUBtract two registers
SUBI	SUBtract an Immediate value from a register
LOAD	LOAD from a data memory address into a register
LOADF	LOAD with an oFfset specified by another register
STORE	STORE a register into a data memory address
STOREF	STORE with an oFfset specified by another register
SHIFTL	SHIFT Left all bits in a register
SHIFTR	SHIFT Right all bits in a register
CMP	CoMPare the values in two registers
JUMP	JUMP unconditionally to a specified address
BRE	BRanch if Equal
BRZ	BRanch if Zero
BRNE	BRanch if Not Equal
BRNZ	BRanch if Not Zero
BRG	BRanch if Greater
BRGE	BRanch if Greater than or Equal

Quick Review:
Decoders

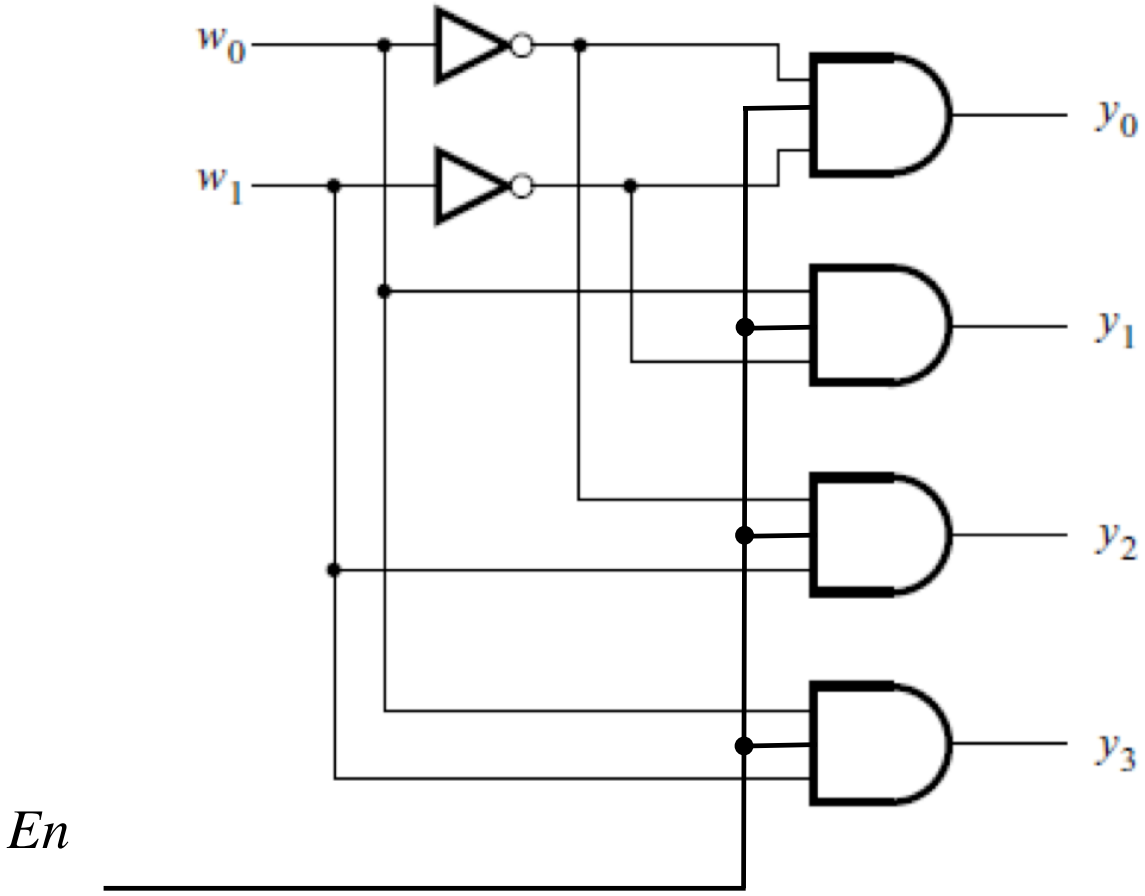
1-to-2 decoder



1-to-2 decoder

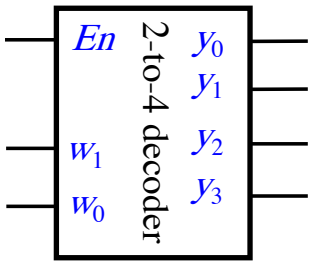


2-to-4 decoder

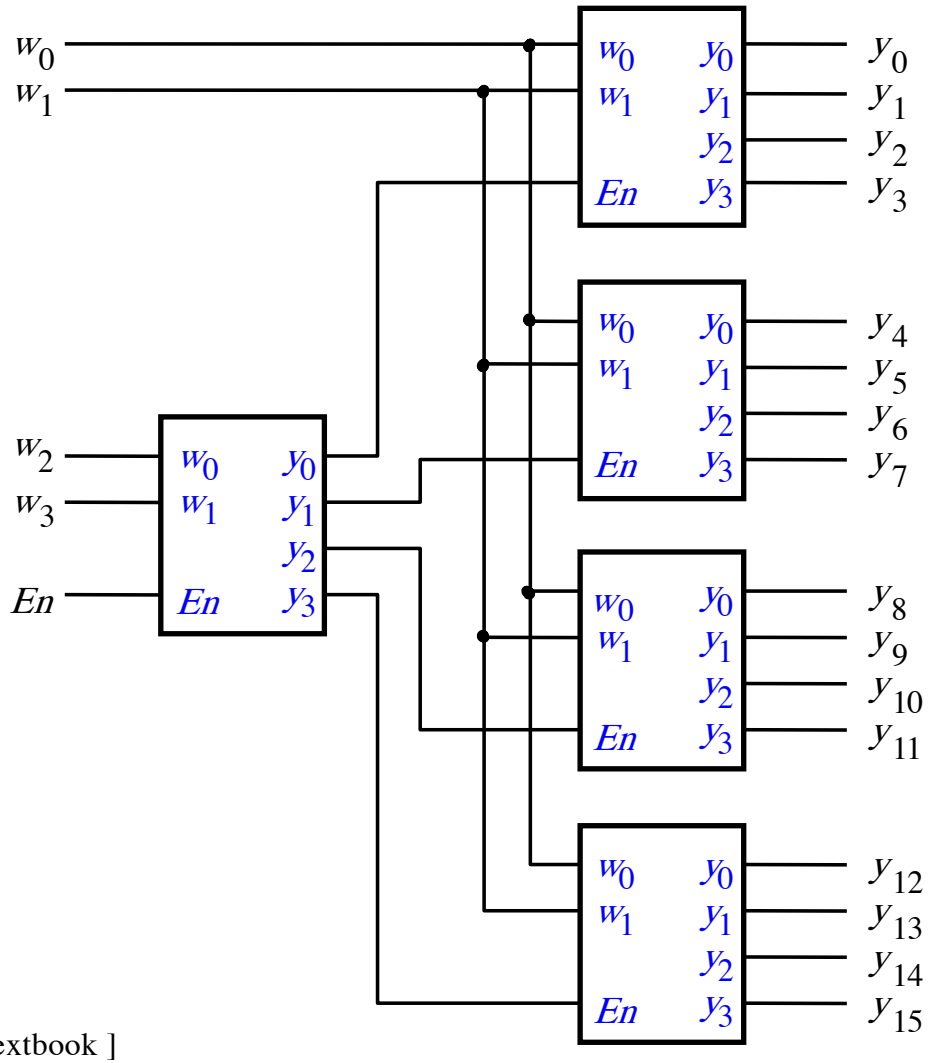


[Figure 4.14c from the textbook]

2-to-4 decoder

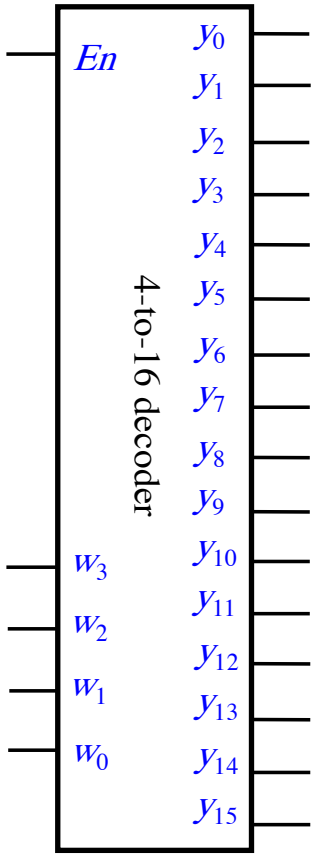


4-to-16 decoder built using a decoder tree

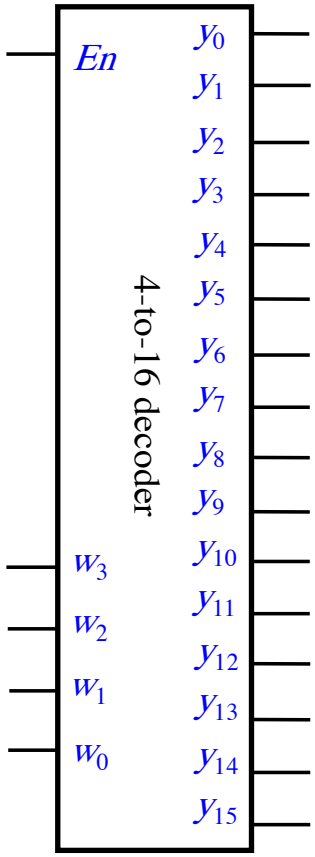


[Figure 4.16 from the textbook]

4-to-16 decoder

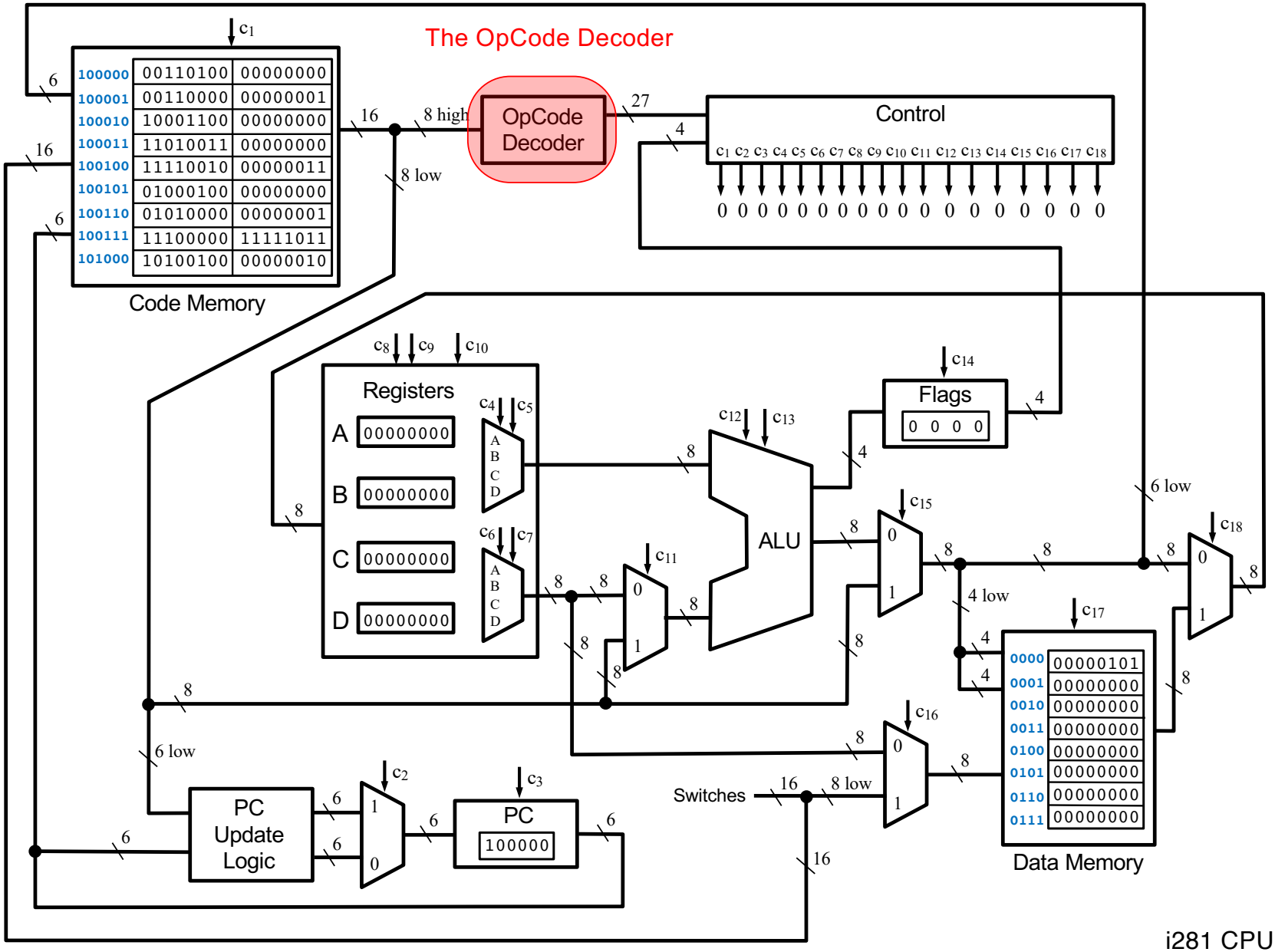


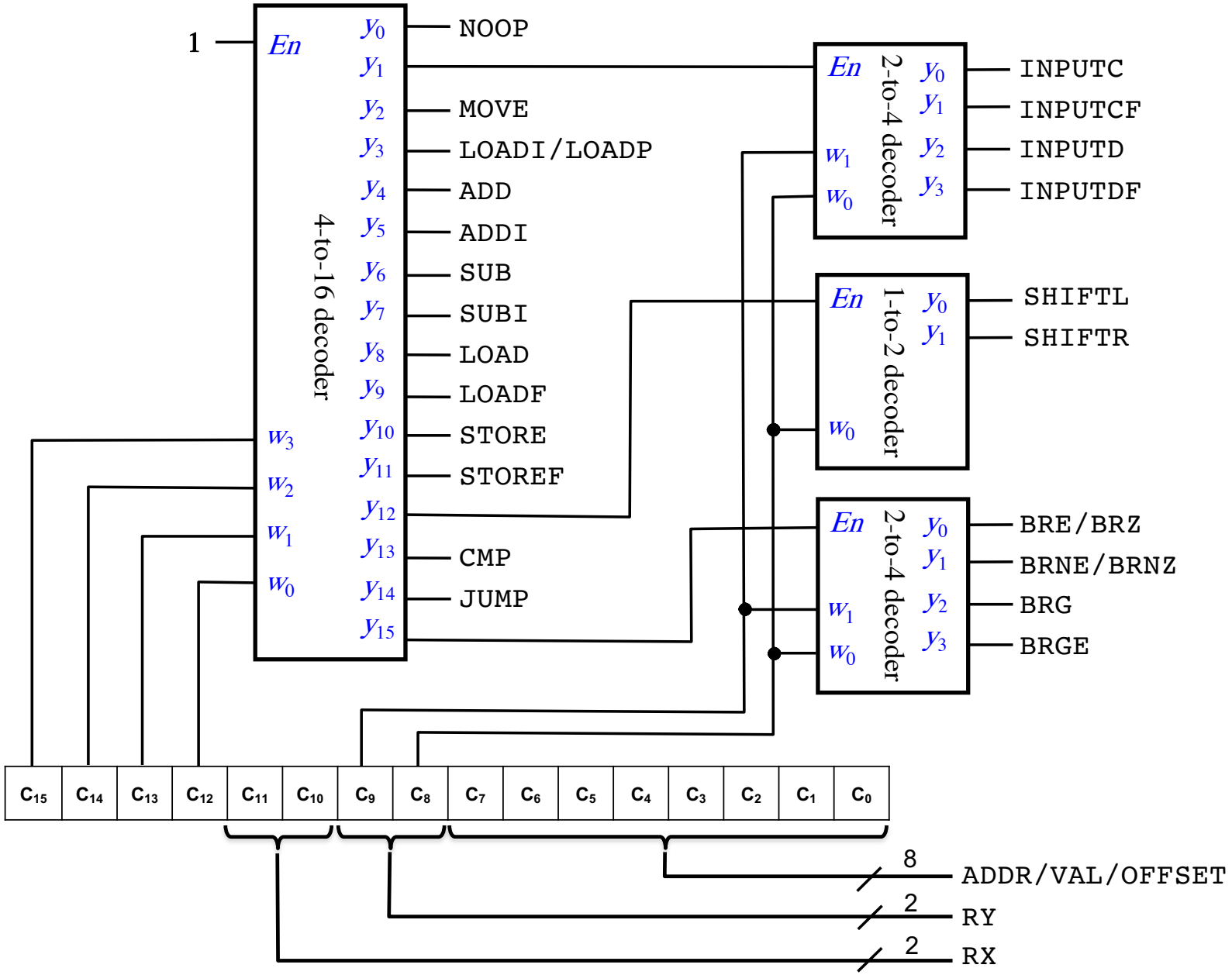
4-to-16 decoder

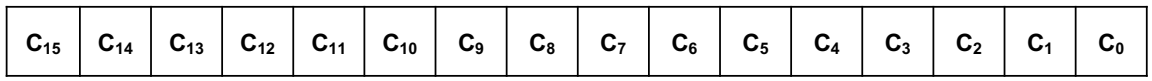


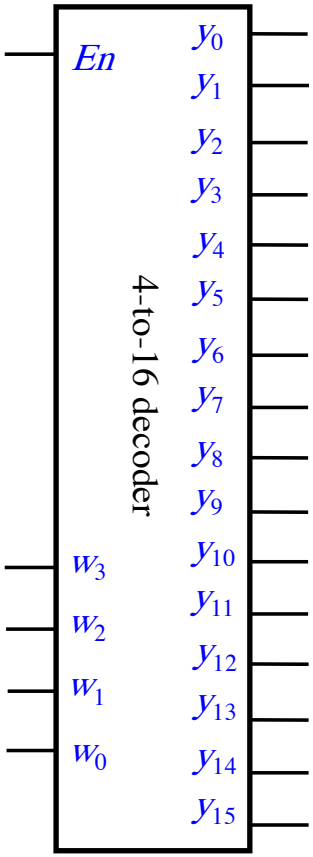
The outputs are one-hot encoded when En=1

OPCODE Decoding Circuit

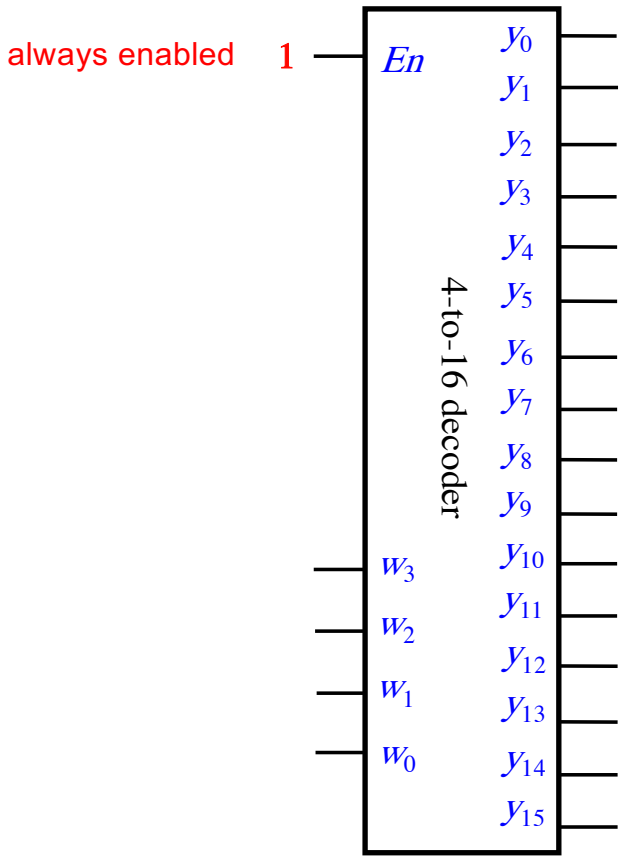

The OpCode Decoder
c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
16
8 high
8 low
OpCode Decoder
27
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
c11
c12 c13
ALU
c14
Flags
0 0 0 0
c15
c16
c17
c18
6 low
4 low
8 low
Switches
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
PC Update Logic
c2
c3
PC
100000
i281 CPU


1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 NOOP
y_1
y_2 MOVE
y_3 LOADI/LOADP
y_4 ADD
y_5 ADDI
y_6 SUB
y_7 SUBI
y_8 LOAD
y_9 LOADF
y_{10} STORE
y_{11} STOREF
y_{12}
y_{13} CMP
y_{14} JUMP
y_{15}
En
2-to-4 decoder
w_1
w_0
y_0 INPUTC
y_1 INPUTCF
y_2 INPUTD
y_3 INPUTDF
En
1-to-2 decoder
w_0
y_0 SHIFTL
y_1 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 BRE/BRZ
y_1 BRNE/BRNZ
y_2 BRG
y_3 BRGE
C_{15} C_{14} C_{13} C_{12} C_{11} C_{10} C_9 C_8 C_7 C_6 C_5 C_4 C_3 C_2 C_1 C_0
8
ADDR/VAL/OFFSET
2
RY
2
RX

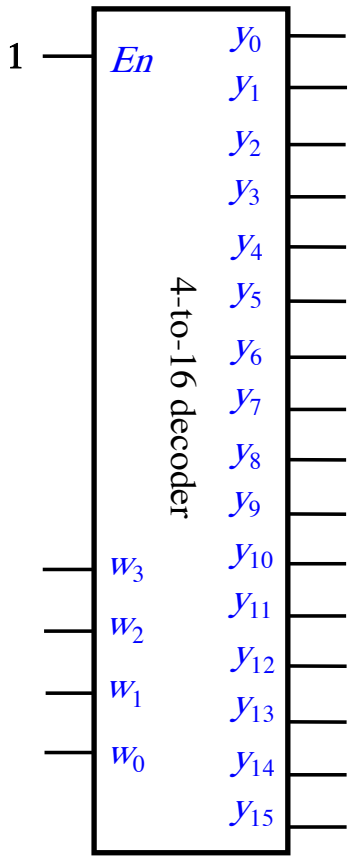

C15
C14
C13
C12
C11
C10
C9
C8
C7
C6
C5
C4
C3
C2
C1
C0



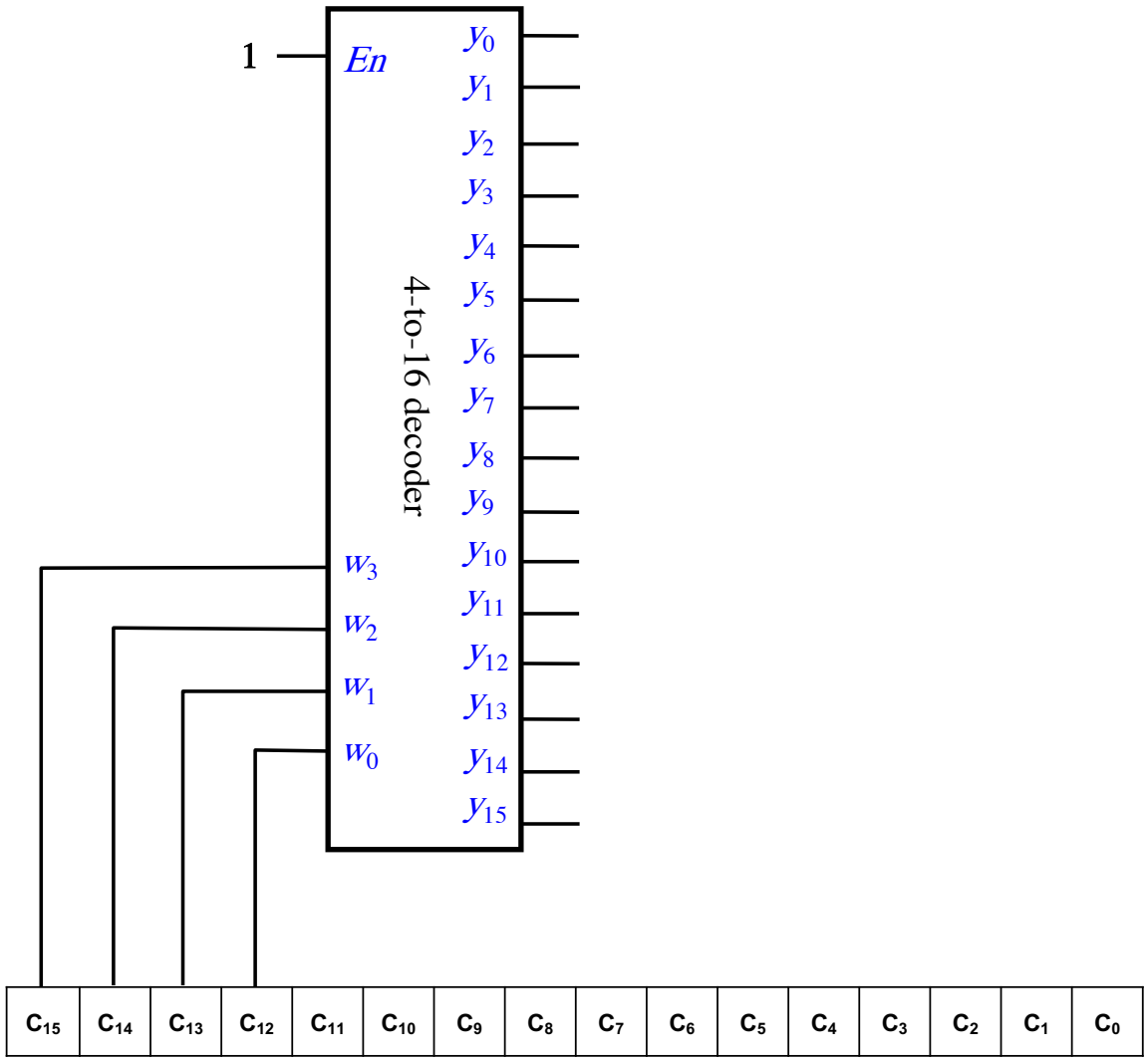
C_{15}	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0

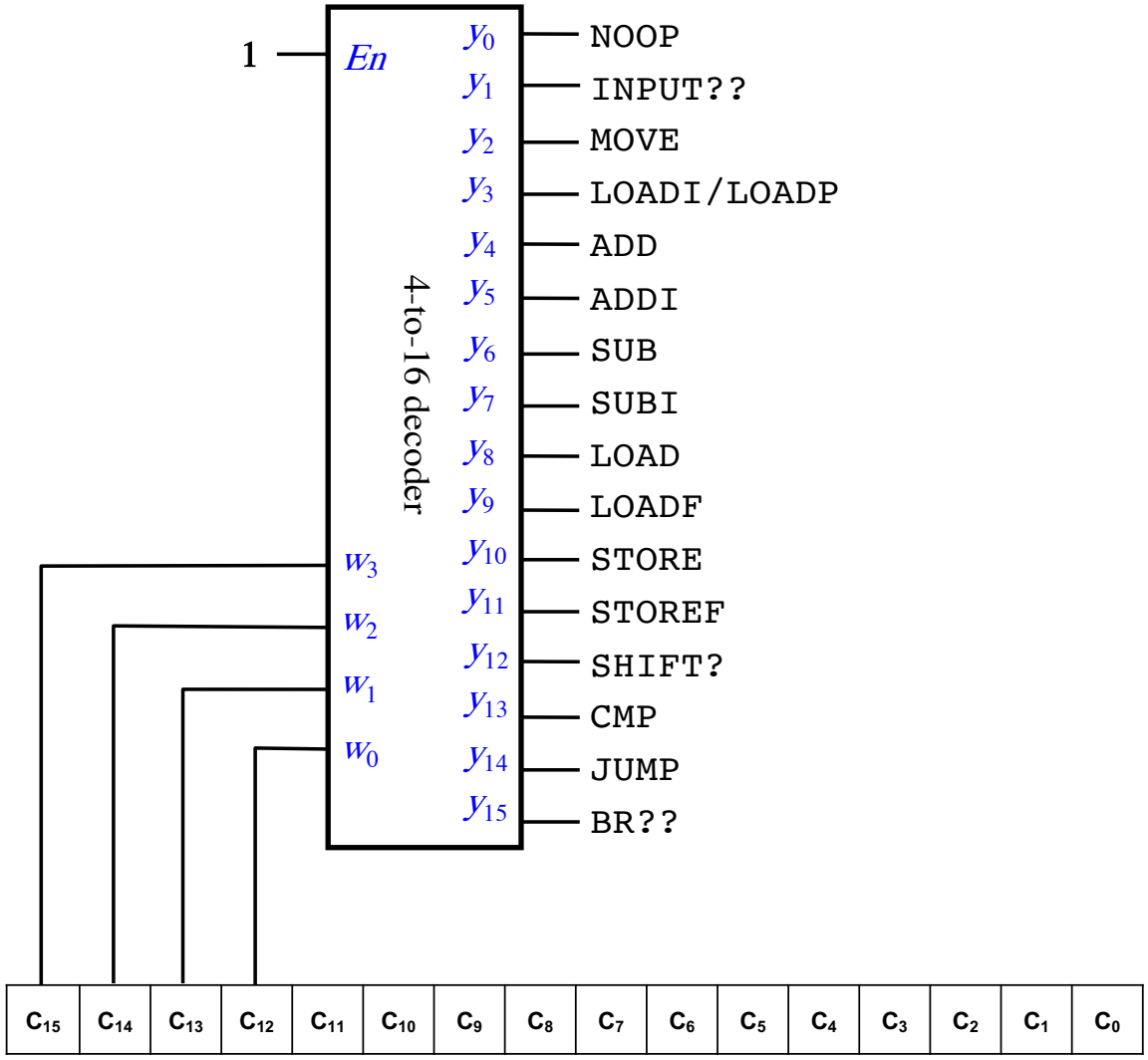


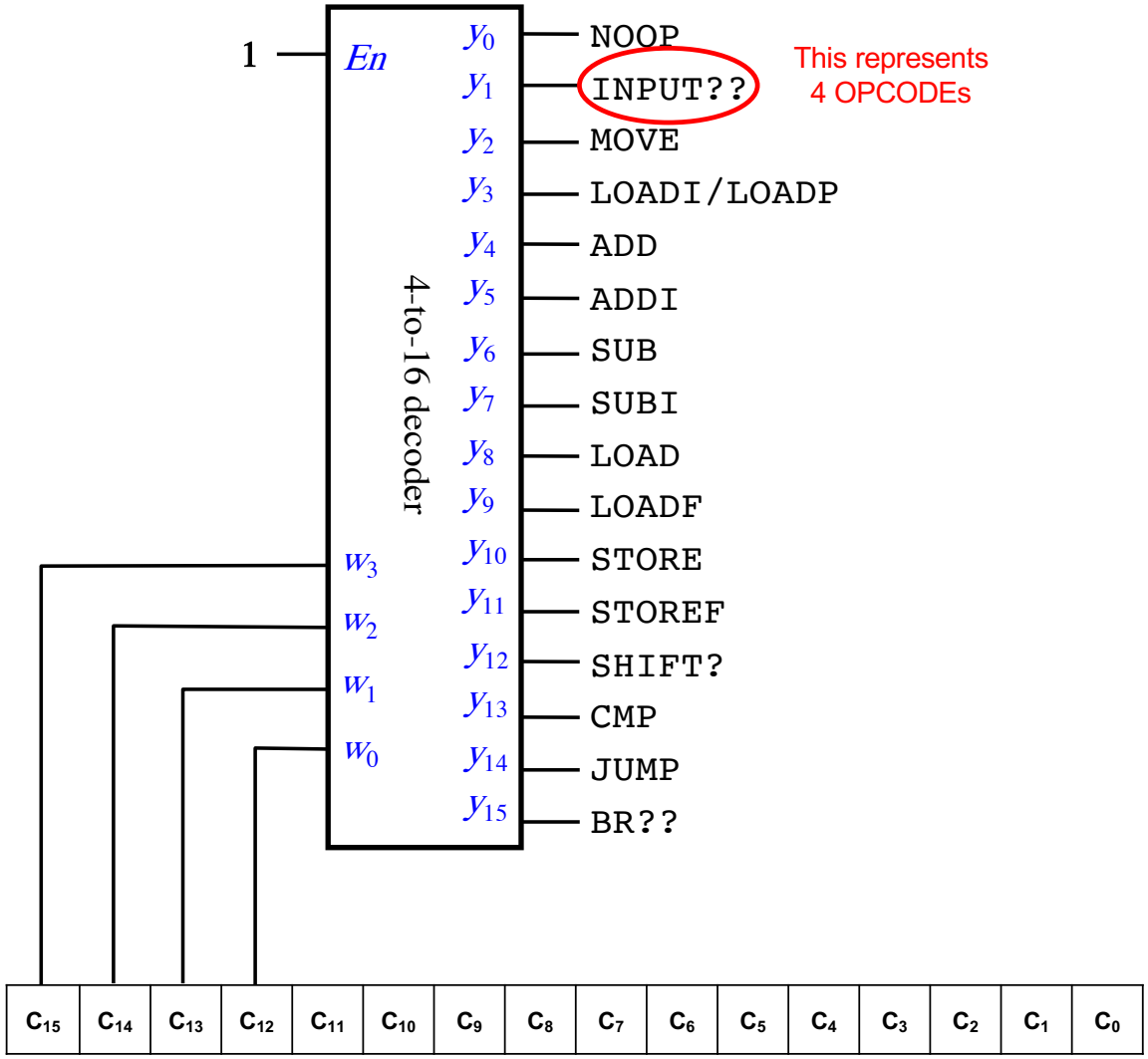
C_{15}	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0

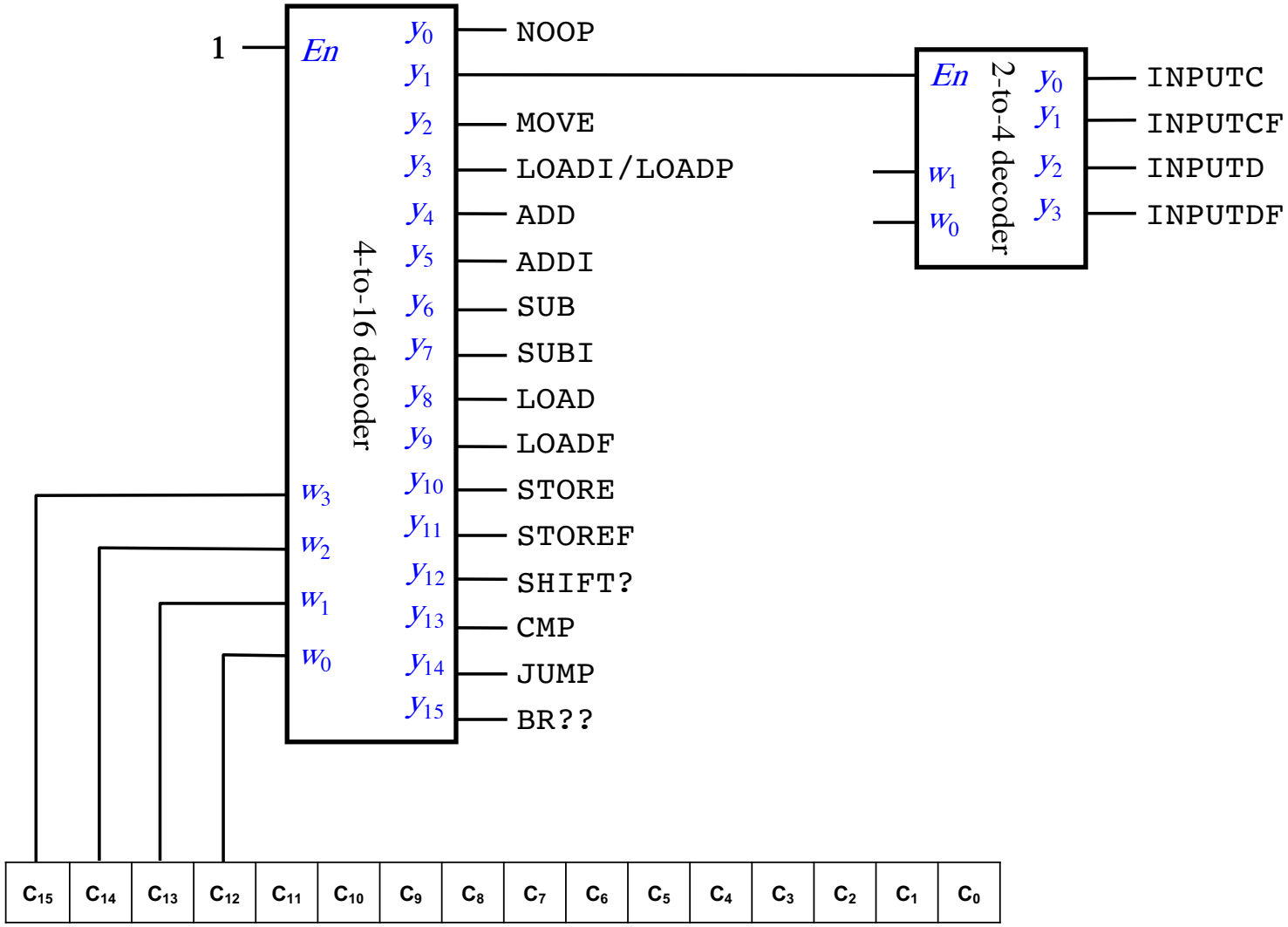


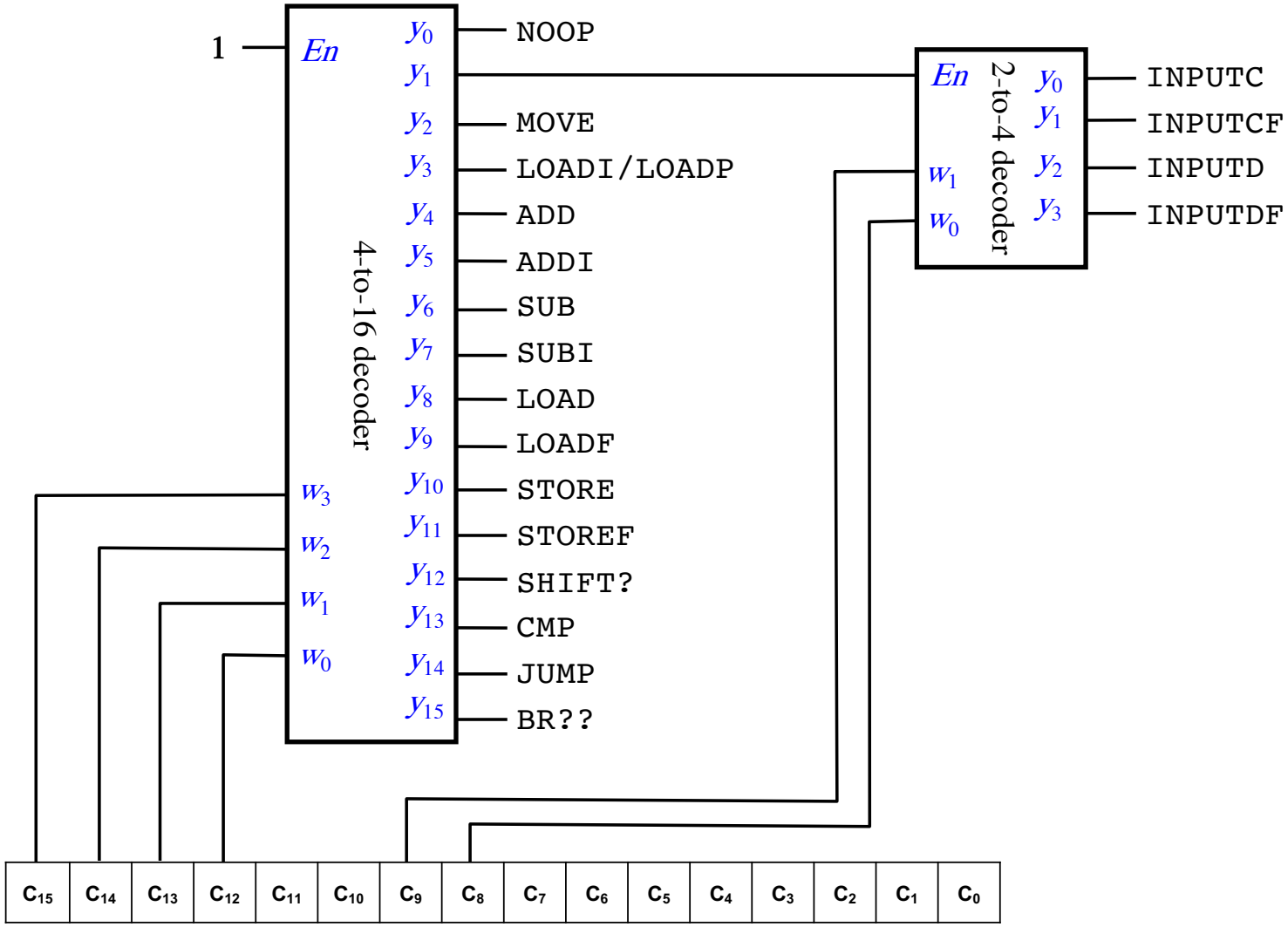
C_{15}	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0

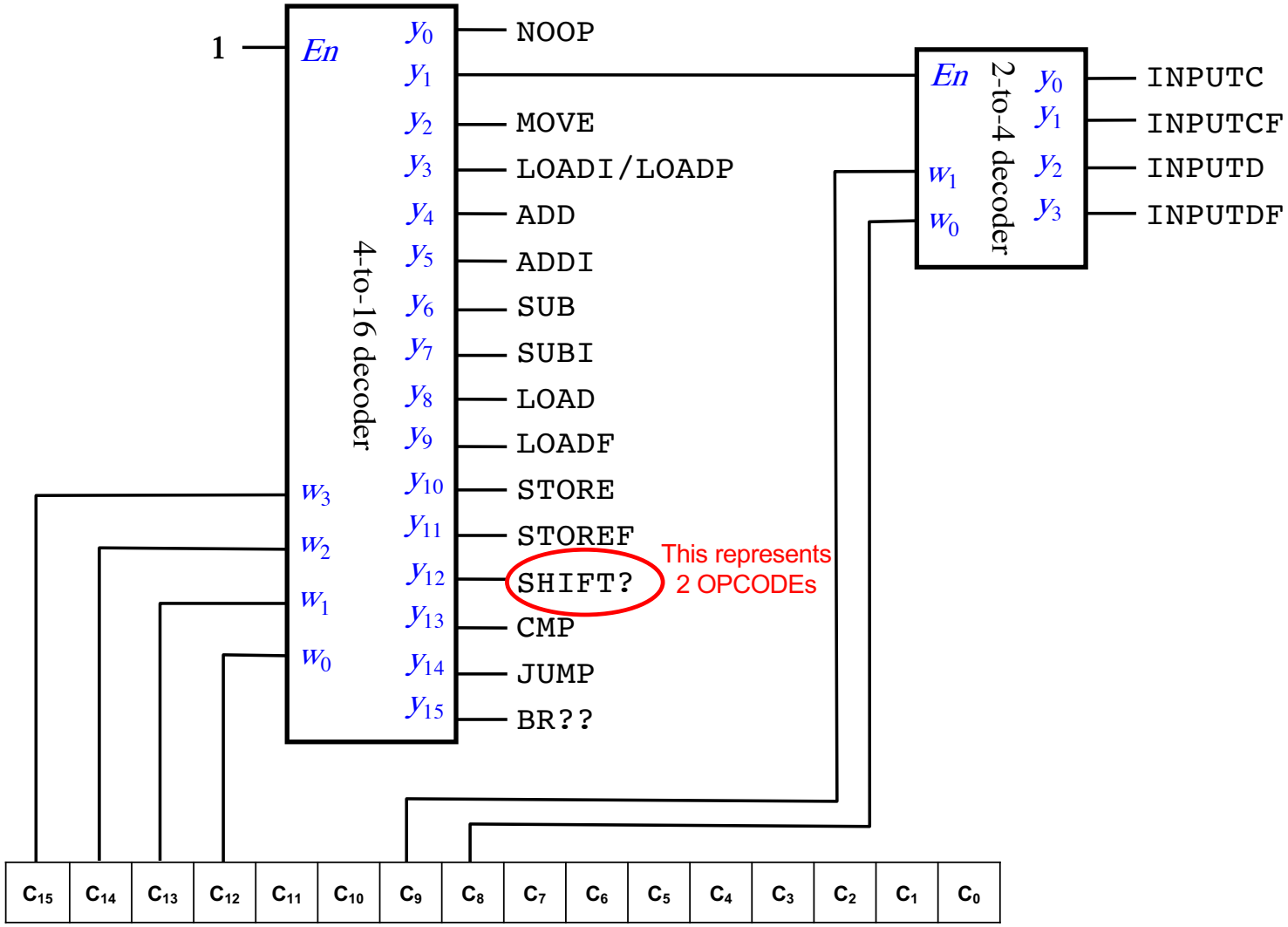

1
En
4-to-16 decoder
y_0
y_1
y_2
y_3
y_4
y_5
y_6
y_7
y_8
y_9
y_{10}
y_{11}
y_{12}
y_{13}
y_{14}
y_{15}
w_3
w_2
w_1
w_0
C15 C14 C13 C12 C11 C10 C9 C8 C7 C6 C5 C4 C3 C2 C1 C0

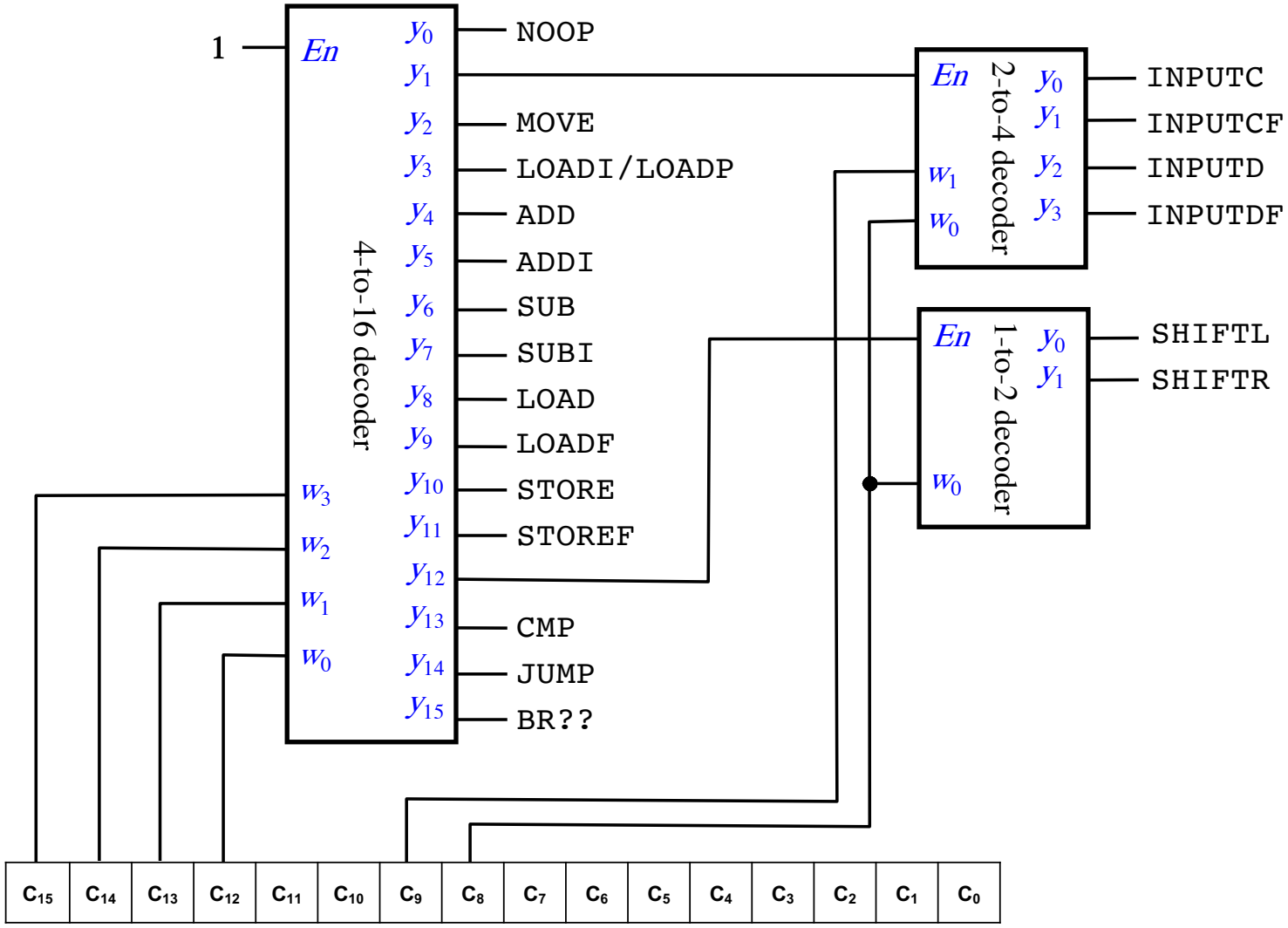

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 NOOP
y_1 INPUT??
y_2 MOVE
y_3 LOADI/LOADP
y_4 ADD
y_5 ADDI
y_6 SUB
y_7 SUBI
y_8 LOAD
y_9 LOADF
y_{10} STORE
y_{11} STOREF
y_{12} SHIFT?
y_{13} CMP
y_{14} JUMP
y_{15} BR??
C15 C14 C13 C12 C11 C10 C9 C8 C7 C6 C5 C4 C3 C2 C1 C0

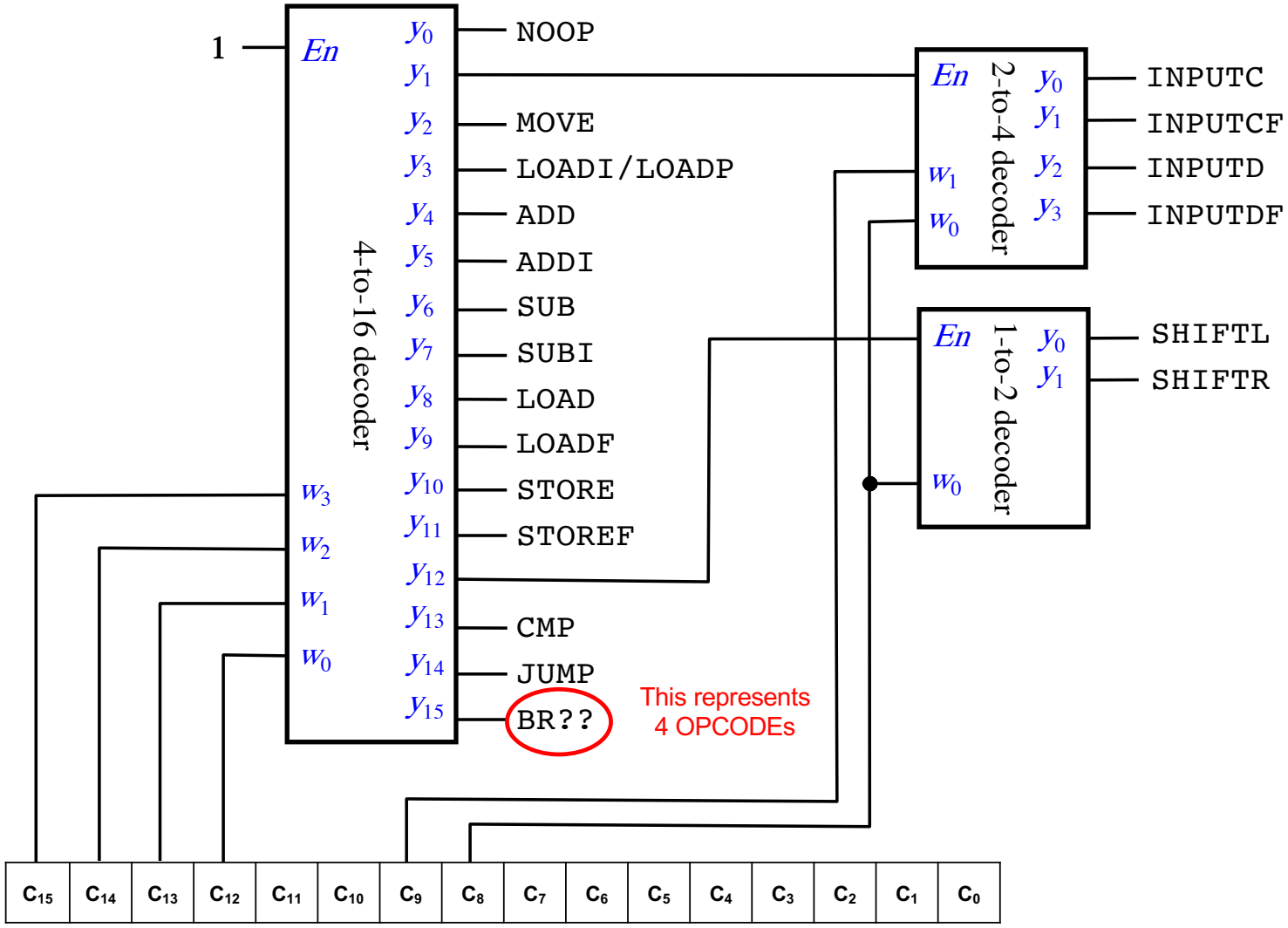

1
En
4-to-16 decoder
y_0 NOOP
y_1 INPUT??
y_2 MOVE
y_3 LOADI/LOADP
y_4 ADD
y_5 ADDI
y_6 SUB
y_7 SUBI
y_8 LOAD
y_9 LOADF
y_{10} STORE
y_{11} STOREF
y_{12} SHIFT?
y_{13} CMP
y_{14} JUMP
y_{15} BR??
w_3
w_2
w_1
w_0
This represents
4 OPCODEs
C_{15} C_{14} C_{13} C_{12} C_{11} C_{10} C_9 C_8 C_7 C_6 C_5 C_4 C_3 C_2 C_1 C_0

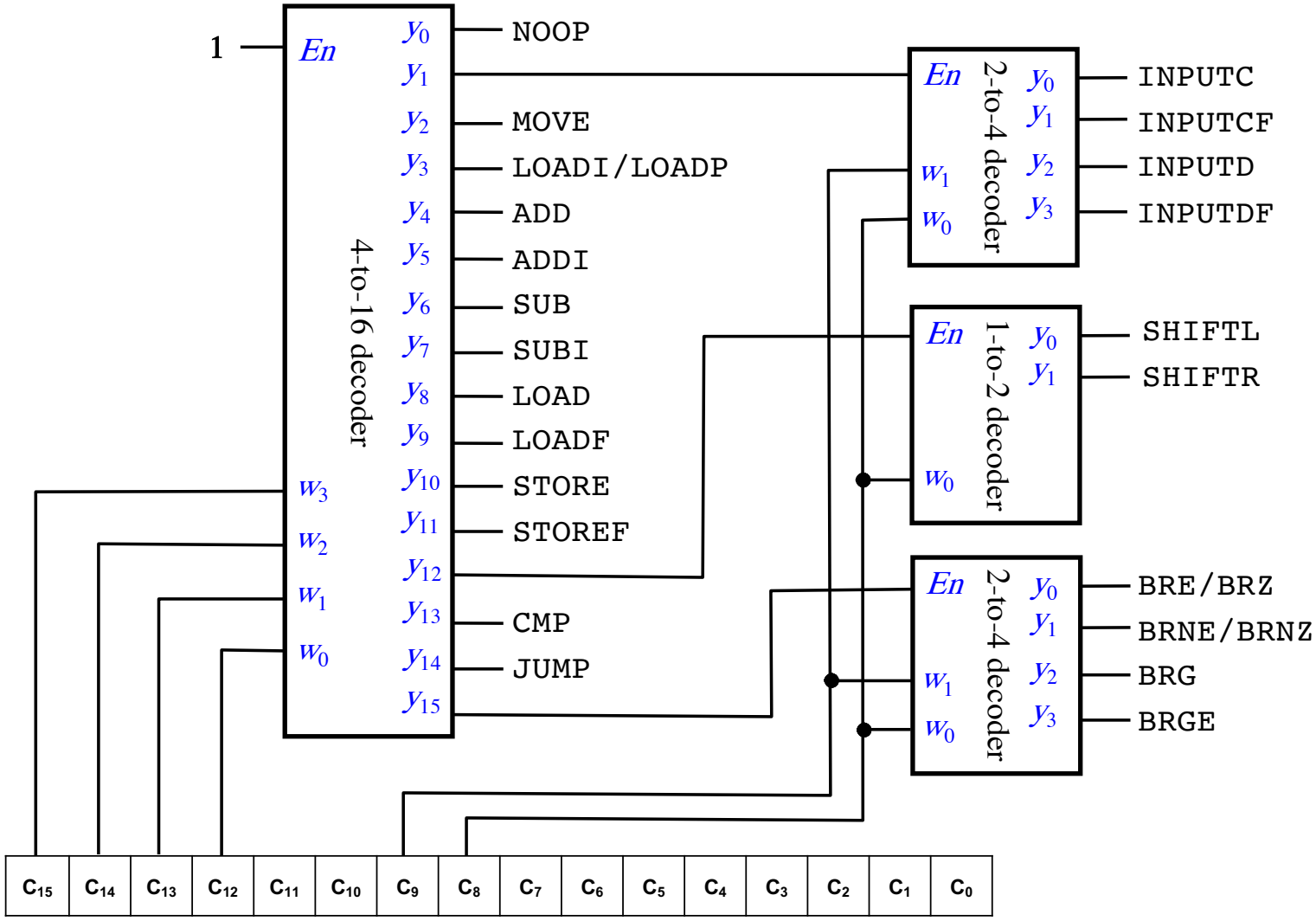

1
En
4-to-16 decoder
y_0
y_1
y_2
y_3
y_4
y_5
y_6
y_7
y_8
y_9
y_{10}
y_{11}
y_{12}
y_{13}
y_{14}
y_{15}
NOOP
MOVE
LOADI/LOADP
ADD
ADDI
SUB
SUBI
LOAD
LOADF
STORE
STOREF
SHIFT?
CMP
JUMP
BR??
w_3
w_2
w_1
w_0
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
INPUTC
INPUTCF
INPUTD
INPUTDF
C_{15} C_{14} C_{13} C_{12} C_{11} C_{10} C_9 C_8 C_7 C_6 C_5 C_4 C_3 C_2 C_1 C_0

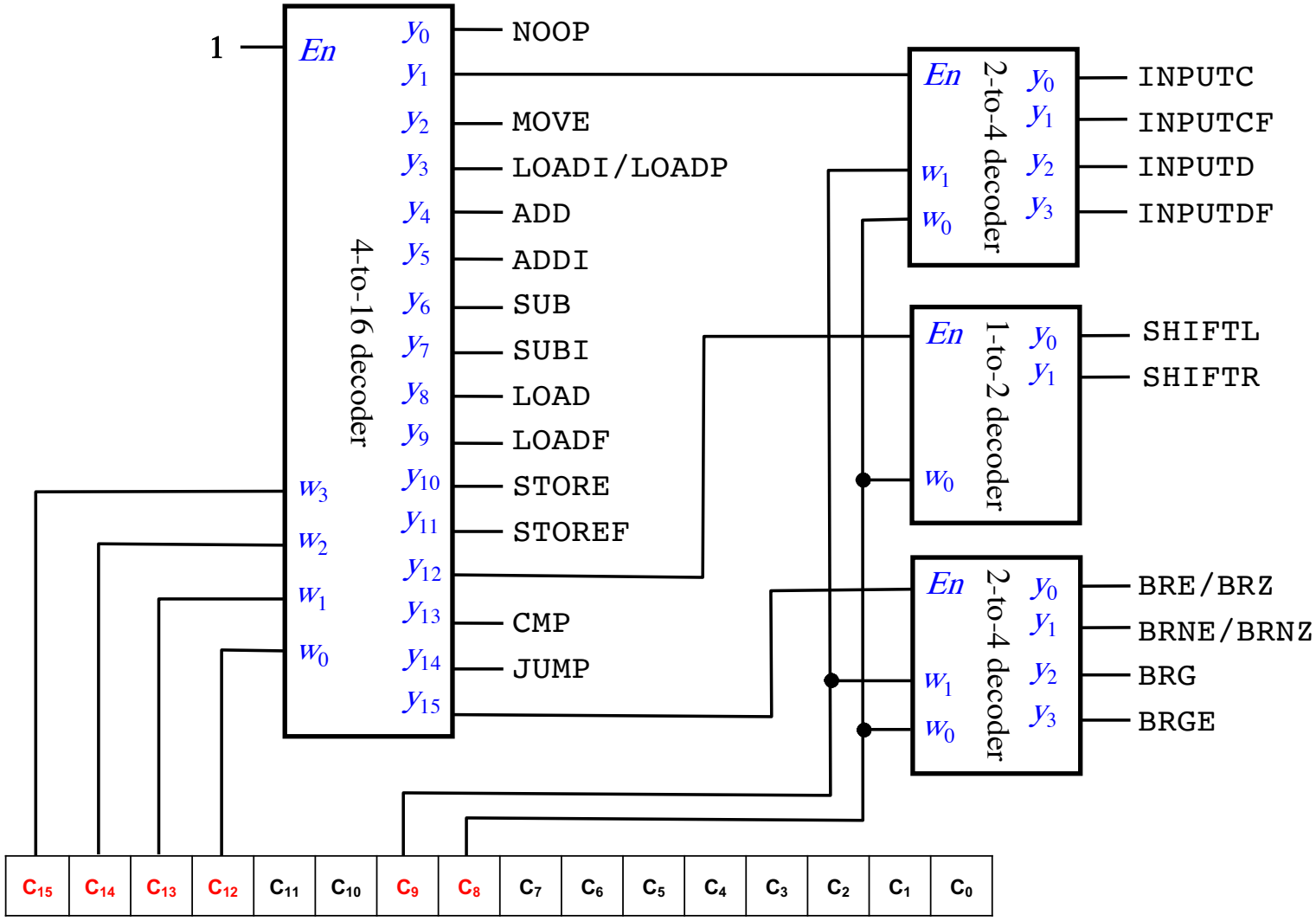

1
En
4-to-16 decoder
y_0
y_1
y_2
y_3
y_4
y_5
y_6
y_7
y_8
y_9
y_{10}
y_{11}
y_{12}
y_{13}
y_{14}
y_{15}
NOOP
MOVE
LOADI/LOADP
ADD
ADDI
SUB
SUBI
LOAD
LOADF
STORE
STOREF
SHIFT?
CMP
JUMP
BR??
w_3
w_2
w_1
w_0
En
2-to-4 decoder
y_0
y_1
y_2
y_3
INPUTC
INPUTCF
INPUTD
INPUTDF
w_1
w_0
C_{15} C_{14} C_{13} C_{12} C_{11} C_{10} C_9 C_8 C_7 C_6 C_5 C_4 C_3 C_2 C_1 C_0


1
En
4-to-16 decoder
y_0 NOOP
y_1
y_2 MOVE
y_3 LOADI/LOADP
y_4 ADD
y_5 ADDI
y_6 SUB
y_7 SUBI
y_8 LOAD
y_9 LOADF
y_{10} STORE
y_{11} STOREF
y_{12} SHIFT?
y_{13} CMP
y_{14} JUMP
y_{15} BR??
w_3
w_2
w_1
w_0
This represents
2 OPCODEs
En
2-to-4 decoder
y_0 INPUTC
y_1 INPUTCF
y_2 INPUTD
y_3 INPUTDF
w_1
w_0
C15 C14 C13 C12 C11 C10 C9 C8 C7 C6 C5 C4 C3 C2 C1 C0

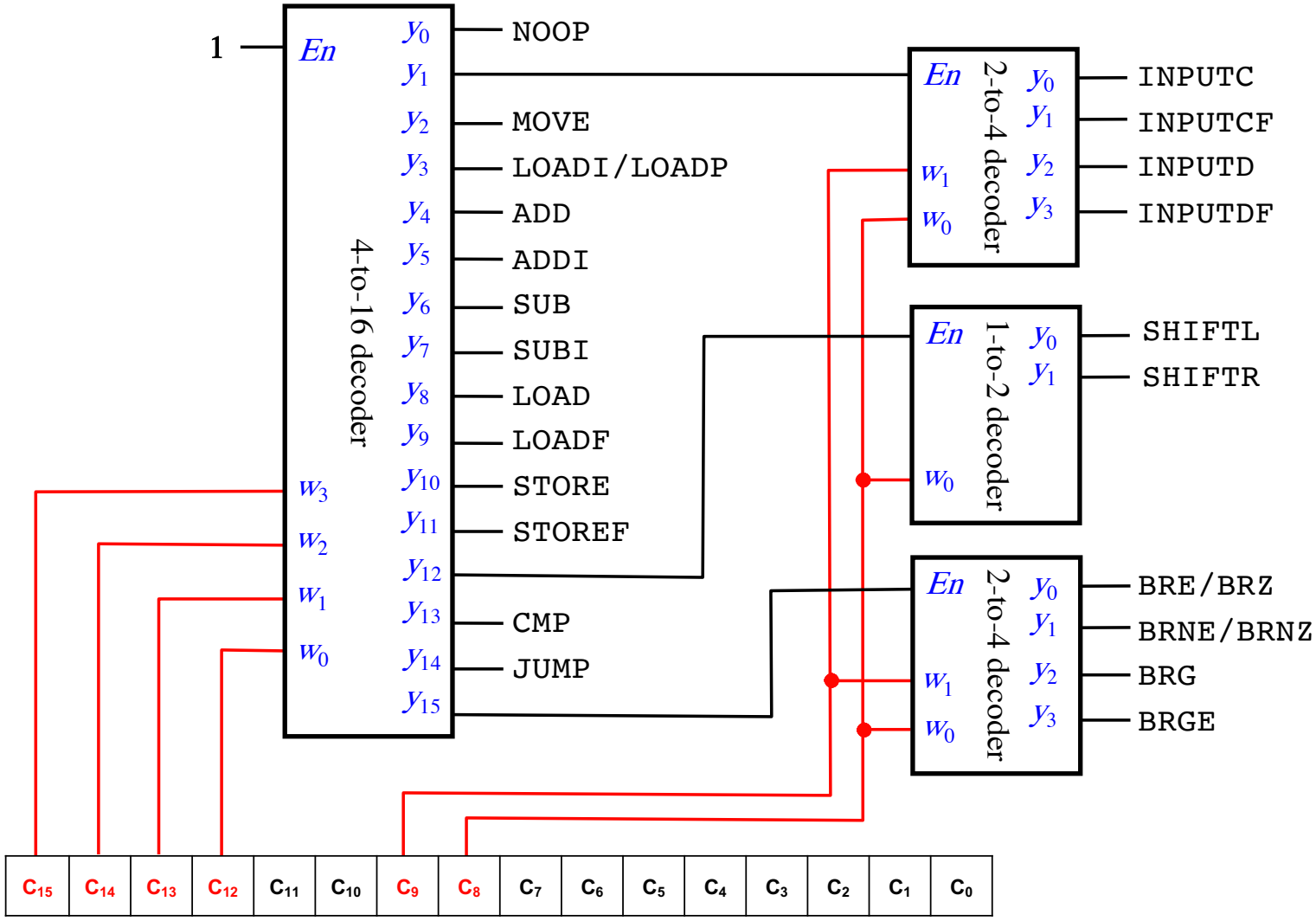

1
En
4-to-16 decoder
y_0
y_1
y_2
y_3
y_4
y_5
y_6
y_7
y_8
y_9
y_{10}
y_{11}
y_{12}
y_{13}
y_{14}
y_{15}
NOOP
MOVE
LOADI/LOADP
ADD
ADDI
SUB
SUBI
LOAD
LOADF
STORE
STOREF
CMP
JUMP
BR??
w_3
w_2
w_1
w_0
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
INPUTC
INPUTCF
INPUTD
INPUTDF
En
1-to-2 decoder
y_0
y_1
w_0
SHIFTL
SHIFTR
C_{15} C_{14} C_{13} C_{12} C_{11} C_{10} C_9 C_8 C_7 C_6 C_5 C_4 C_3 C_2 C_1 C_0


1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 NOOP
y_1
y_2 MOVE
y_3 LOADI/LOADP
y_4 ADD
y_5 ADDI
y_6 SUB
y_7 SUBI
y_8 LOAD
y_9 LOADF
y_{10} STORE
y_{11} STOREF
y_{12}
y_{13} CMP
y_{14} JUMP
y_{15} BR??
This represents 4 OPCODEs
En
2-to-4 decoder
w_1
w_0
y_0 INPUTC
y_1 INPUTCF
y_2 INPUTD
y_3 INPUTDF
En
1-to-2 decoder
w_0
y_0 SHIFTL
y_1 SHIFTR
C15 C14 C13 C12 C11 C10 C9 C8 C7 C6 C5 C4 C3 C2 C1 C0

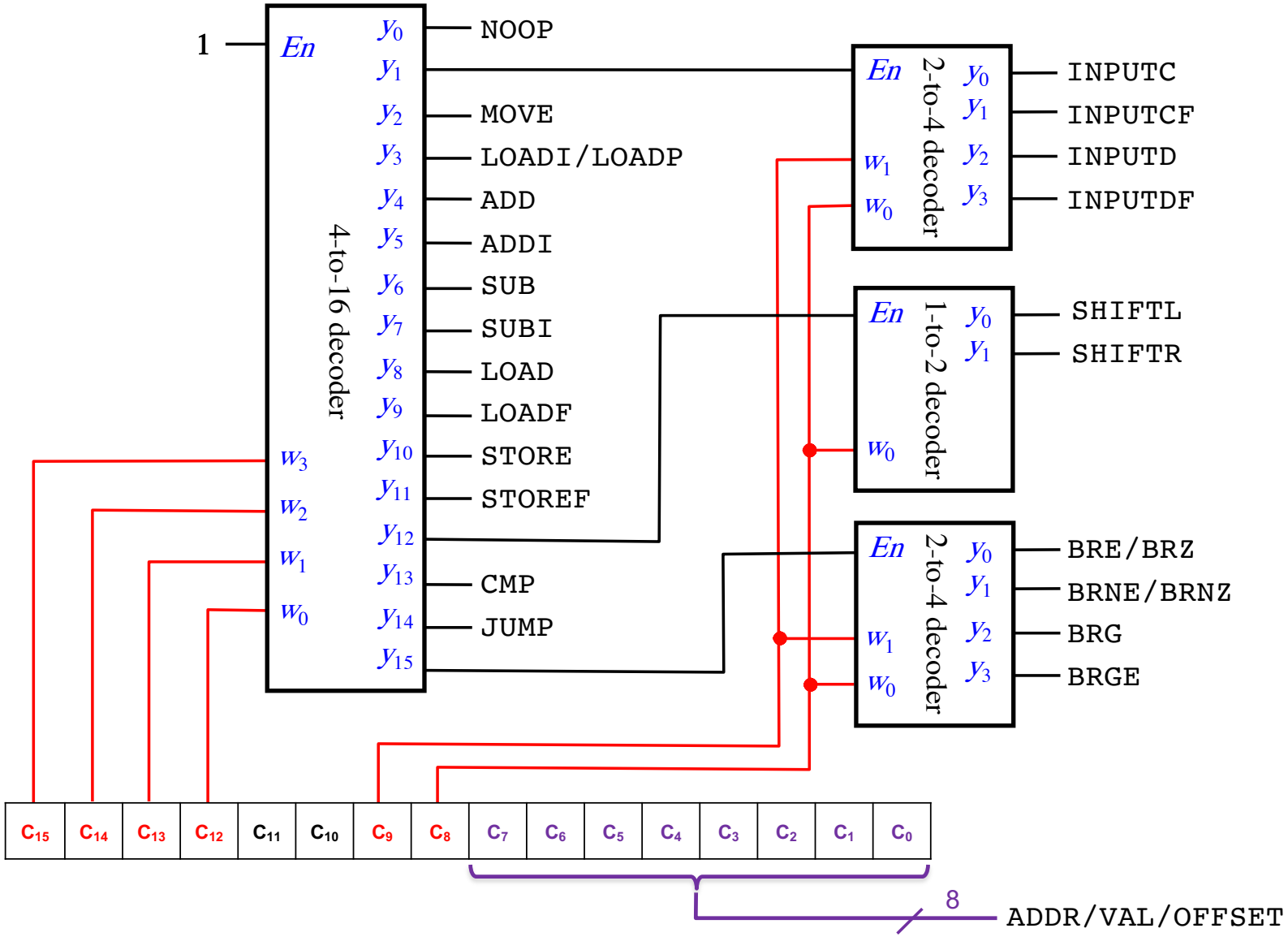

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0
y_1
y_2
y_3
y_4
y_5
y_6
y_7
y_8
y_9
y_{10}
y_{11}
y_{12}
y_{13}
y_{14}
y_{15}
NOOP
MOVE
LOADI/LOADP
ADD
ADDI
SUB
SUBI
LOAD
LOADF
STORE
STOREF
CMP
JUMP
En
2-to-4 decoder
w_1
w_0
y_0
y_1
y_2
y_3
INPUTC
INPUTCF
INPUTD
INPUTDF
En
1-to-2 decoder
w_0
y_0
y_1
SHIFTL
SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0
y_1
y_2
y_3
BRE/BRZ
BRNE/BRNZ
BRG
BRGE
C_{15} C_{14} C_{13} C_{12} C_{11} C_{10} C_9 C_8 C_7 C_6 C_5 C_4 C_3 C_2 C_1 C_0

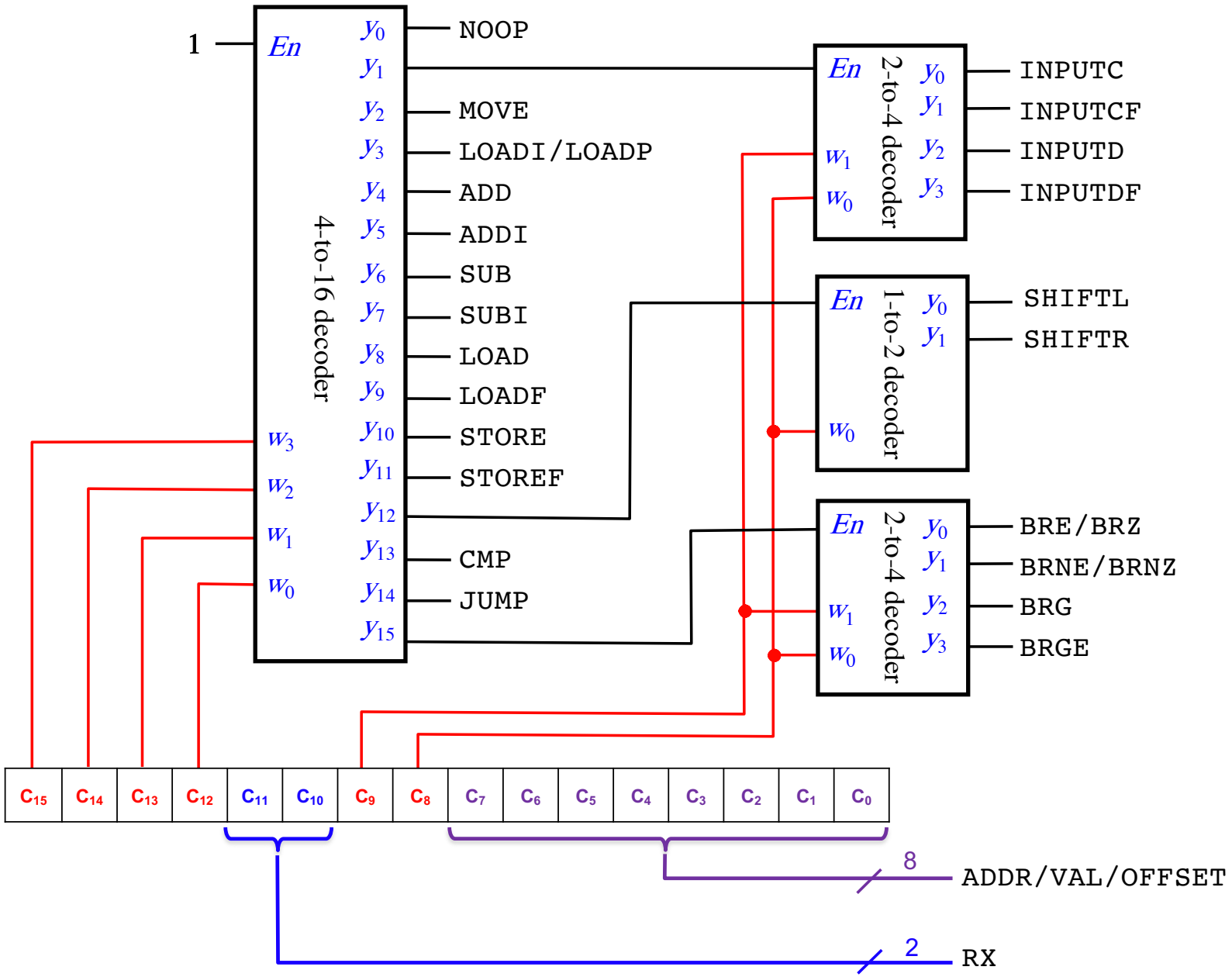


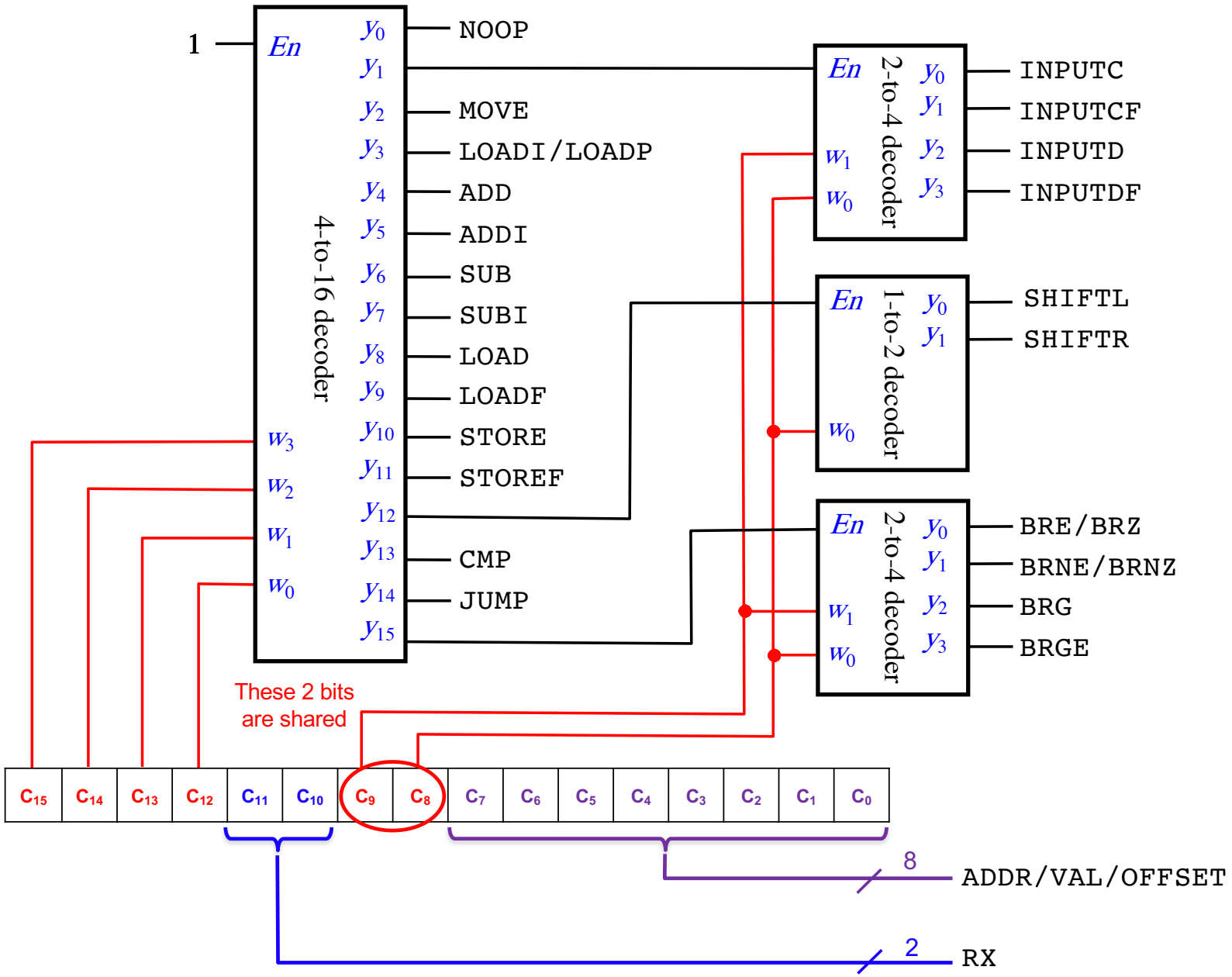
These 6 bits represent the OPCODEs

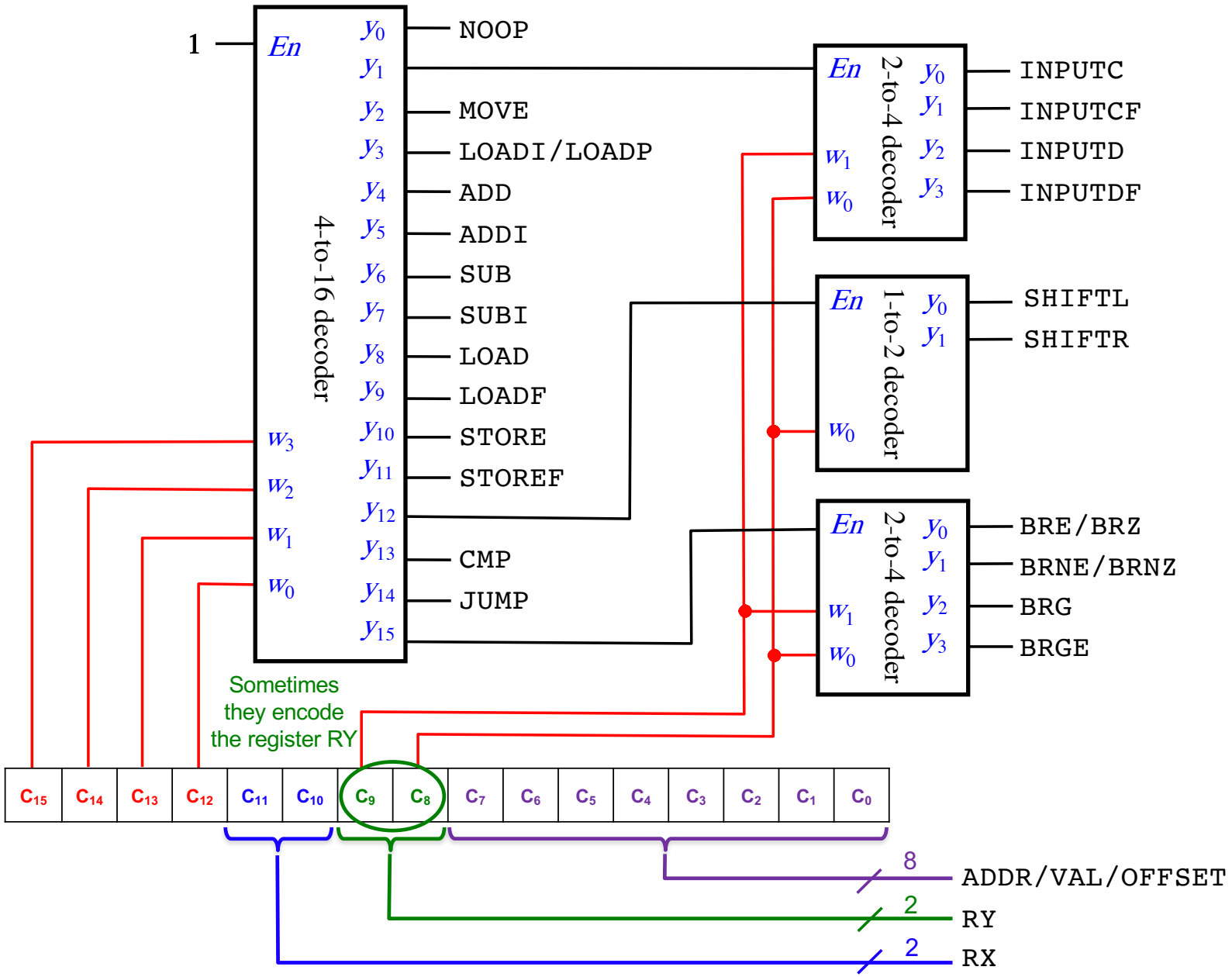


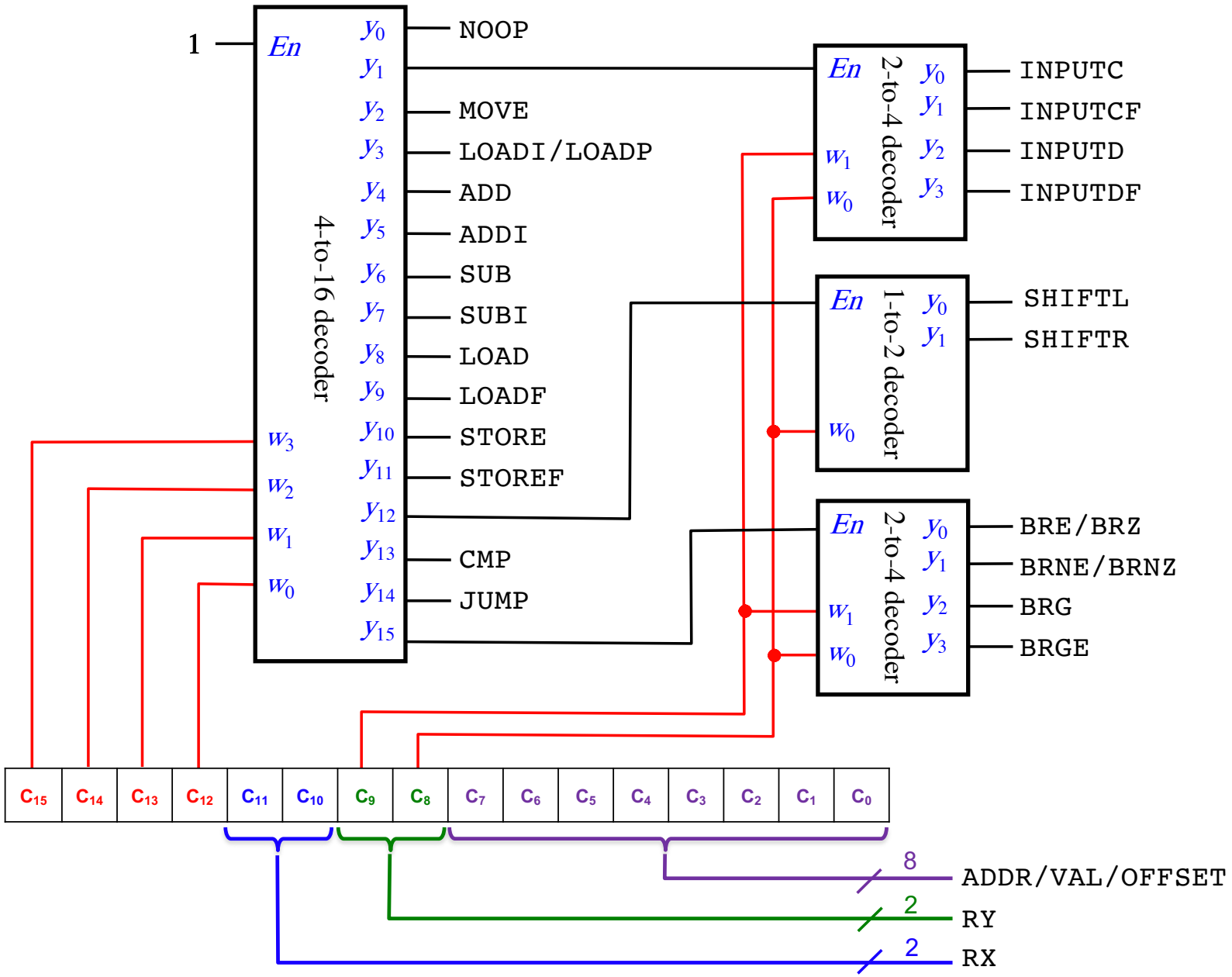
These 6 bits represent the OPCODEs

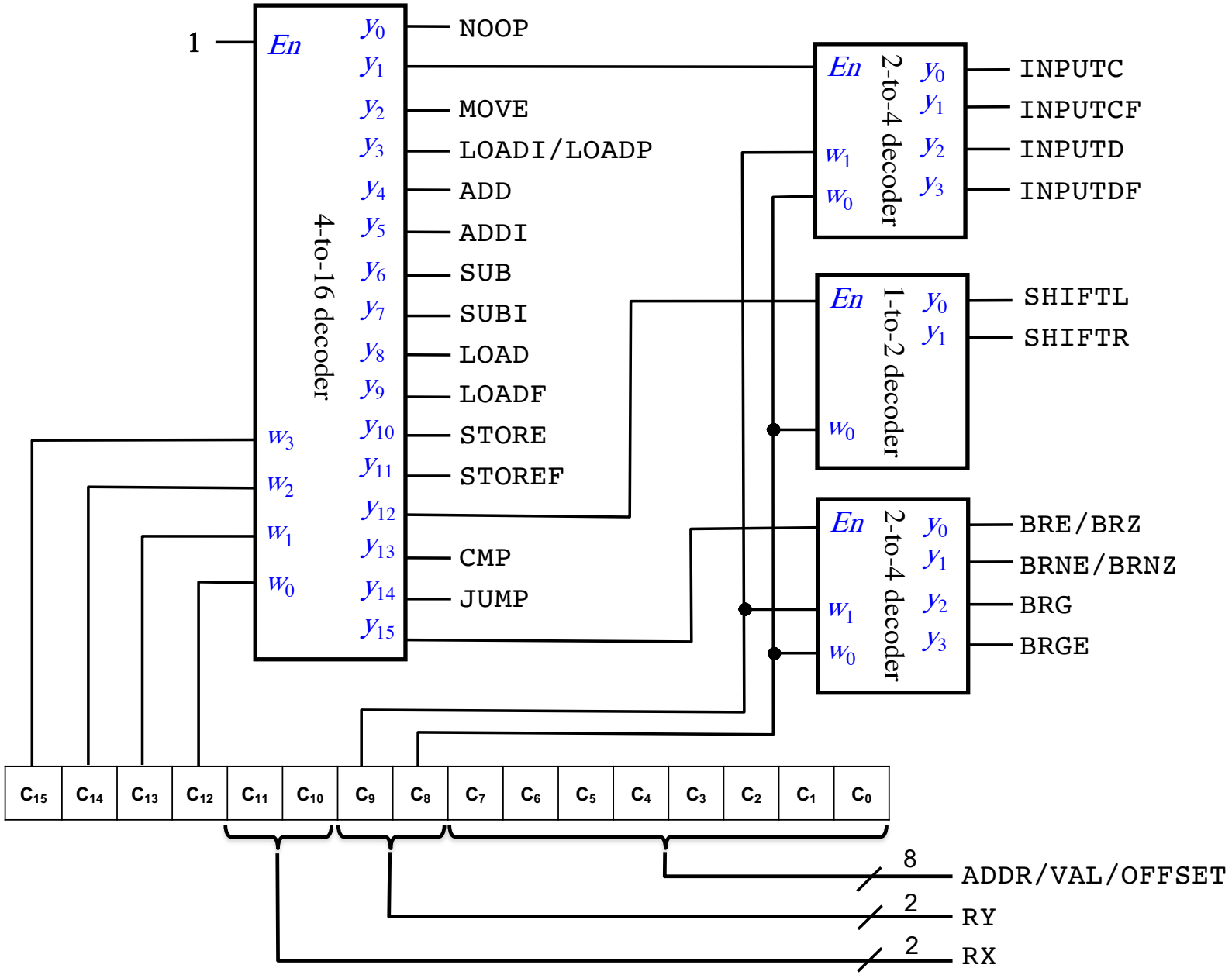

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 NOOP
y_1
y_2 MOVE
y_3 LOADI/LOADP
y_4 ADD
y_5 ADDI
y_6 SUB
y_7 SUBI
y_8 LOAD
y_9 LOADF
y_{10} STORE
y_{11} STOREF
y_{12}
y_{13} CMP
y_{14} JUMP
y_{15}
En
2-to-4 decoder
w_1
w_0
y_0 INPUTC
y_1 INPUTCF
y_2 INPUTD
y_3 INPUTDF
En
1-to-2 decoder
w_0
y_0 SHIFTL
y_1 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 BRE/BRZ
y_1 BRNE/BRNZ
y_2 BRG
y_3 BRGE
C_{15} C_{14} C_{13} C_{12} C_{11} C_{10} C_9 C_8 C_7 C_6 C_5 C_4 C_3 C_2 C_1 C_0
8
ADDR/VAL/OFFSET


1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 NOOP
y_1
y_2 MOVE
y_3 LOADI/LOADP
y_4 ADD
y_5 ADDI
y_6 SUB
y_7 SUBI
y_8 LOAD
y_9 LOADF
y_{10} STORE
y_{11} STOREF
y_{12}
y_{13} CMP
y_{14} JUMP
y_{15}
En
2-to-4 decoder
w_1
w_0
y_0 INPUTC
y_1 INPUTCF
y_2 INPUTD
y_3 INPUTDF
En
1-to-2 decoder
w_0
y_0 SHIFTL
y_1 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 BRE/BRZ
y_1 BRNE/BRNZ
y_2 BRG
y_3 BRGE
C15 C14 C13 C12 C11 C10 C9 C8 C7 C6 C5 C4 C3 C2 C1 C0
8
ADDR/VAL/OFFSET
2
RX

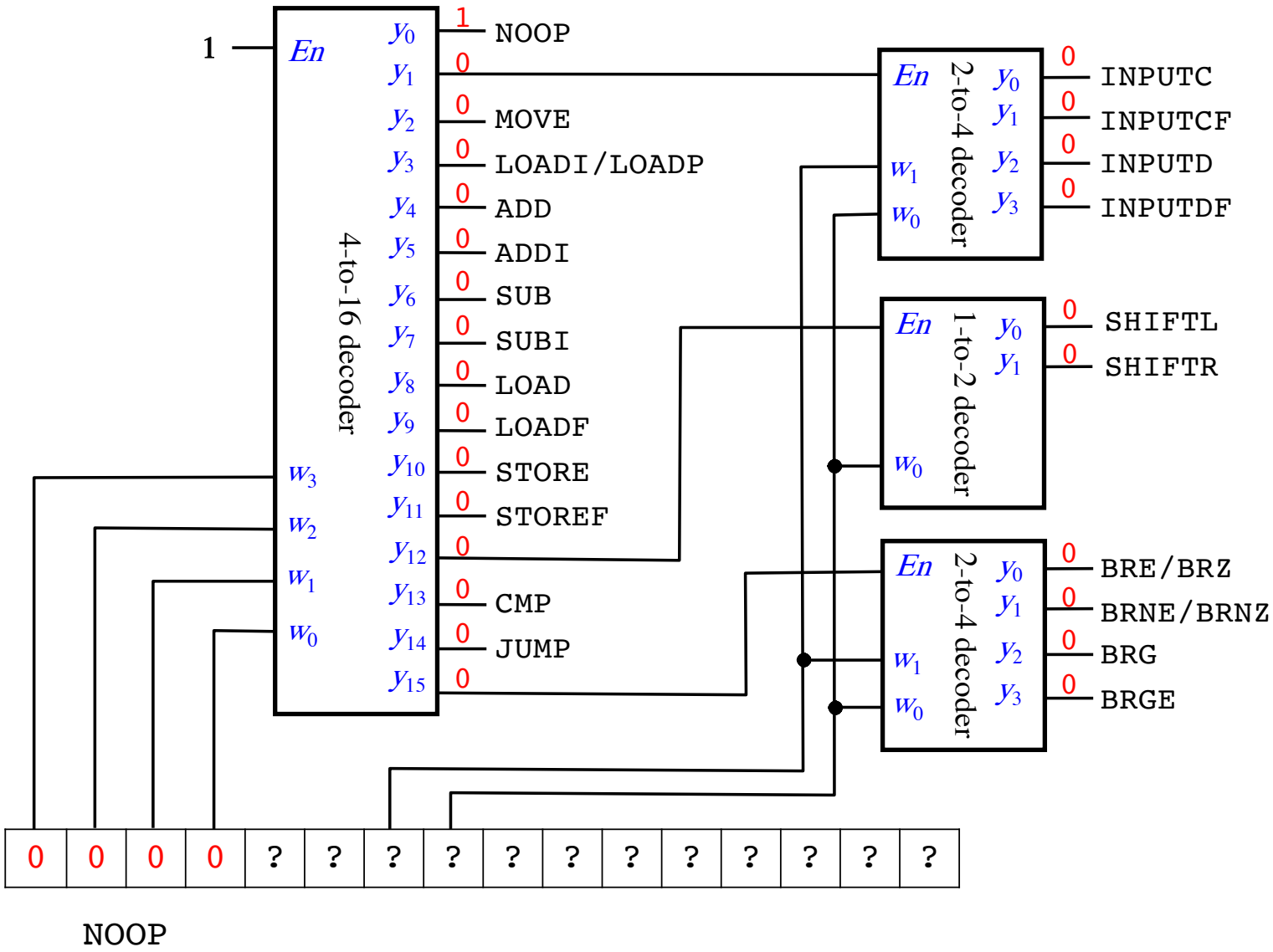

1
En
4-to-16 decoder
y_0 NOOP
y_1
y_2 MOVE
y_3 LOADI/LOADP
y_4 ADD
y_5 ADDI
y_6 SUB
y_7 SUBI
y_8 LOAD
y_9 LOADF
y_{10} STORE
y_{11} STOREF
y_{12}
y_{13} CMP
y_{14} JUMP
y_{15}
w_3
w_2
w_1
w_0
En
2-to-4 decoder
y_0 INPUTC
y_1 INPUTCF
y_2 INPUTD
y_3 INPUTDF
w_1
w_0
En
1-to-2 decoder
y_0 SHIFTL
y_1 SHIFTR
w_0
En
2-to-4 decoder
y_0 BRE/BRZ
y_1 BRNE/BRNZ
y_2 BRG
y_3 BRGE
w_1
w_0
These 2 bits
are shared
C_{15} C_{14} C_{13} C_{12} C_{11} C_{10} C_9 C_8 C_7 C_6 C_5 C_4 C_3 C_2 C_1 C_0
8
ADDR/VAL/OFFSET
2
RX

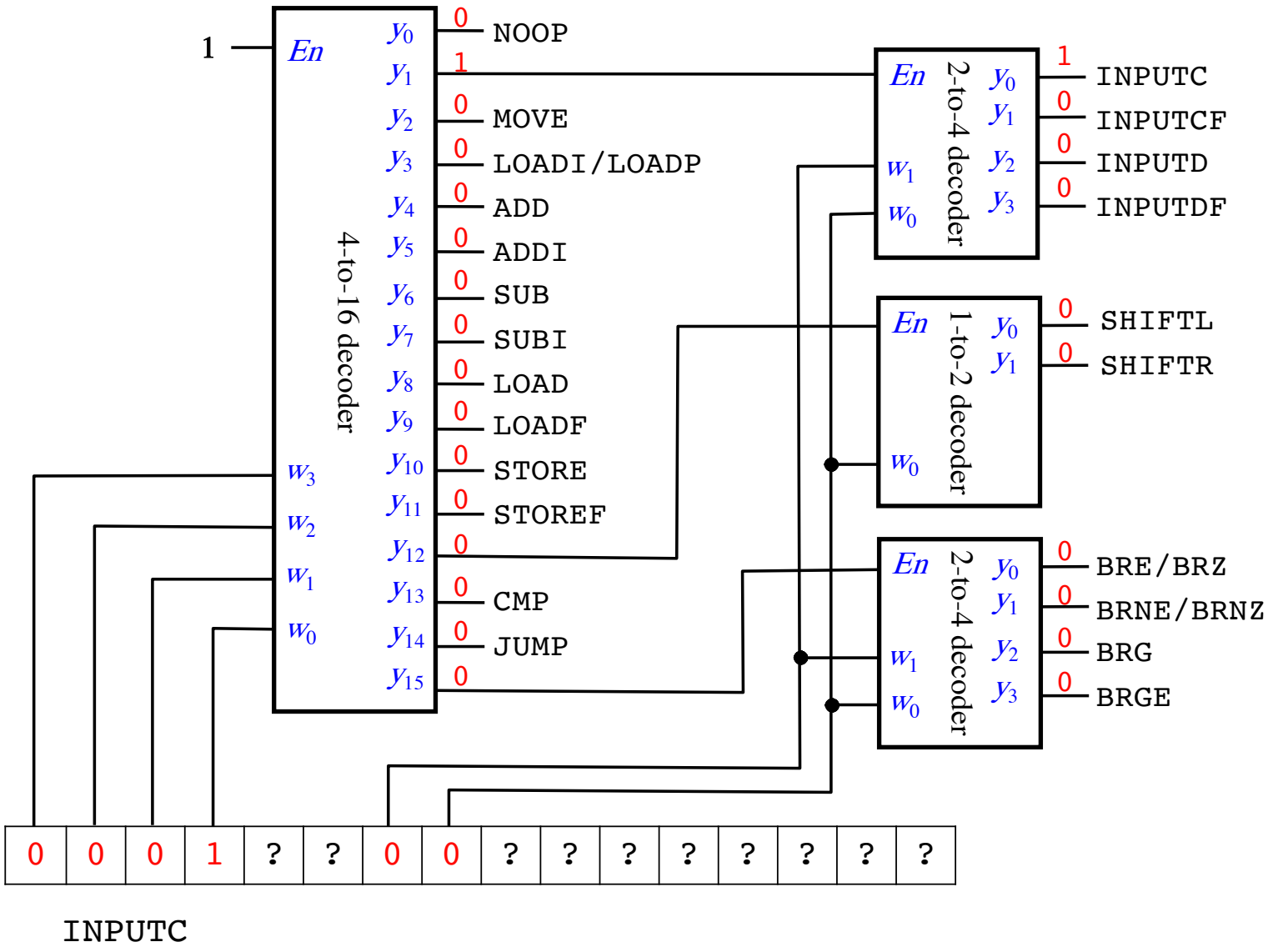

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 NOOP
y_1
y_2 MOVE
y_3 LOADI/LOADP
y_4 ADD
y_5 ADDI
y_6 SUB
y_7 SUBI
y_8 LOAD
y_9 LOADF
y_{10} STORE
y_{11} STOREF
y_{12}
y_{13} CMP
y_{14} JUMP
y_{15}
En
2-to-4 decoder
w_1
w_0
y_0 INPUTC
y_1 INPUTCF
y_2 INPUTD
y_3 INPUTDF
En
1-to-2 decoder
w_0
y_0 SHIFTL
y_1 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 BRE/BRZ
y_1 BRNE/BRNZ
y_2 BRG
y_3 BRGE
Sometimes
they encode
the register RY
C15 C14 C13 C12 C11 C10 C9 C8 C7 C6 C5 C4 C3 C2 C1 C0
8
ADDR/VAL/OFFSET
2
RY
2
RX

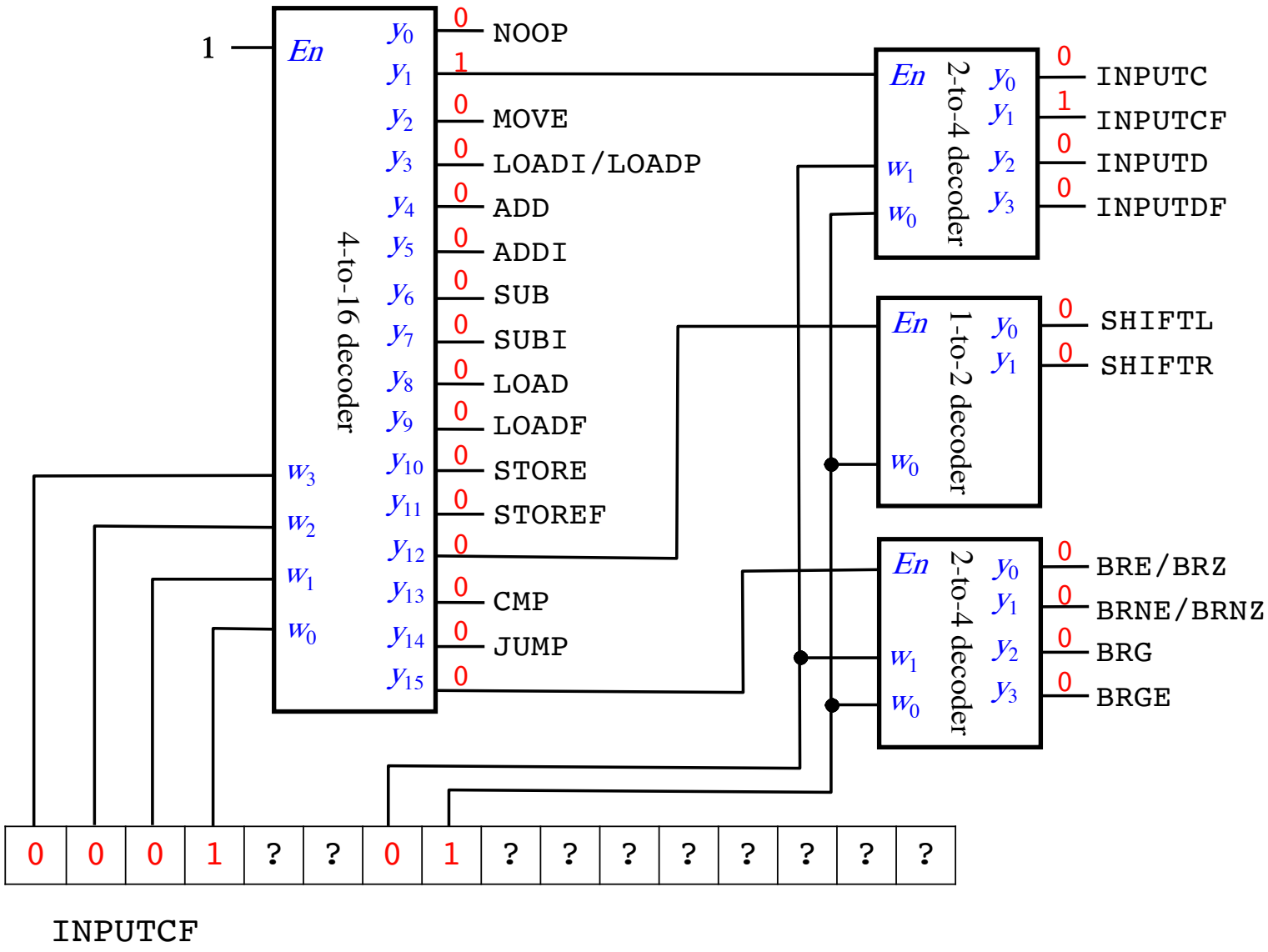

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 NOOP
y_1
y_2 MOVE
y_3 LOADI/LOADP
y_4 ADD
y_5 ADDI
y_6 SUB
y_7 SUBI
y_8 LOAD
y_9 LOADF
y_{10} STORE
y_{11} STOREF
y_{12}
y_{13} CMP
y_{14} JUMP
y_{15}
En
2-to-4 decoder
w_1
w_0
y_0 INPUTC
y_1 INPUTCF
y_2 INPUTD
y_3 INPUTDF
En
1-to-2 decoder
w_0
y_0 SHIFTL
y_1 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 BRE/BRZ
y_1 BRNE/BRNZ
y_2 BRG
y_3 BRGE
C15 C14 C13 C12 C11 C10 C9 C8 C7 C6 C5 C4 C3 C2 C1 C0
8
ADDR/VAL/OFFSET
2
RY
2
RX

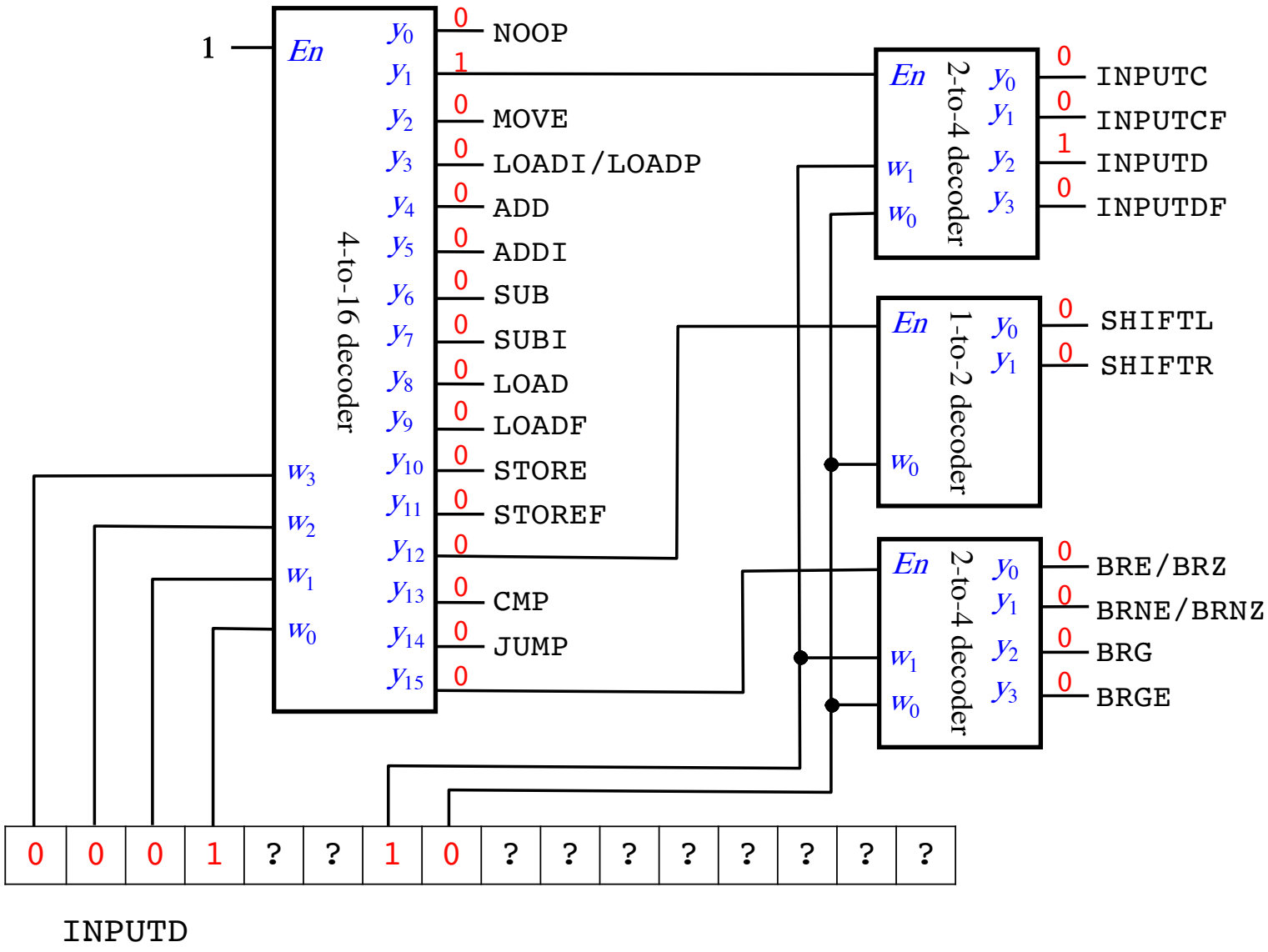

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 NOOP
y_1
y_2 MOVE
y_3 LOADI/LOADP
y_4 ADD
y_5 ADDI
y_6 SUB
y_7 SUBI
y_8 LOAD
y_9 LOADF
y_{10} STORE
y_{11} STOREF
y_{12}
y_{13} CMP
y_{14} JUMP
y_{15}
En
2-to-4 decoder
w_1
w_0
y_0 INPUTC
y_1 INPUTCF
y_2 INPUTD
y_3 INPUTDF
En
1-to-2 decoder
w_0
y_0 SHIFTL
y_1 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 BRE/BRZ
y_1 BRNE/BRNZ
y_2 BRG
y_3 BRGE
C_{15} C_{14} C_{13} C_{12} C_{11} C_{10} C_9 C_8 C_7 C_6 C_5 C_4 C_3 C_2 C_1 C_0
8
ADDR/VAL/OFFSET
2
RY
2
RX

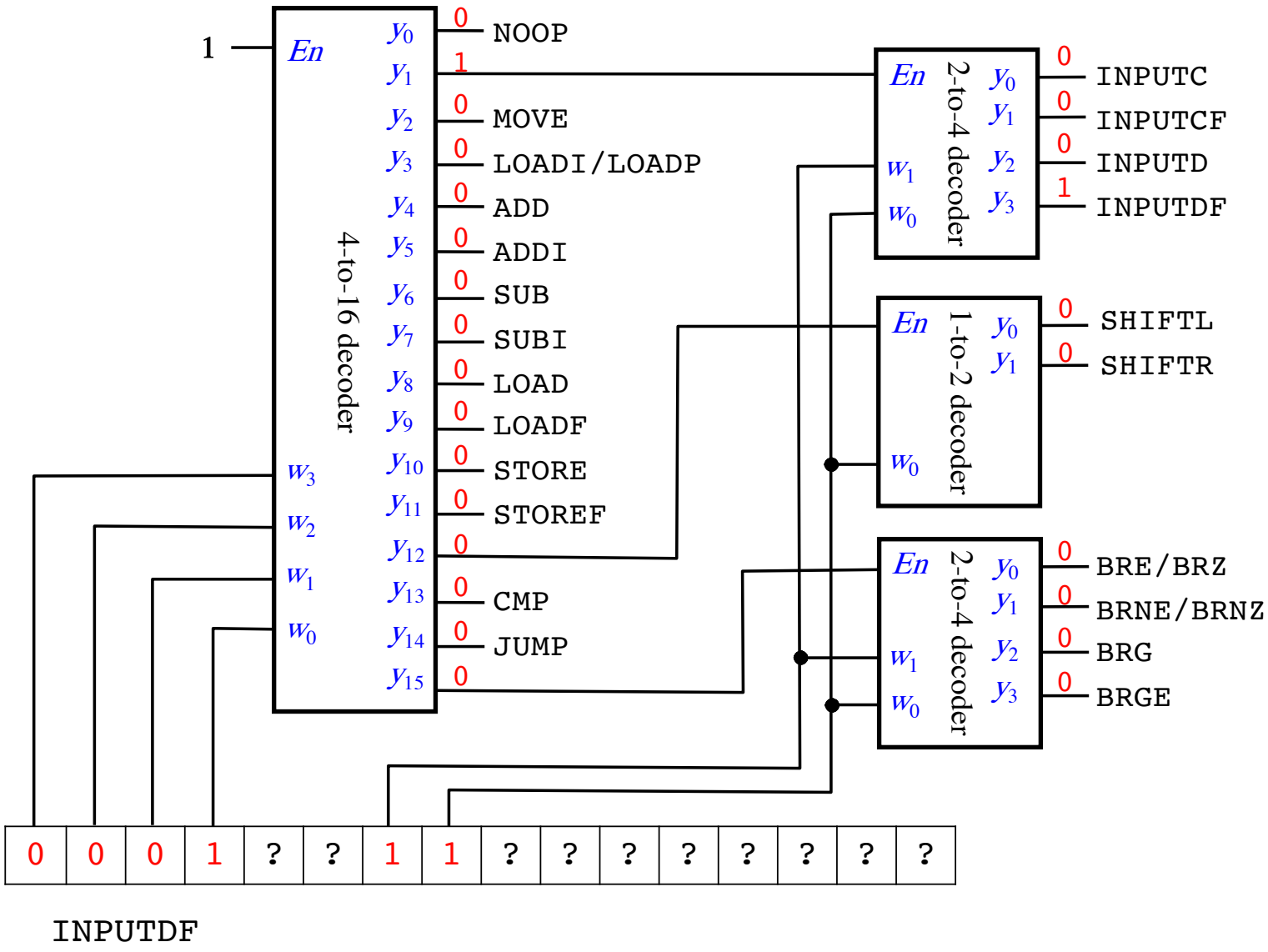
The OPCODE decoder outputs are one-hot encoded

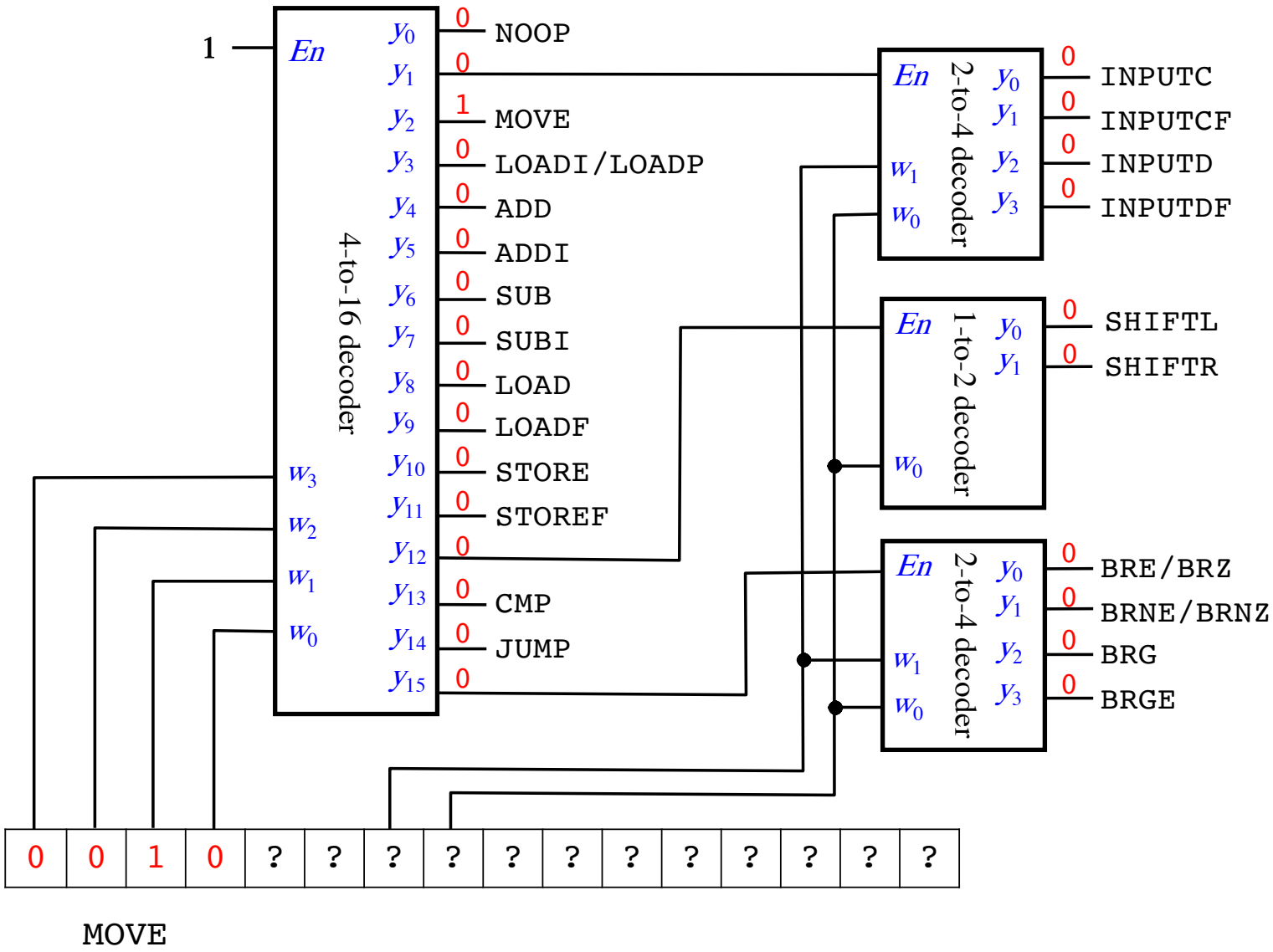

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 1 NOOP
y_1 0
y_2 0 MOVE
y_3 0 LOADI/LOADP
y_4 0 ADD
y_5 0 ADDI
y_6 0 SUB
y_7 0 SUBI
y_8 0 LOAD
y_9 0 LOADF
y_{10} 0 STORE
y_{11} 0 STOREF
y_{12} 0
y_{13} 0 CMP
y_{14} 0 JUMP
y_{15} 0
En
2-to-4 decoder
w_1
w_0
y_0 0 INPUTC
y_1 0 INPUTCF
y_2 0 INPUTD
y_3 0 INPUTDF
En
1-to-2 decoder
w_0
y_0 0 SHIFTL
y_1 0 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 0 BRE/BRZ
y_1 0 BRNE/BRNZ
y_2 0 BRG
y_3 0 BRGE
0 0 0 0 ? ? ? ? ? ? ? ? ? ? ? ?
NOOP

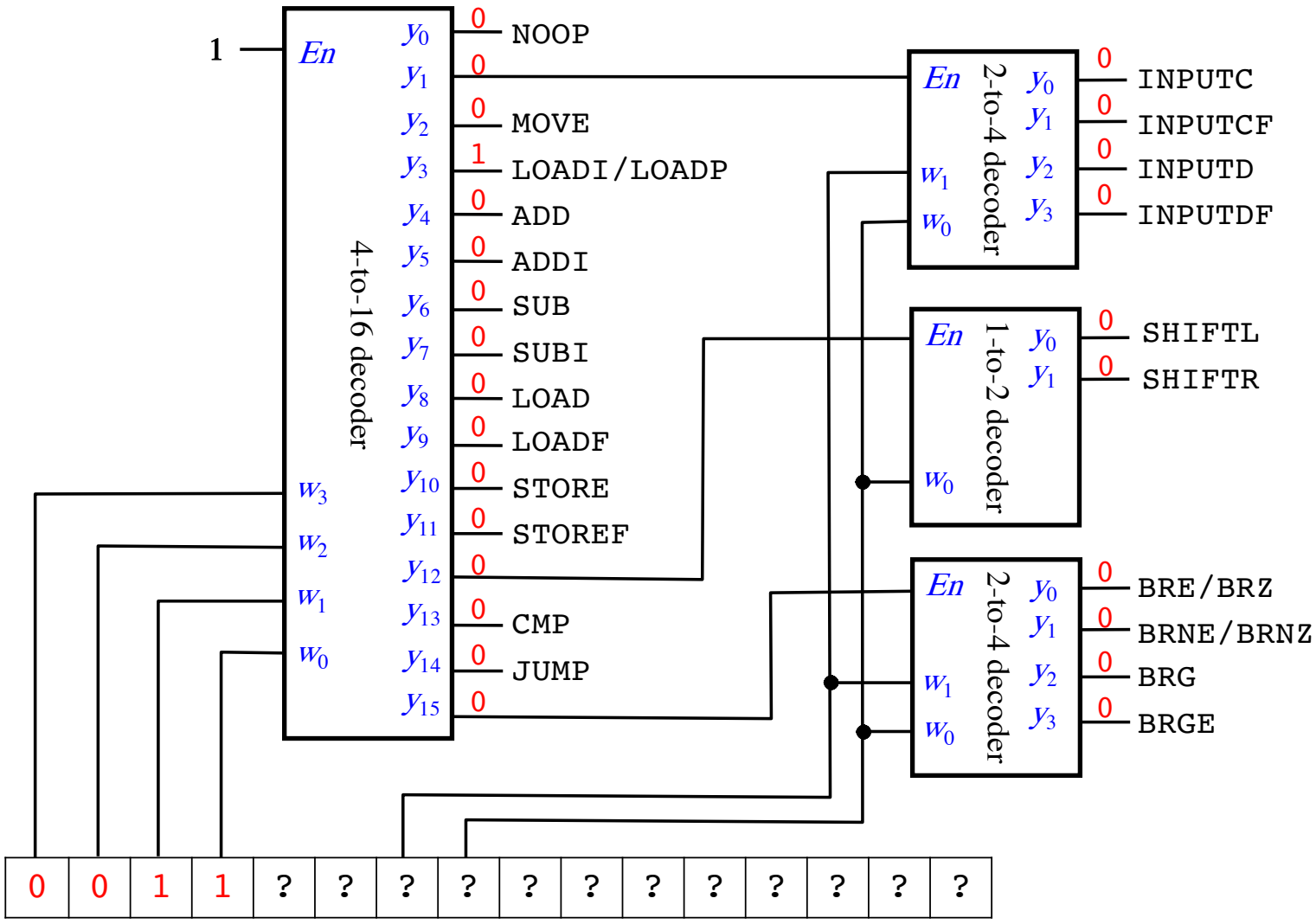

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 0 NOOP
y_1 1
y_2 0 MOVE
y_3 0 LOADI/LOADP
y_4 0 ADD
y_5 0 ADDI
y_6 0 SUB
y_7 0 SUBI
y_8 0 LOAD
y_9 0 LOADF
y_{10} 0 STORE
y_{11} 0 STOREF
y_{12} 0
y_{13} 0 CMP
y_{14} 0 JUMP
y_{15} 0
En
2-to-4 decoder
w_1
w_0
y_0 1 INPUTC
y_1 0 INPUTCF
y_2 0 INPUTD
y_3 0 INPUTDF
En
1-to-2 decoder
w_0
y_0 0 SHIFTL
y_1 0 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 0 BRE/BRZ
y_1 0 BRNE/BRNZ
y_2 0 BRG
y_3 0 BRGE
0 0 0 1 ? ? 0 0 ? ? ? ? ? ? ? ?
INPUTC

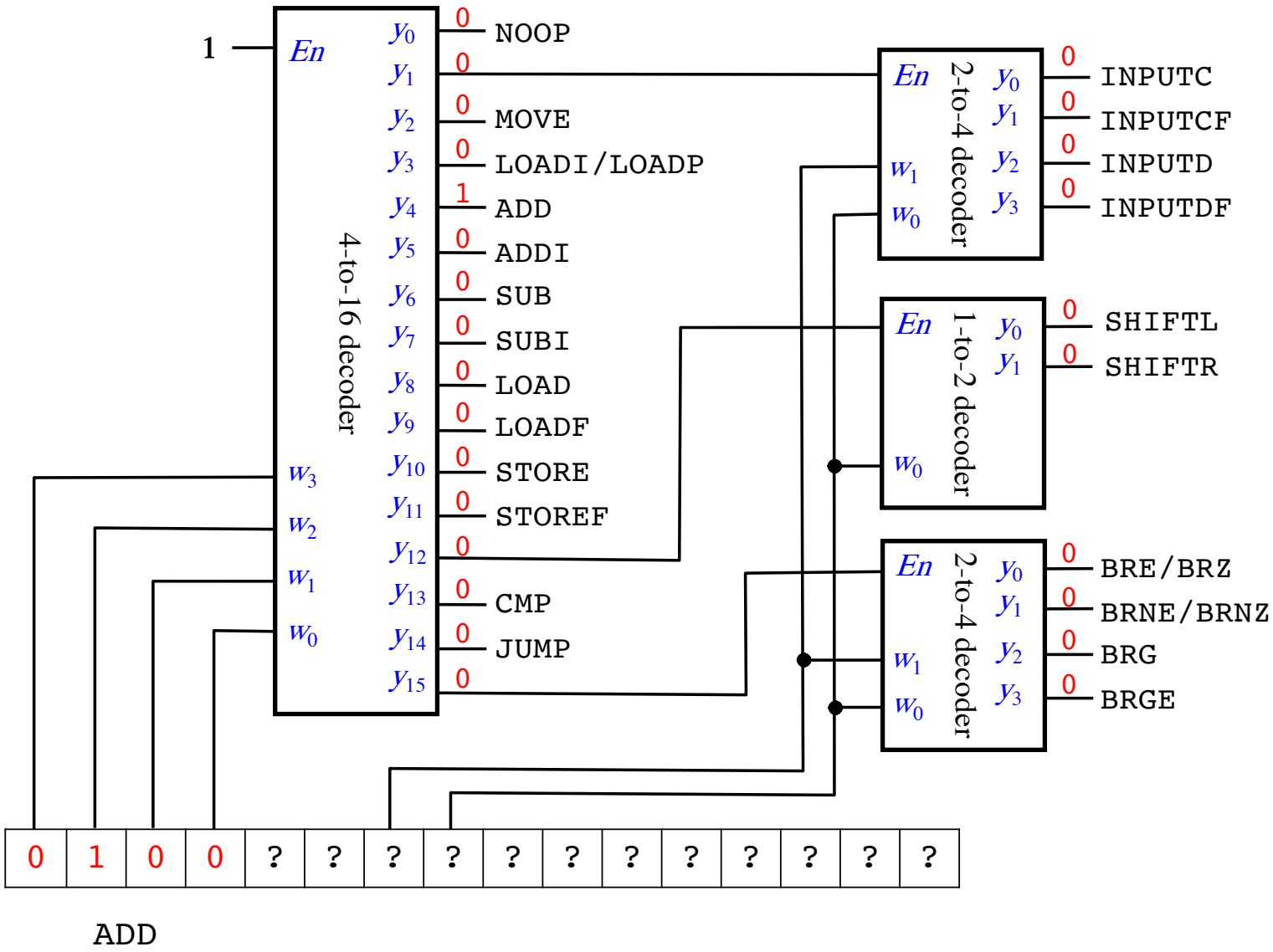

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 0 NOOP
y_1 1
y_2 0 MOVE
y_3 0 LOADI/LOADP
y_4 0 ADD
y_5 0 ADDI
y_6 0 SUB
y_7 0 SUBI
y_8 0 LOAD
y_9 0 LOADF
y_{10} 0 STORE
y_{11} 0 STOREF
y_{12} 0
y_{13} 0 CMP
y_{14} 0 JUMP
y_{15} 0
En
2-to-4 decoder
w_1
w_0
y_0 0 INPUTC
y_1 1 INPUTCF
y_2 0 INPUTD
y_3 0 INPUTDF
En
1-to-2 decoder
w_0
y_0 0 SHIFTL
y_1 0 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 0 BRE/BRZ
y_1 0 BRNE/BRNZ
y_2 0 BRG
y_3 0 BRGE
0 0 0 1 ? ? 0 1 ? ? ? ? ? ? ? ?
INPUTCF

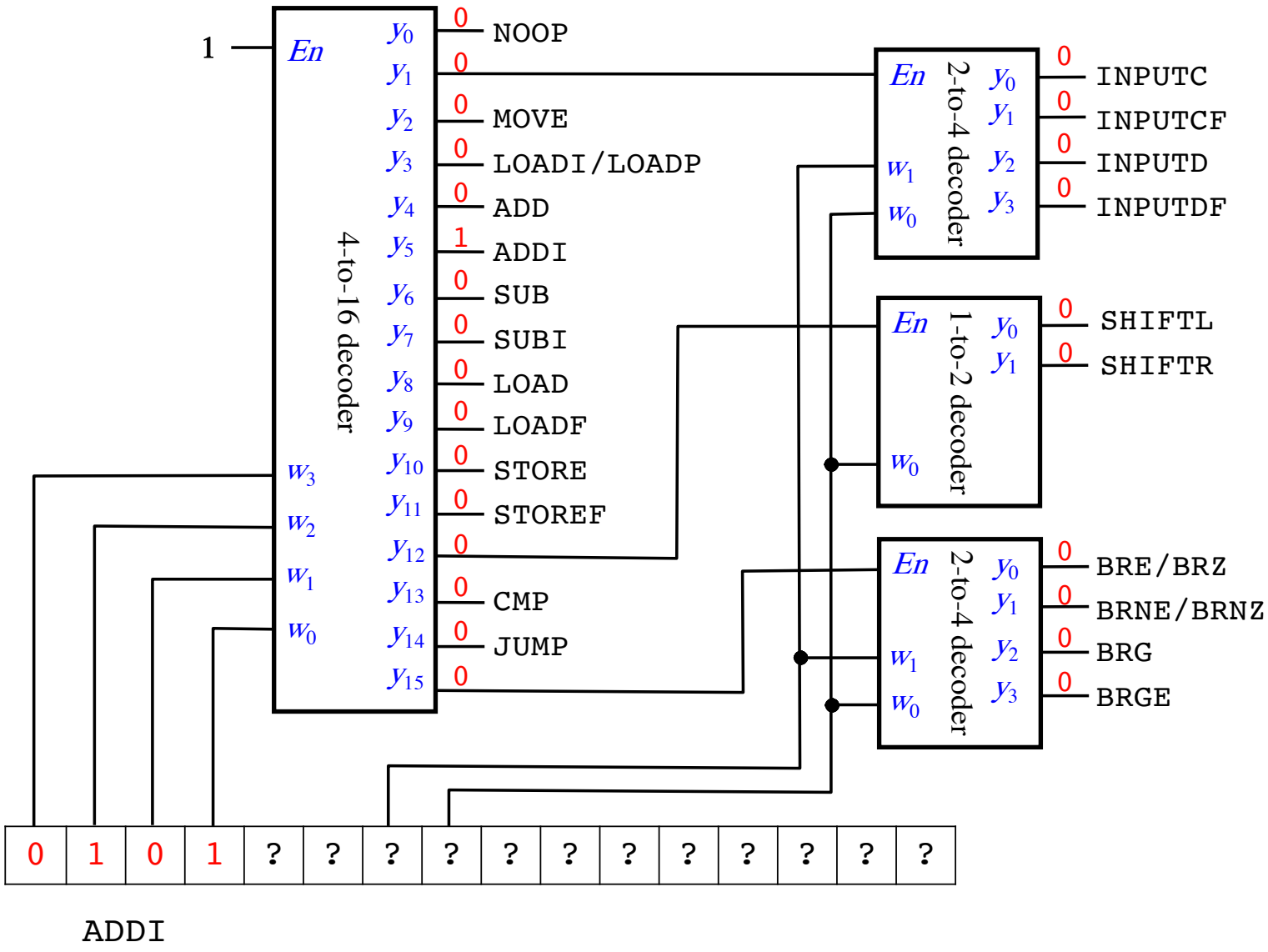

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 0 NOOP
y_1 1
y_2 0 MOVE
y_3 0 LOADI/LOADP
y_4 0 ADD
y_5 0 ADDI
y_6 0 SUB
y_7 0 SUBI
y_8 0 LOAD
y_9 0 LOADF
y_{10} 0 STORE
y_{11} 0 STOREF
y_{12} 0
y_{13} 0 CMP
y_{14} 0 JUMP
y_{15} 0
En
2-to-4 decoder
w_1
w_0
y_0 0 INPUTC
y_1 0 INPUTCF
y_2 1 INPUTD
y_3 0 INPUTDF
En
1-to-2 decoder
w_0
y_0 0 SHIFTL
y_1 0 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 0 BRE/BRZ
y_1 0 BRNE/BRNZ
y_2 0 BRG
y_3 0 BRGE
0 0 0 1 ? ? 1 0 ? ? ? ? ? ? ? ?
INPUTD

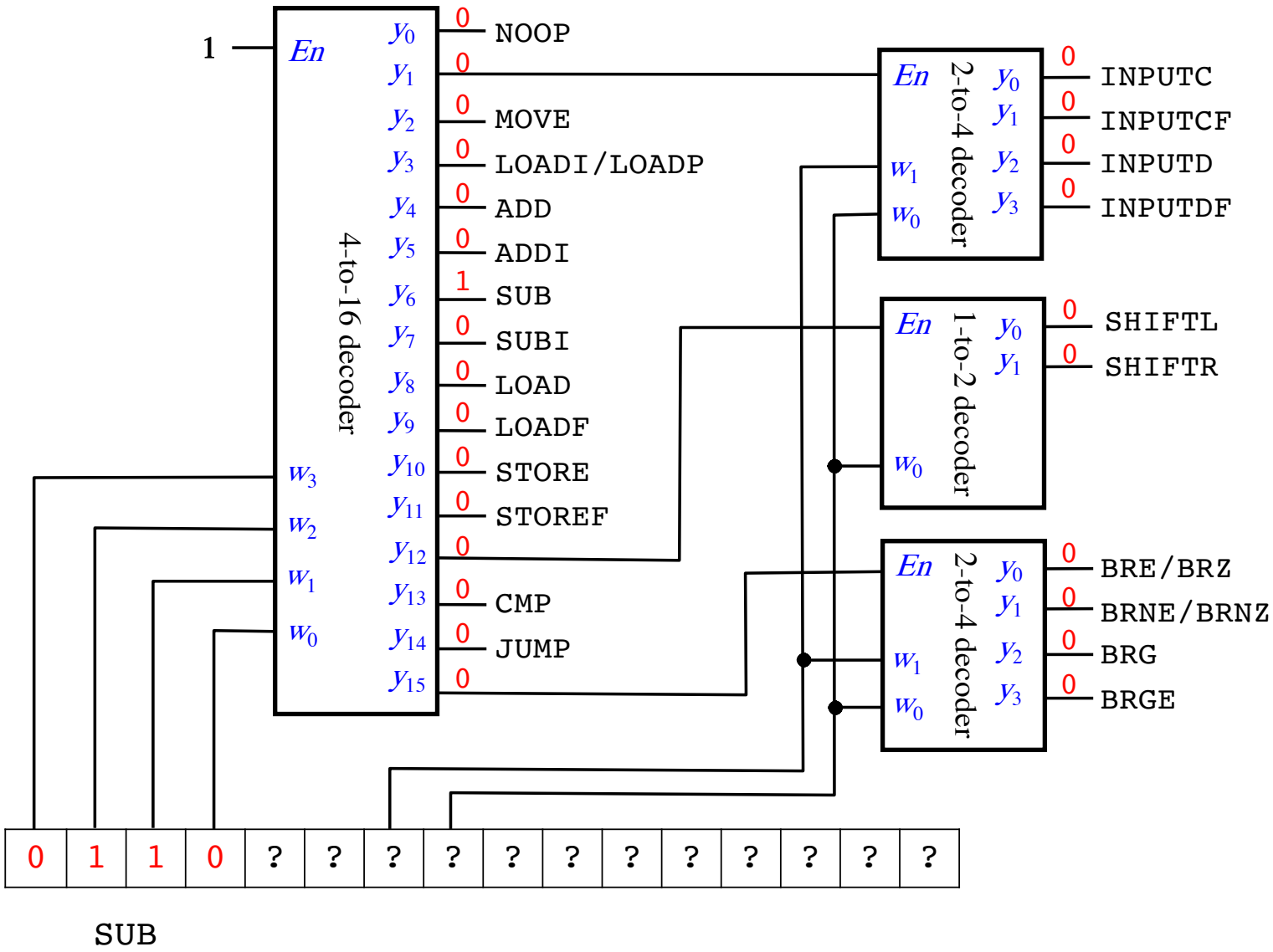

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 0 NOOP
y_1 1
y_2 0 MOVE
y_3 0 LOADI/LOADP
y_4 0 ADD
y_5 0 ADDI
y_6 0 SUB
y_7 0 SUBI
y_8 0 LOAD
y_9 0 LOADF
y_{10} 0 STORE
y_{11} 0 STOREF
y_{12} 0
y_{13} 0 CMP
y_{14} 0 JUMP
y_{15} 0
En
2-to-4 decoder
w_1
w_0
y_0 0 INPUTC
y_1 0 INPUTCF
y_2 0 INPUTD
y_3 1 INPUTDF
En
1-to-2 decoder
w_0
y_0 0 SHIFTL
y_1 0 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 0 BRE/BRZ
y_1 0 BRNE/BRNZ
y_2 0 BRG
y_3 0 BRGE
0 0 0 1 ? ? 1 1 ? ? ? ? ? ? ? ?
INPUTDF

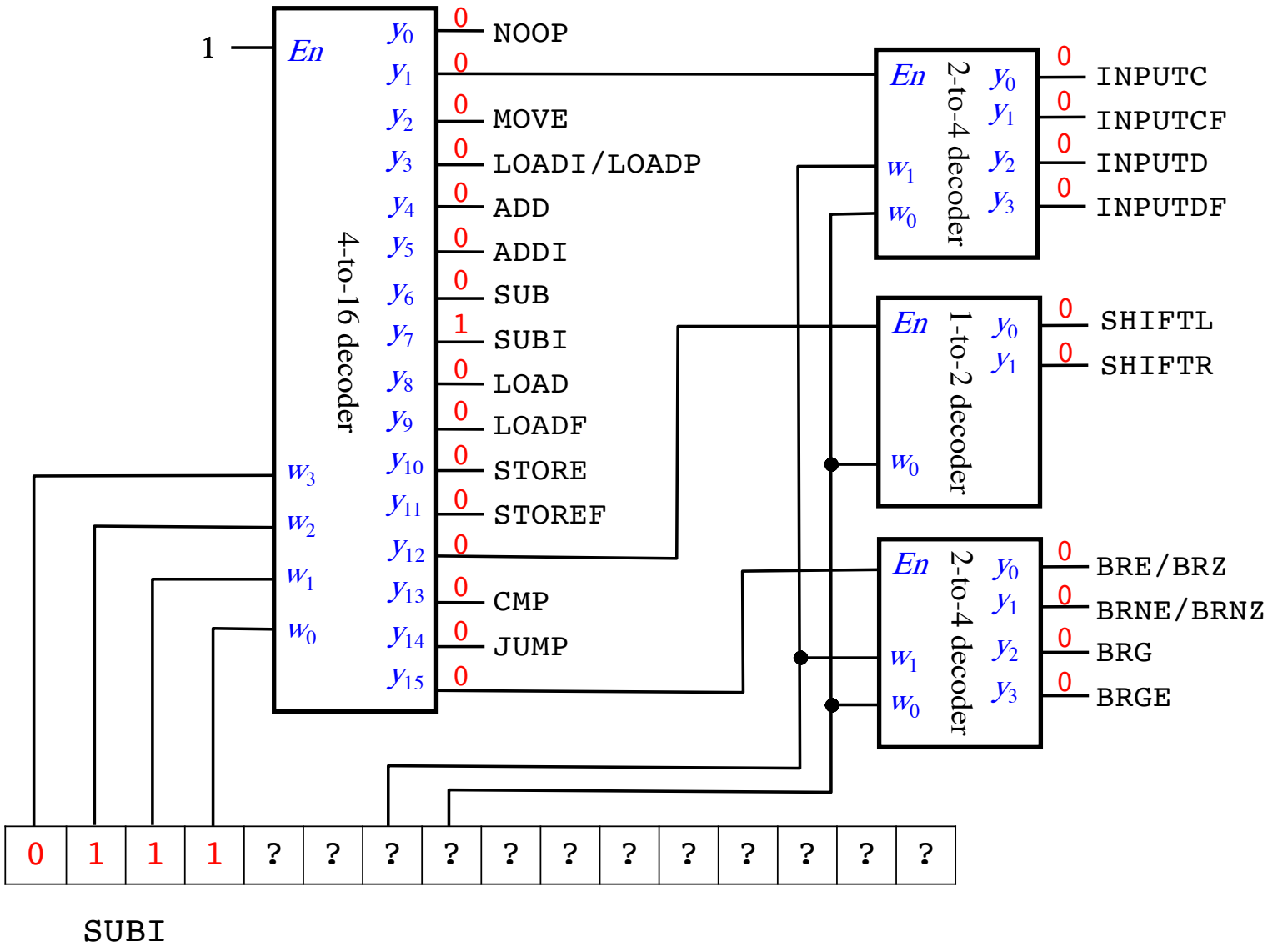

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 0 NOOP
y_1 0
y_2 1 MOVE
y_3 0 LOADI/LOADP
y_4 0 ADD
y_5 0 ADDI
y_6 0 SUB
y_7 0 SUBI
y_8 0 LOAD
y_9 0 LOADF
y_{10} 0 STORE
y_{11} 0 STOREF
y_{12} 0
y_{13} 0 CMP
y_{14} 0 JUMP
y_{15} 0
En
2-to-4 decoder
w_1
w_0
y_0 0 INPUTC
y_1 0 INPUTCF
y_2 0 INPUTD
y_3 0 INPUTDF
En
1-to-2 decoder
w_0
y_0 0 SHIFTL
y_1 0 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 0 BRE/BRZ
y_1 0 BRNE/BRNZ
y_2 0 BRG
y_3 0 BRGE
0 0 1 0 ? ? ? ? ? ? ? ? ? ? ? ?
MOVE

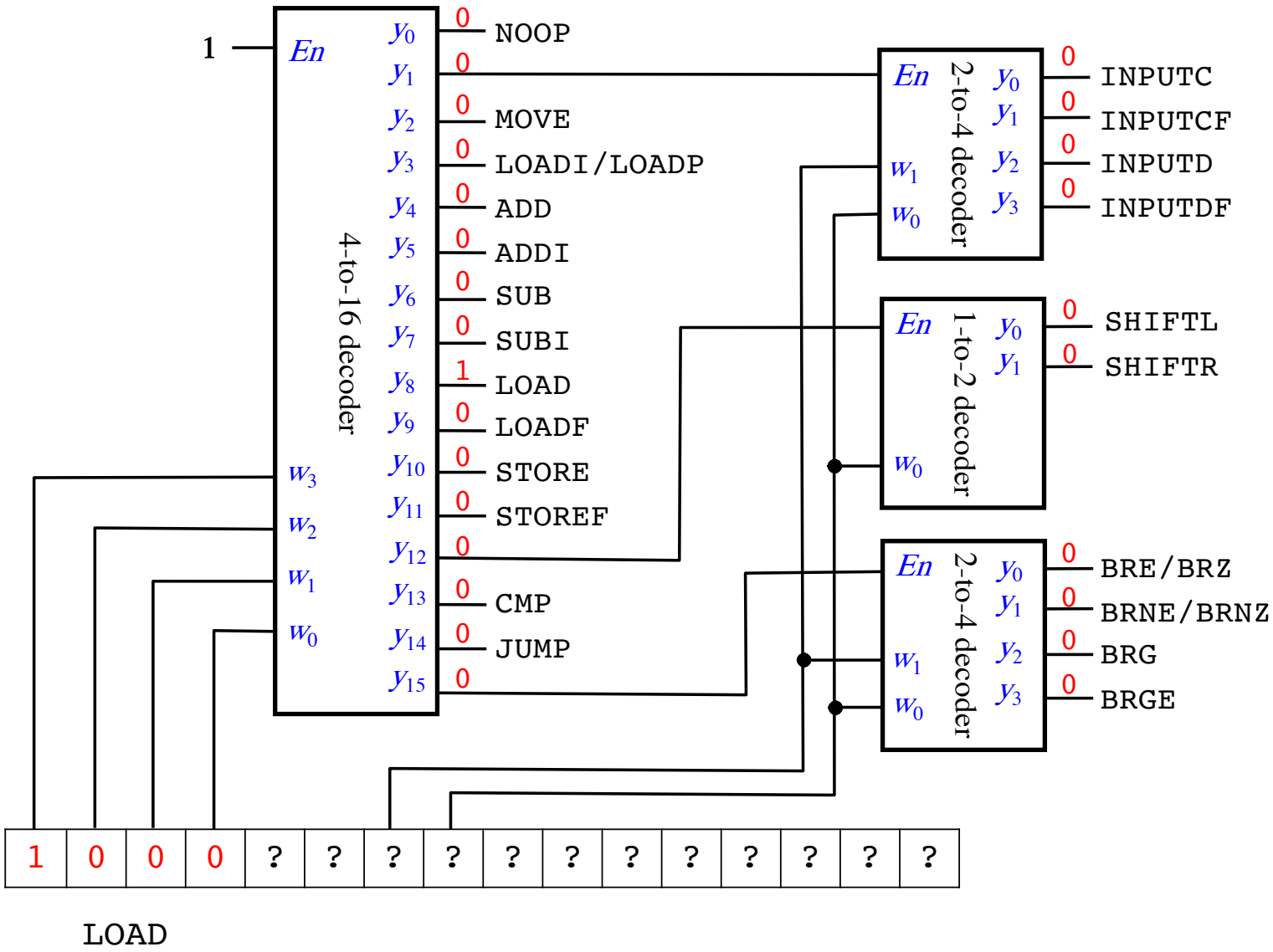

1
En
4-to-16 decoder
y_0
y_1
y_2
y_3
y_4
y_5
y_6
y_7
y_8
y_9
y_{10}
y_{11}
y_{12}
y_{13}
y_{14}
y_{15}
w_3
w_2
w_1
w_0
0 NOOP
0
0 MOVE
1 LOADI/LOADP
0 ADD
0 ADDI
0 SUB
0 SUBI
0 LOAD
0 LOADF
0 STORE
0 STOREF
0
0 CMP
0 JUMP
0
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
0 INPUTC
0 INPUTCF
0 INPUTD
0 INPUTDF
En
1-to-2 decoder
y_0
y_1
w_0
0 SHIFTL
0 SHIFTR
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
0 BRE/BRZ
0 BRNE/BRNZ
0 BRG
0 BRGE
0 0 1 1 ? ? ? ? ? ? ? ? ? ? ? ?
LOADI/LOADP

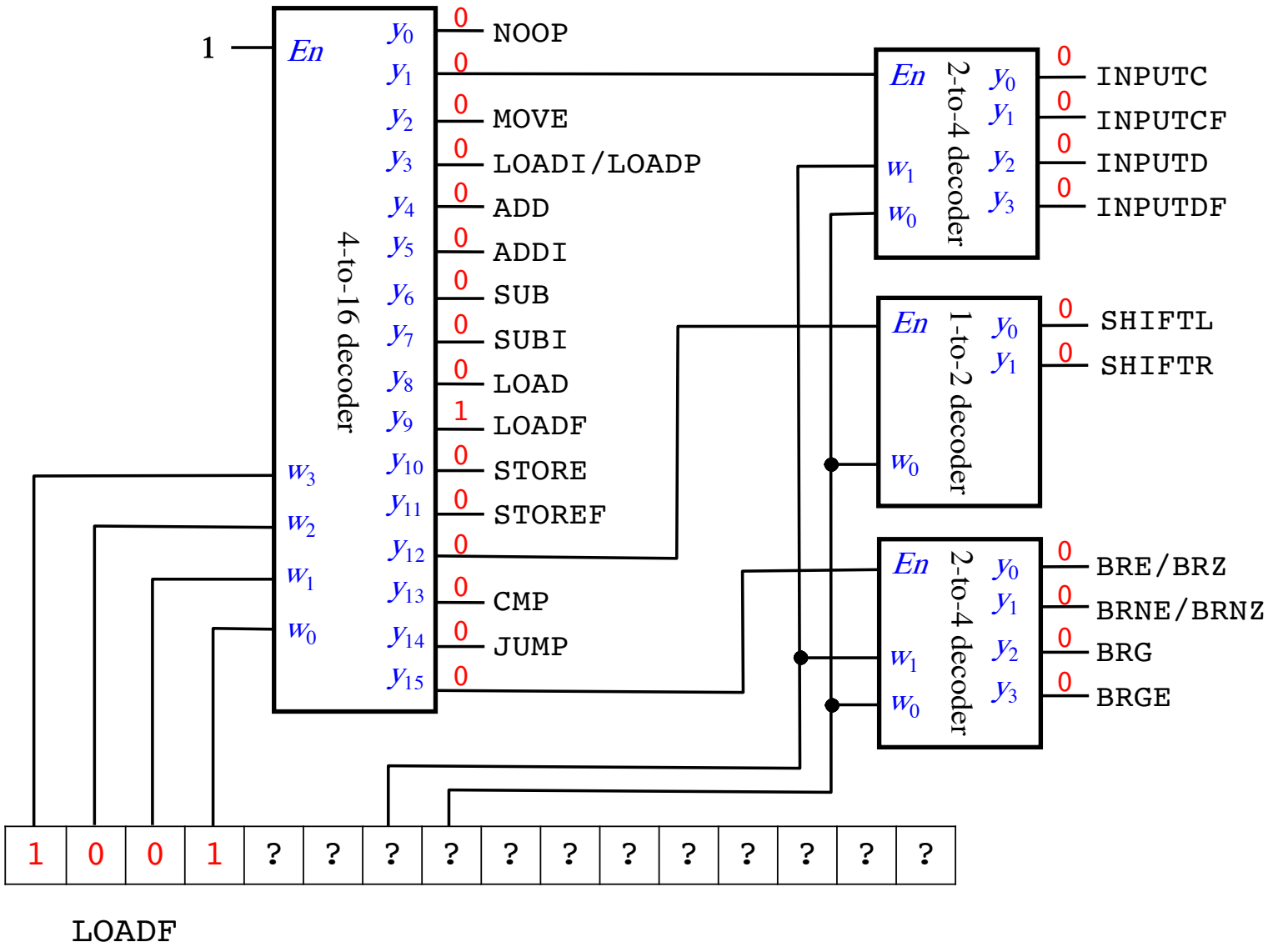

1
En
4-to-16 decoder
y_0
y_1
y_2
y_3
y_4
y_5
y_6
y_7
y_8
y_9
y_{10}
y_{11}
y_{12}
y_{13}
y_{14}
y_{15}
w_3
w_2
w_1
w_0
0 NOOP
0
0 MOVE
0 LOADI/LOADP
1 ADD
0 ADDI
0 SUB
0 SUBI
0 LOAD
0 LOADF
0 STORE
0 STOREF
0
0 CMP
0 JUMP
0
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
0 INPUTC
0 INPUTCF
0 INPUTD
0 INPUTDF
En
1-to-2 decoder
y_0
y_1
w_0
0 SHIFTL
0 SHIFTR
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
0 BRE/BRZ
0 BRNE/BRNZ
0 BRG
0 BRGE
0 1 0 0 ? ? ? ? ? ? ? ? ? ? ? ?
ADD

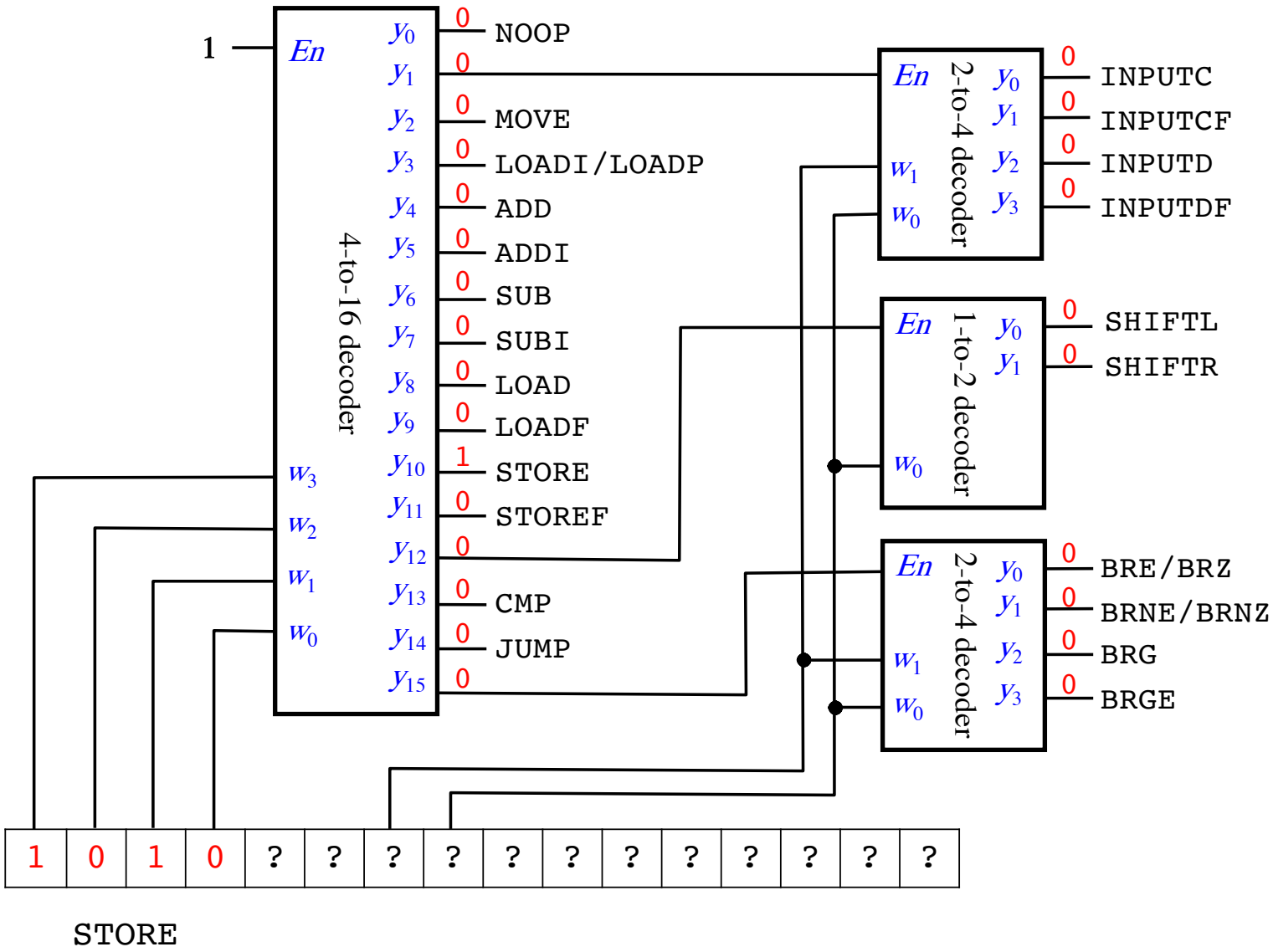

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 0 NOOP
y_1 0
y_2 0 MOVE
y_3 0 LOADI/LOADP
y_4 0 ADD
y_5 1 ADDI
y_6 0 SUB
y_7 0 SUBI
y_8 0 LOAD
y_9 0 LOADF
y_{10} 0 STORE
y_{11} 0 STOREF
y_{12} 0
y_{13} 0 CMP
y_{14} 0 JUMP
y_{15} 0
En
2-to-4 decoder
w_1
w_0
y_0 0 INPUTC
y_1 0 INPUTCF
y_2 0 INPUTD
y_3 0 INPUTDF
En
1-to-2 decoder
w_0
y_0 0 SHIFTL
y_1 0 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 0 BRE/BRZ
y_1 0 BRNE/BRNZ
y_2 0 BRG
y_3 0 BRGE
0 1 0 1 ? ? ? ? ? ? ? ? ? ? ? ?
ADDI

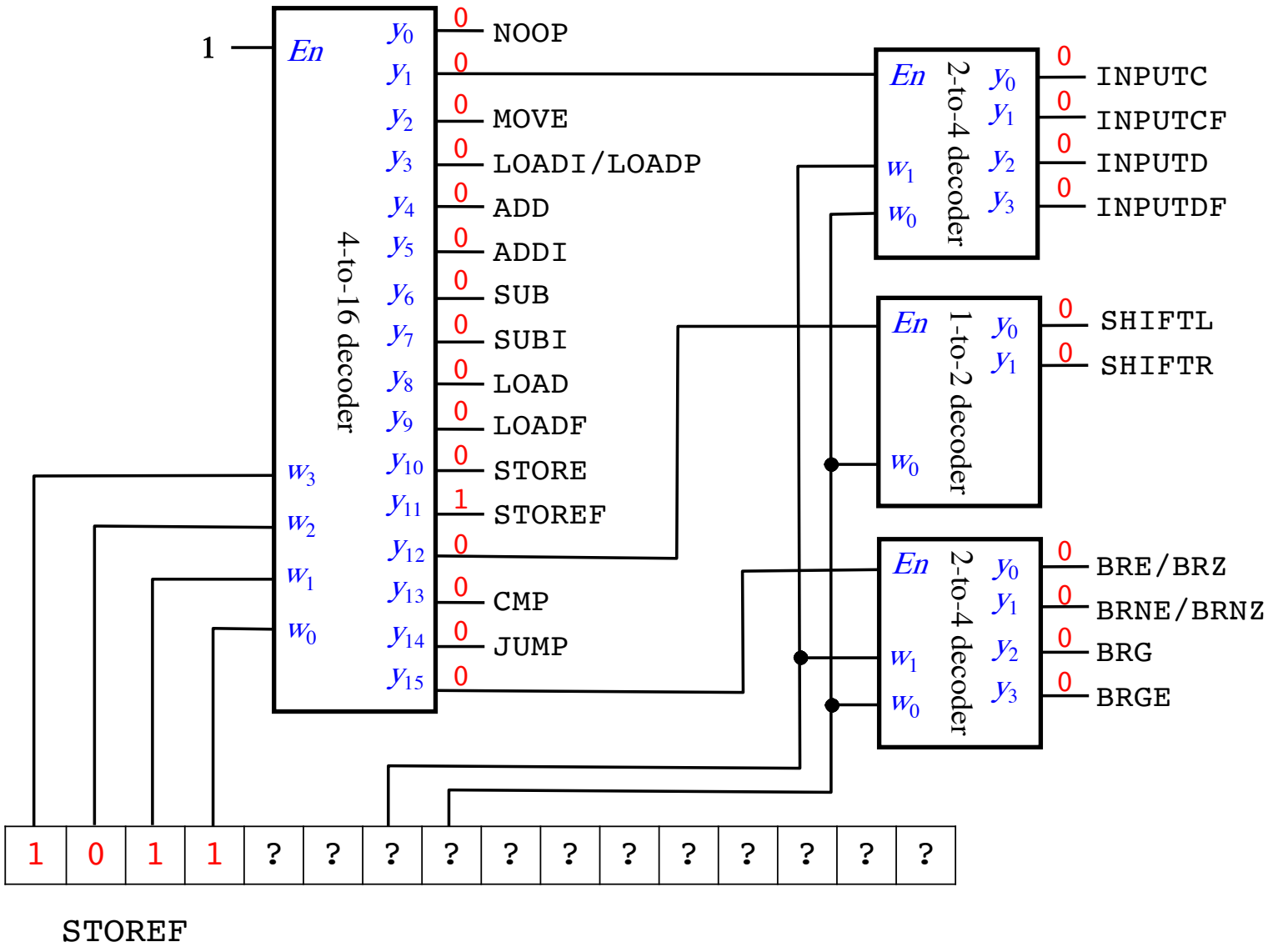

1
En
4-to-16 decoder
y_0
y_1
y_2
y_3
y_4
y_5
y_6
y_7
y_8
y_9
y_{10}
y_{11}
y_{12}
y_{13}
y_{14}
y_{15}
w_3
w_2
w_1
w_0
0 NOOP
0
0 MOVE
0 LOADI/LOADP
0 ADD
0 ADDI
1 SUB
0 SUBI
0 LOAD
0 LOADF
0 STORE
0 STOREF
0
0 CMP
0 JUMP
0
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
0 INPUTC
0 INPUTCF
0 INPUTD
0 INPUTDF
En
1-to-2 decoder
y_0
y_1
w_0
0 SHIFTL
0 SHIFTR
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
0 BRE/BRZ
0 BRNE/BRNZ
0 BRG
0 BRGE
0 1 1 0 ? ? ? ? ? ? ? ? ? ? ? ?
SUB

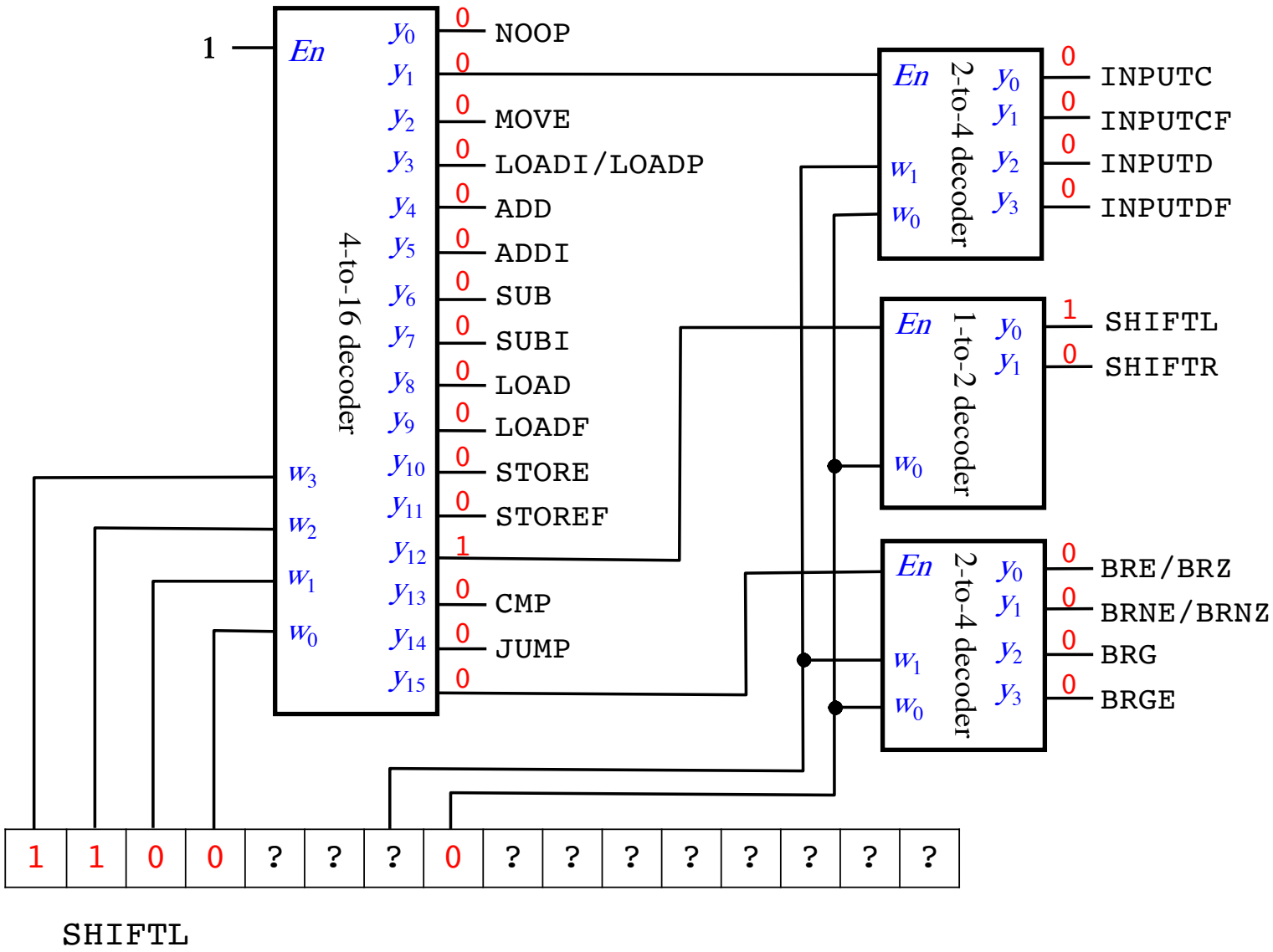

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 0 NOOP
y_1 0
y_2 0 MOVE
y_3 0 LOADI/LOADP
y_4 0 ADD
y_5 0 ADDI
y_6 0 SUB
y_7 1 SUBI
y_8 0 LOAD
y_9 0 LOADF
y_{10} 0 STORE
y_{11} 0 STOREF
y_{12} 0
y_{13} 0 CMP
y_{14} 0 JUMP
y_{15} 0
En
2-to-4 decoder
w_1
w_0
y_0 0 INPUTC
y_1 0 INPUTCF
y_2 0 INPUTD
y_3 0 INPUTDF
En
1-to-2 decoder
w_0
y_0 0 SHIFTL
y_1 0 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 0 BRE/BRZ
y_1 0 BRNE/BRNZ
y_2 0 BRG
y_3 0 BRGE
0 1 1 1 ? ? ? ? ? ? ? ? ? ? ? ?
SUBI

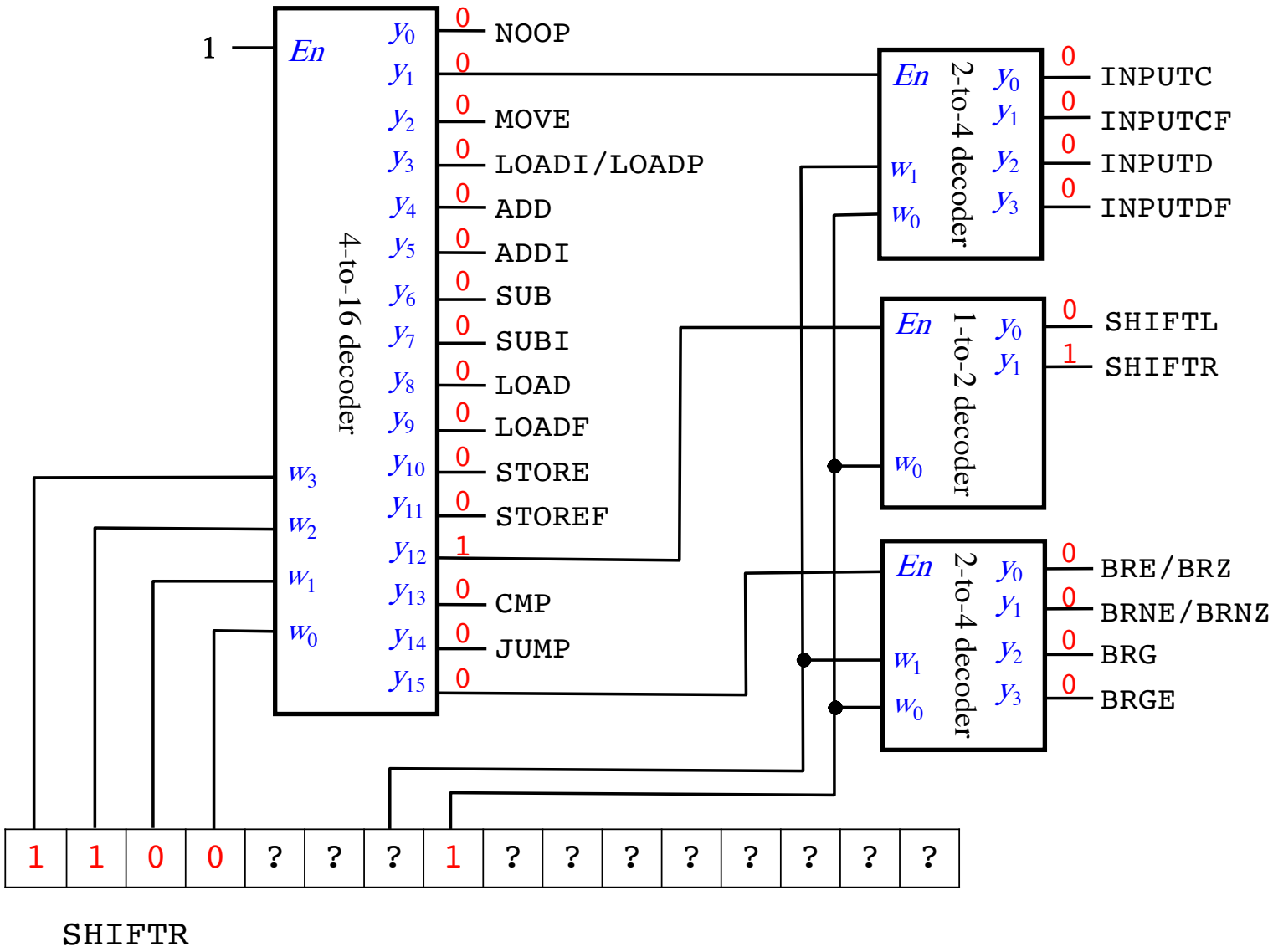

1
En
4-to-16 decoder
y_0
y_1
y_2
y_3
y_4
y_5
y_6
y_7
y_8
y_9
y_{10}
y_{11}
y_{12}
y_{13}
y_{14}
y_{15}
w_3
w_2
w_1
w_0
0 NOOP
0
0 MOVE
0 LOADI/LOADP
0 ADD
0 ADDI
0 SUB
0 SUBI
1 LOAD
0 LOADF
0 STORE
0 STOREF
0
0 CMP
0 JUMP
0
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
0 INPUTC
0 INPUTCF
0 INPUTD
0 INPUTDF
En
1-to-2 decoder
y_0
y_1
w_0
0 SHIFTL
0 SHIFTR
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
0 BRE/BRZ
0 BRNE/BRNZ
0 BRG
0 BRGE
1 0 0 0 ? ? ? ? ? ? ? ? ? ? ? ?
LOAD

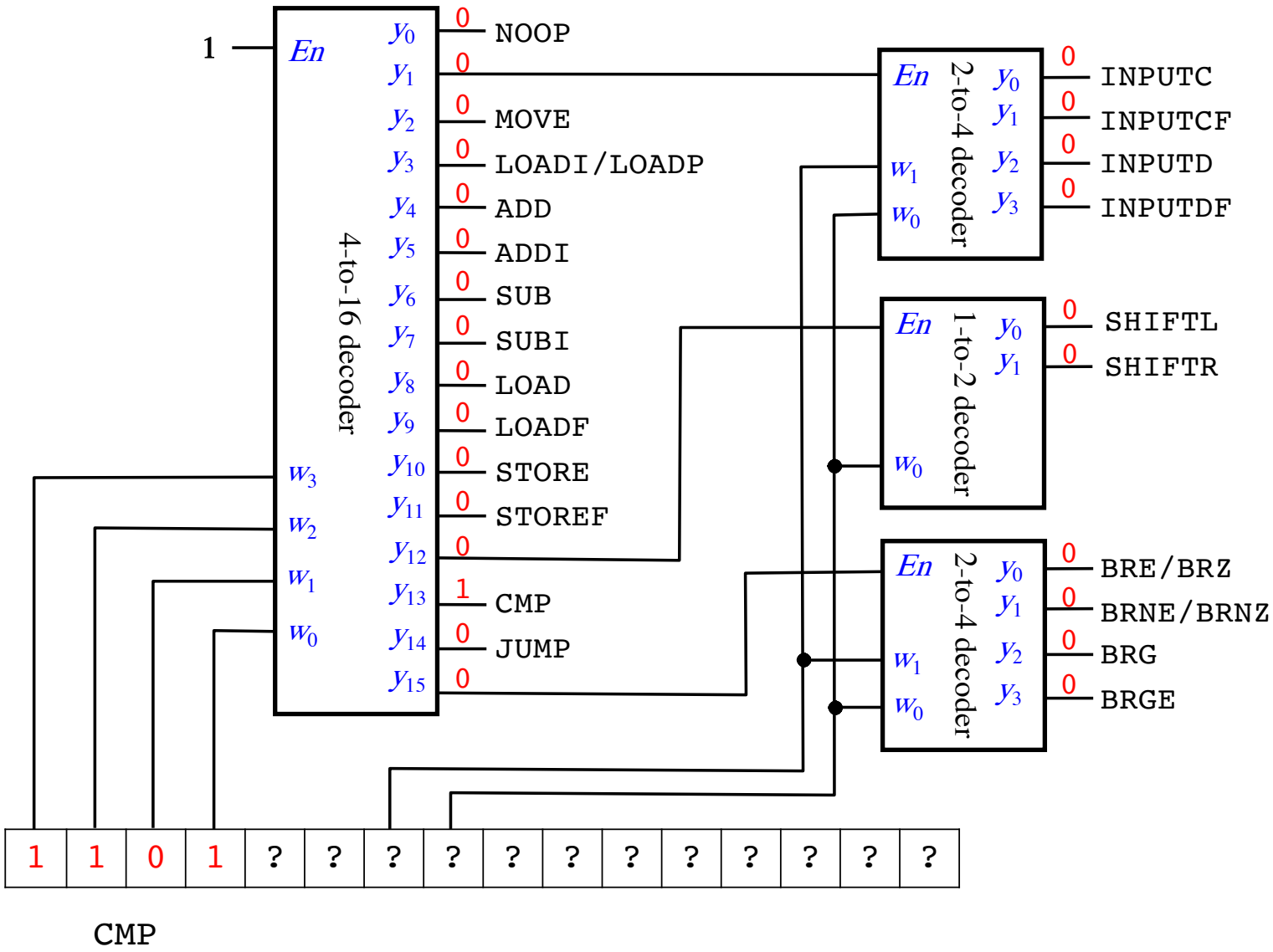

1
En
4-to-16 decoder
y_0
y_1
y_2
y_3
y_4
y_5
y_6
y_7
y_8
y_9
y_{10}
y_{11}
y_{12}
y_{13}
y_{14}
y_{15}
w_3
w_2
w_1
w_0
0 NOOP
0
0 MOVE
0 LOADI/LOADP
0 ADD
0 ADDI
0 SUB
0 SUBI
0 LOAD
1 LOADF
0 STORE
0 STOREF
0
0 CMP
0 JUMP
0
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
0 INPUTC
0 INPUTCF
0 INPUTD
0 INPUTDF
En
1-to-2 decoder
y_0
y_1
w_0
0 SHIFTL
0 SHIFTR
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
0 BRE/BRZ
0 BRNE/BRNZ
0 BRG
0 BRGE
1 0 0 1 ? ? ? ? ? ? ? ? ? ? ? ?
LOADF

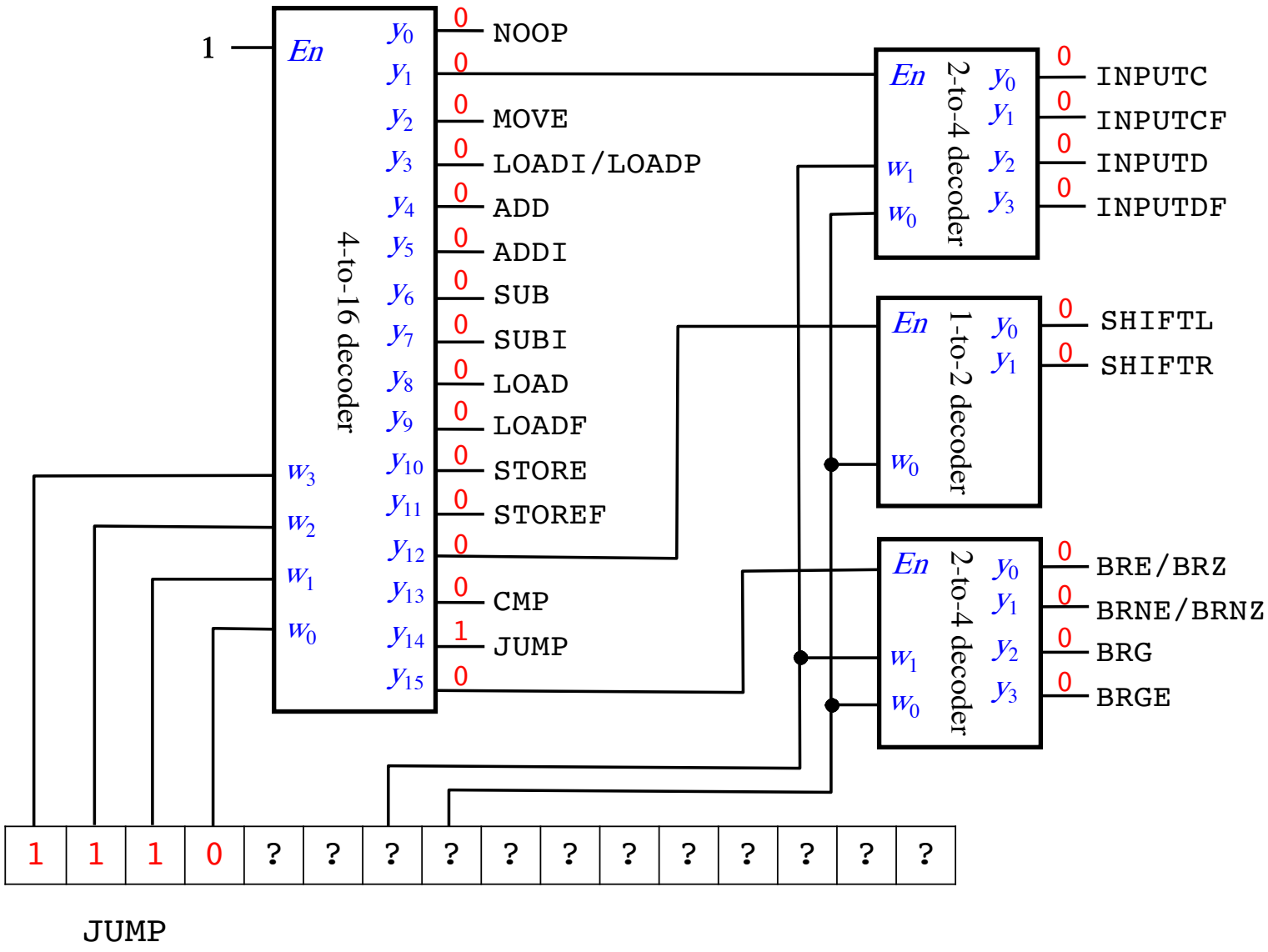

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 0 NOOP
y_1 0
y_2 0 MOVE
y_3 0 LOADI/LOADP
y_4 0 ADD
y_5 0 ADDI
y_6 0 SUB
y_7 0 SUBI
y_8 0 LOAD
y_9 0 LOADF
y_{10} 1 STORE
y_{11} 0 STOREF
y_{12} 0
y_{13} 0 CMP
y_{14} 0 JUMP
y_{15} 0
En
2-to-4 decoder
w_1
w_0
y_0 0 INPUTC
y_1 0 INPUTCF
y_2 0 INPUTD
y_3 0 INPUTDF
En
1-to-2 decoder
w_0
y_0 0 SHIFTL
y_1 0 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 0 BRE/BRZ
y_1 0 BRNE/BRNZ
y_2 0 BRG
y_3 0 BRGE
1 0 1 0 ? ? ? ? ? ? ? ? ? ? ? ?
STORE

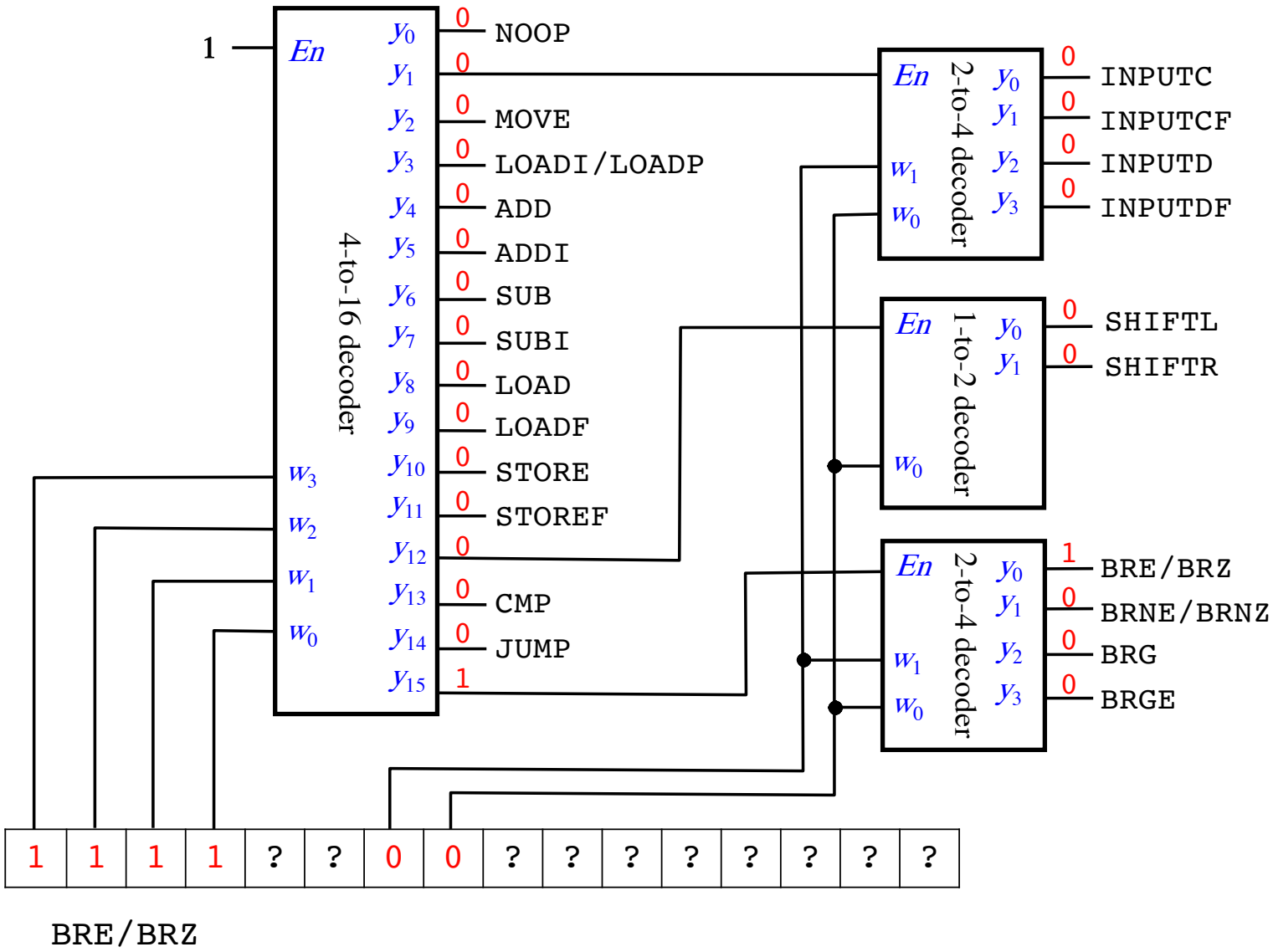

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 0 NOOP
y_1 0
y_2 0 MOVE
y_3 0 LOADI/LOADP
y_4 0 ADD
y_5 0 ADDI
y_6 0 SUB
y_7 0 SUBI
y_8 0 LOAD
y_9 0 LOADF
y_{10} 0 STORE
y_{11} 1 STOREF
y_{12} 0
y_{13} 0 CMP
y_{14} 0 JUMP
y_{15} 0
En
2-to-4 decoder
w_1
w_0
y_0 0 INPUTC
y_1 0 INPUTCF
y_2 0 INPUTD
y_3 0 INPUTDF
En
1-to-2 decoder
w_0
y_0 0 SHIFTL
y_1 0 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 0 BRE/BRZ
y_1 0 BRNE/BRNZ
y_2 0 BRG
y_3 0 BRGE
1 0 1 1 ? ? ? ? ? ? ? ? ? ? ? ?
STOREF

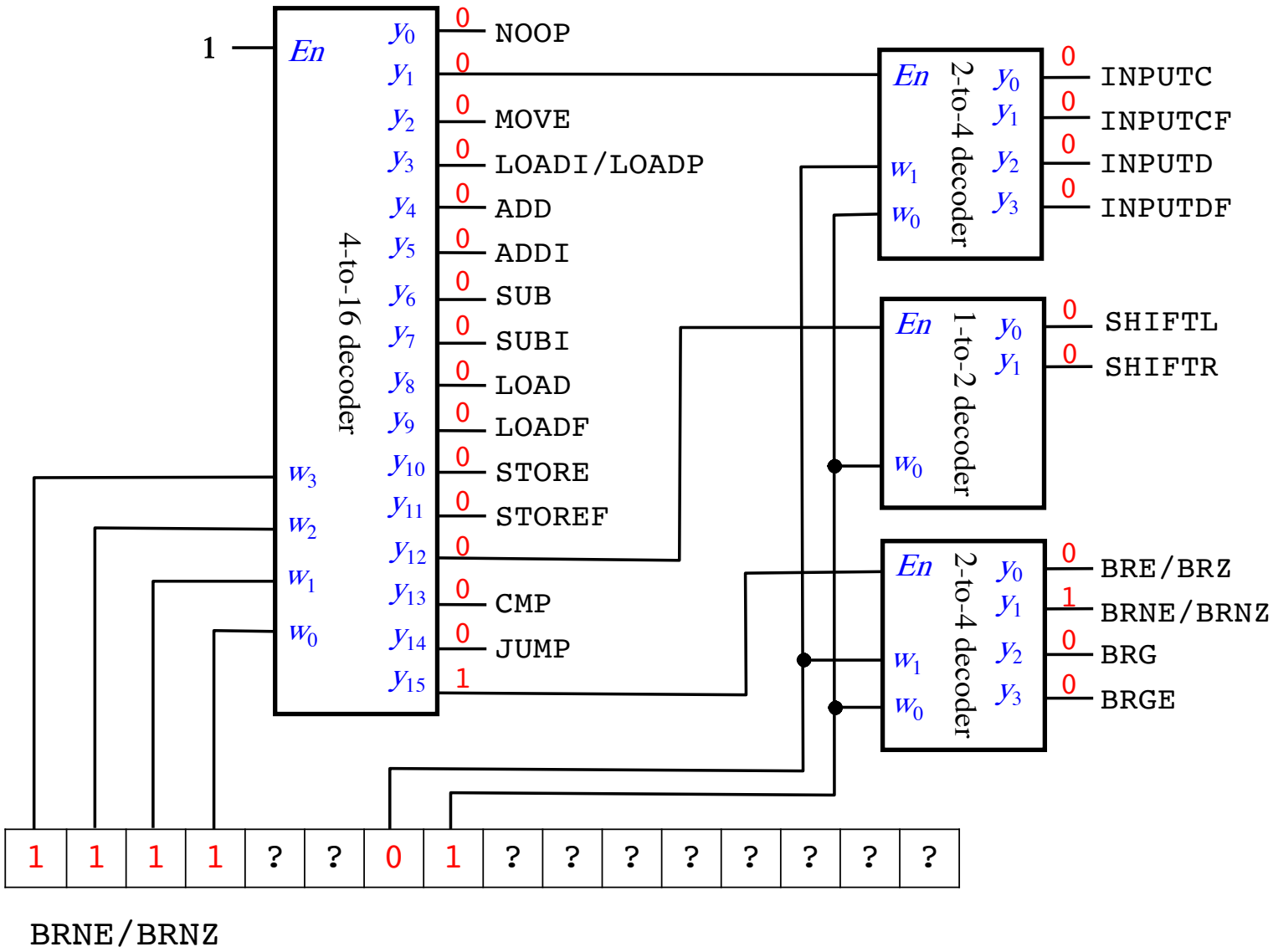

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 0 NOOP
y_1 0
y_2 0 MOVE
y_3 0 LOADI/LOADP
y_4 0 ADD
y_5 0 ADDI
y_6 0 SUB
y_7 0 SUBI
y_8 0 LOAD
y_9 0 LOADF
y_{10} 0 STORE
y_{11} 0 STOREF
y_{12} 1
y_{13} 0 CMP
y_{14} 0 JUMP
y_{15} 0
En
2-to-4 decoder
w_1
w_0
y_0 0 INPUTC
y_1 0 INPUTCF
y_2 0 INPUTD
y_3 0 INPUTDF
En
1-to-2 decoder
w_0
y_0 1 SHIFTL
y_1 0 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 0 BRE/BRZ
y_1 0 BRNE/BRNZ
y_2 0 BRG
y_3 0 BRGE
1 1 0 0 ? ? ? 0 ? ? ? ? ? ? ? ?
SHIFTL

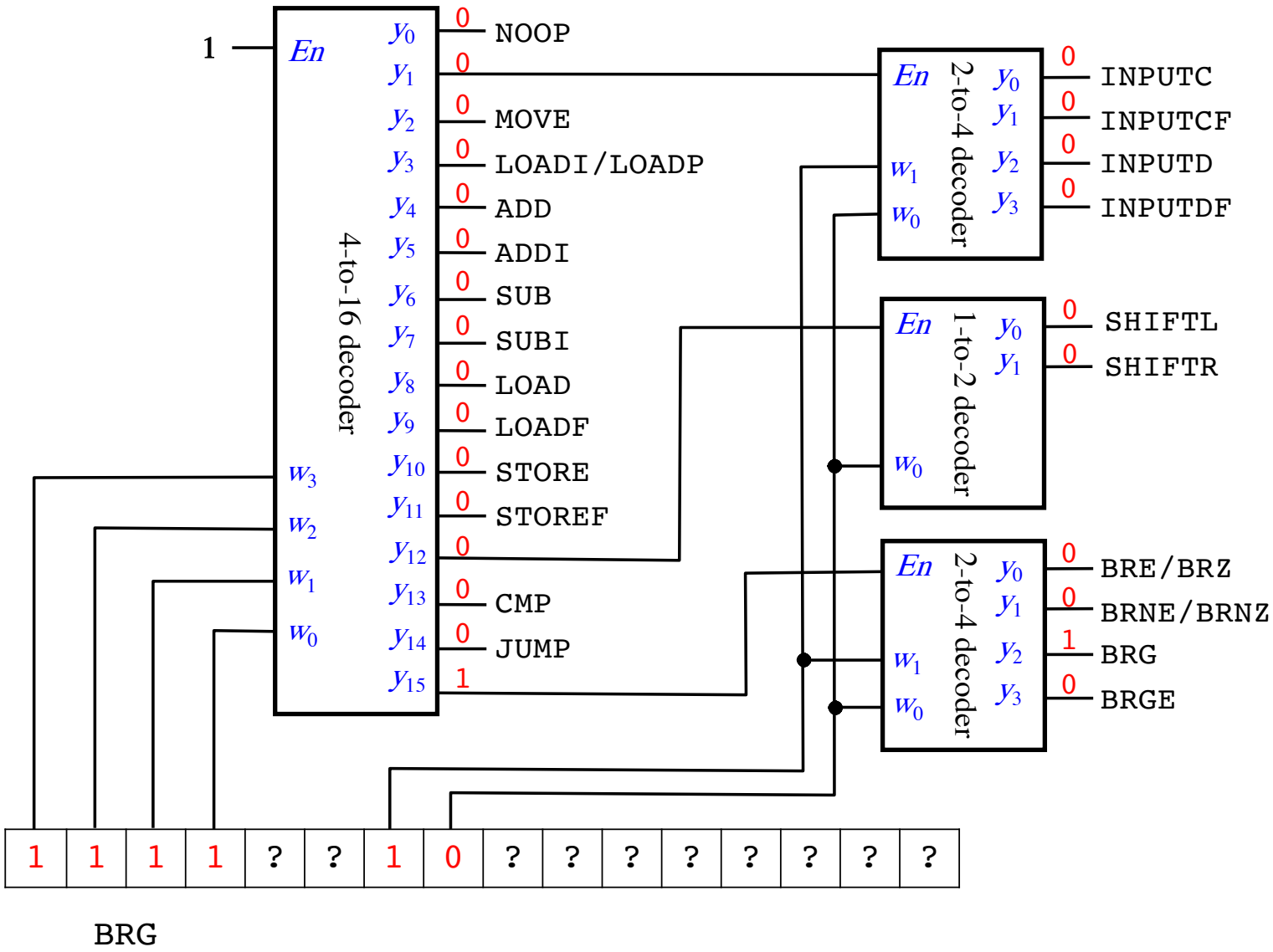

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 0 NOOP
y_1 0
y_2 0 MOVE
y_3 0 LOADI/LOADP
y_4 0 ADD
y_5 0 ADDI
y_6 0 SUB
y_7 0 SUBI
y_8 0 LOAD
y_9 0 LOADF
y_{10} 0 STORE
y_{11} 0 STOREF
y_{12} 1
y_{13} 0 CMP
y_{14} 0 JUMP
y_{15} 0
En
2-to-4 decoder
w_1
w_0
y_0 0 INPUTC
y_1 0 INPUTCF
y_2 0 INPUTD
y_3 0 INPUTDF
En
1-to-2 decoder
w_0
y_0 0 SHIFTL
y_1 1 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 0 BRE/BRZ
y_1 0 BRNE/BRNZ
y_2 0 BRG
y_3 0 BRGE
1 1 0 0 ? ? ? 1 ? ? ? ? ? ? ? ?
SHIFTR

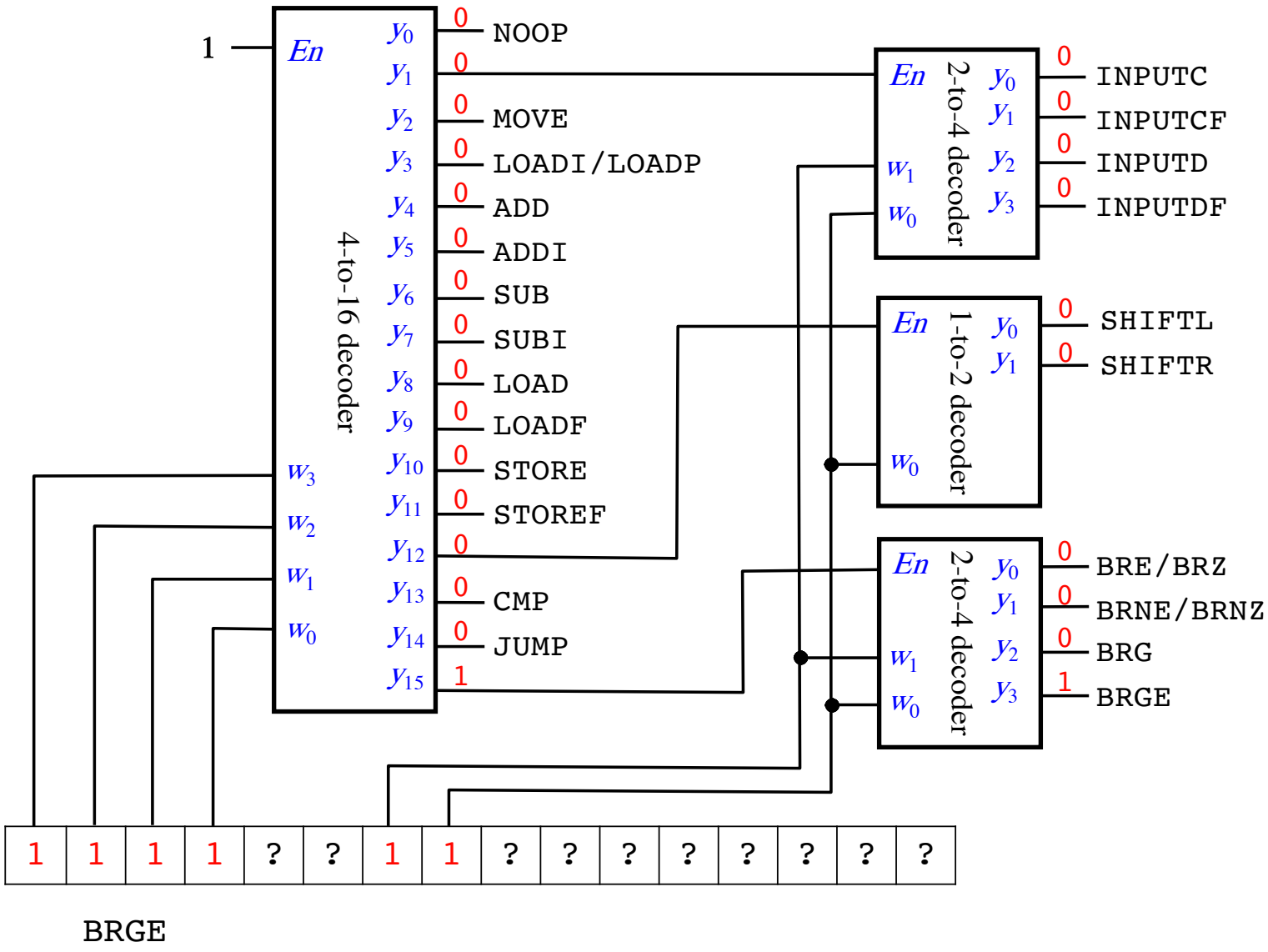

1
En
4-to-16 decoder
y_0
y_1
y_2
y_3
y_4
y_5
y_6
y_7
y_8
y_9
y_{10}
y_{11}
y_{12}
y_{13}
y_{14}
y_{15}
w_3
w_2
w_1
w_0
0 NOOP
0
0 MOVE
0 LOADI/LOADP
0 ADD
0 ADDI
0 SUB
0 SUBI
0 LOAD
0 LOADF
0 STORE
0 STOREF
0
1 CMP
0 JUMP
0
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
0 INPUTC
0 INPUTCF
0 INPUTD
0 INPUTDF
En
1-to-2 decoder
y_0
y_1
w_0
0 SHIFTL
0 SHIFTR
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
0 BRE/BRZ
0 BRNE/BRNZ
0 BRG
0 BRGE
1 1 0 1 ? ? ? ? ? ? ? ? ? ? ? ?
CMP

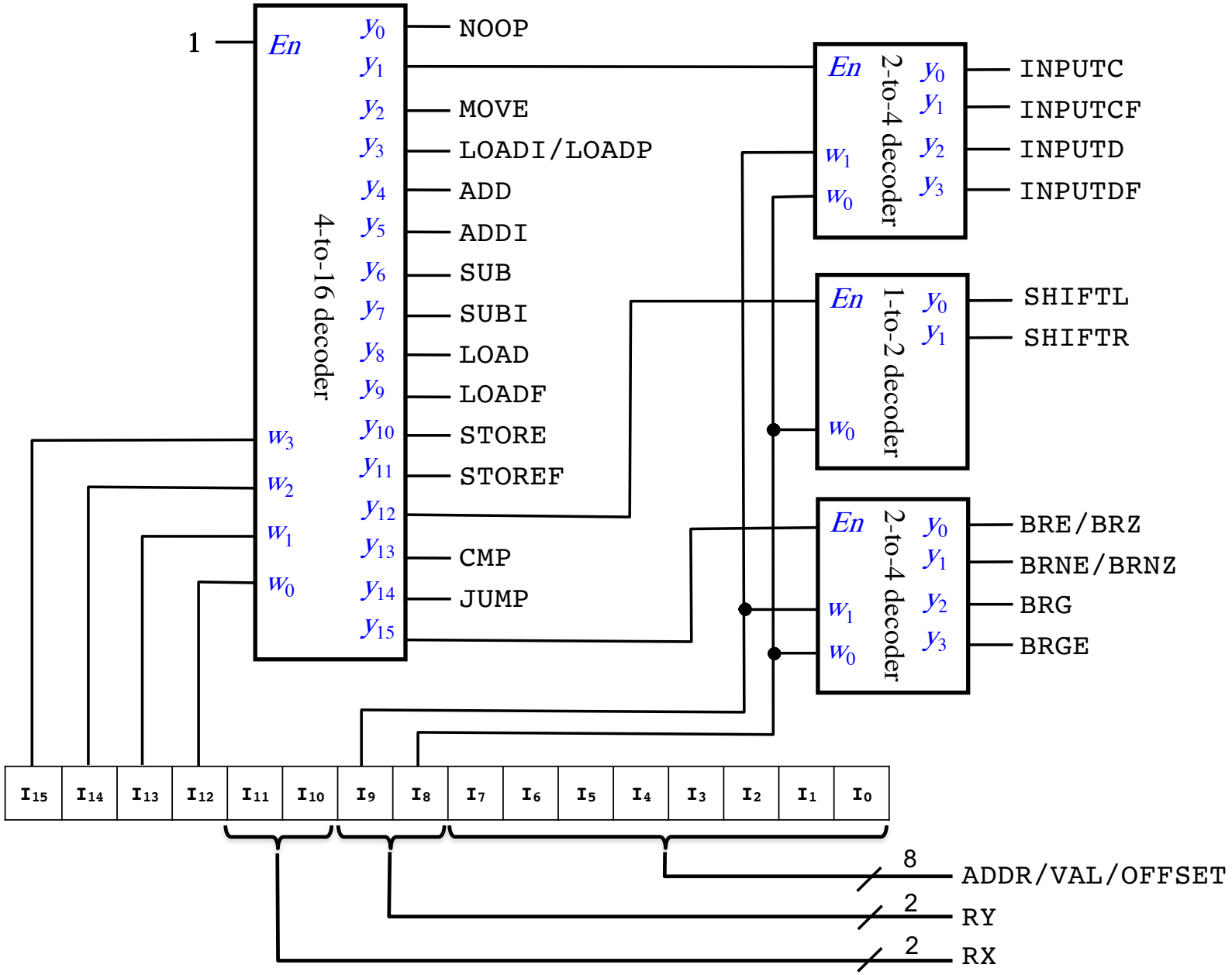

1
En
4-to-16 decoder
y_0
y_1
y_2
y_3
y_4
y_5
y_6
y_7
y_8
y_9
y_{10}
y_{11}
y_{12}
y_{13}
y_{14}
y_{15}
w_3
w_2
w_1
w_0
0 NOOP
0
0 MOVE
0 LOADI/LOADP
0 ADD
0 ADDI
0 SUB
0 SUBI
0 LOAD
0 LOADF
0 STORE
0 STOREF
0
0 CMP
1 JUMP
0
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
0 INPUTC
0 INPUTCF
0 INPUTD
0 INPUTDF
En
1-to-2 decoder
y_0
y_1
w_0
0 SHIFTL
0 SHIFTR
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
0 BRE/BRZ
0 BRNE/BRNZ
0 BRG
0 BRGE
1 1 1 0 ? ? ? ? ? ? ? ? ? ? ? ?
JUMP

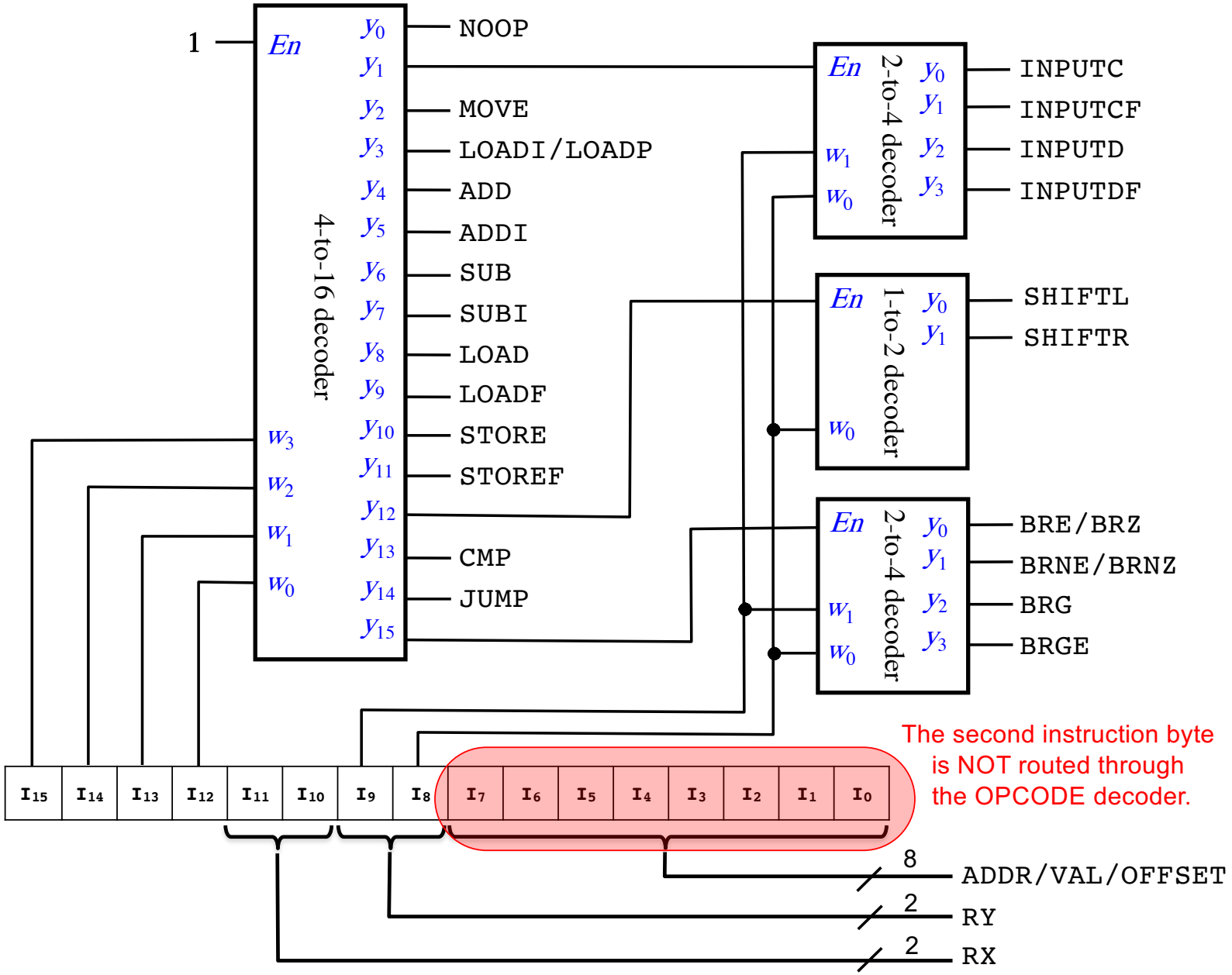

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 0 NOOP
y_1 0
y_2 0 MOVE
y_3 0 LOADI/LOADP
y_4 0 ADD
y_5 0 ADDI
y_6 0 SUB
y_7 0 SUBI
y_8 0 LOAD
y_9 0 LOADF
y_{10} 0 STORE
y_{11} 0 STOREF
y_{12} 0
y_{13} 0 CMP
y_{14} 0 JUMP
y_{15} 1
En
2-to-4 decoder
w_1
w_0
y_0 0 INPUTC
y_1 0 INPUTCF
y_2 0 INPUTD
y_3 0 INPUTDF
En
1-to-2 decoder
w_0
y_0 0 SHIFTL
y_1 0 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 1 BRE/BRZ
y_1 0 BRNE/BRNZ
y_2 0 BRG
y_3 0 BRGE
1 1 1 1 ? ? 0 0 ? ? ? ? ? ? ? ?
BRE/BRZ

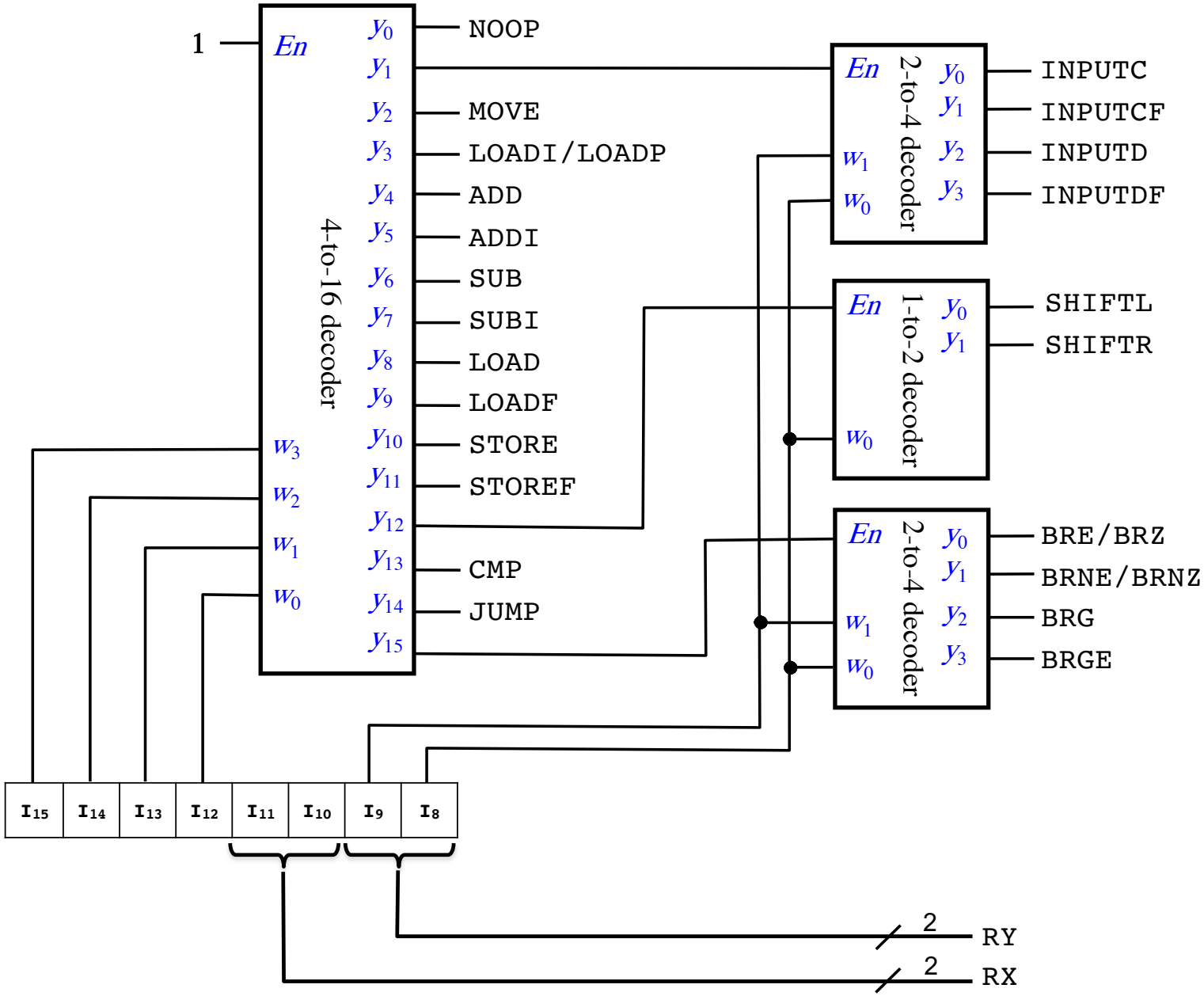

1
En
4-to-16 decoder
y_0
y_1
y_2
y_3
y_4
y_5
y_6
y_7
y_8
y_9
y_{10}
y_{11}
y_{12}
y_{13}
y_{14}
y_{15}
w_3
w_2
w_1
w_0
0 NOOP
0
0 MOVE
0 LOADI/LOADP
0 ADD
0 ADDI
0 SUB
0 SUBI
0 LOAD
0 LOADF
0 STORE
0 STOREF
0
0 CMP
0 JUMP
1
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
0 INPUTC
0 INPUTCF
0 INPUTD
0 INPUTDF
En
1-to-2 decoder
y_0
y_1
w_0
0 SHIFTL
0 SHIFTR
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
0 BRE/BRZ
1 BRNE/BRNZ
0 BRG
0 BRGE
1 1 1 1 ? ? 0 1 ? ? ? ? ? ? ? ?
BRNE/BRNZ

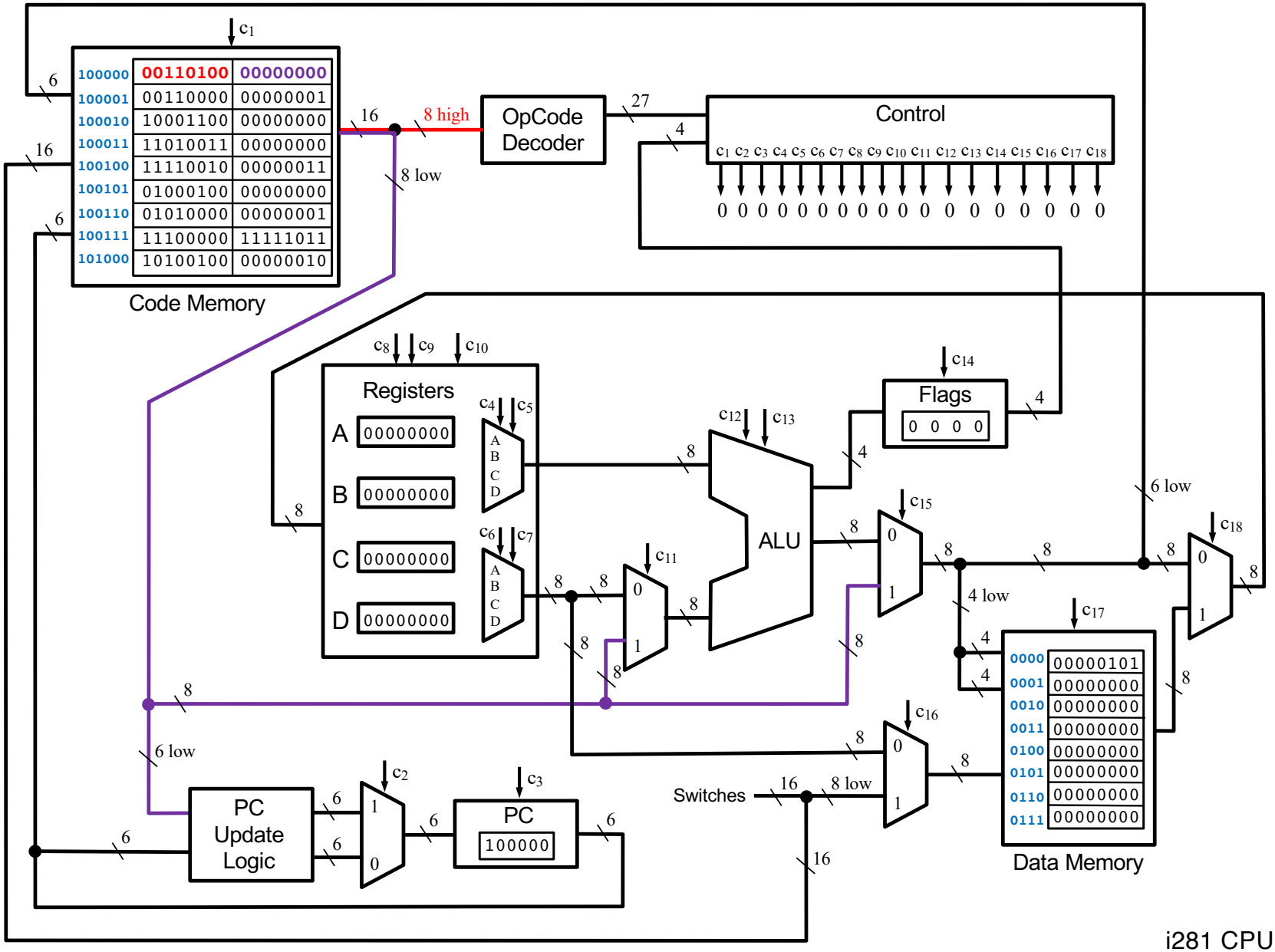

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0
y_1
y_2
y_3
y_4
y_5
y_6
y_7
y_8
y_9
y_{10}
y_{11}
y_{12}
y_{13}
y_{14}
y_{15}
0 NOOP
0
0 MOVE
0 LOADI/LOADP
0 ADD
0 ADDI
0 SUB
0 SUBI
0 LOAD
0 LOADF
0 STORE
0 STOREF
0
0 CMP
0 JUMP
1
En
2-to-4 decoder
w_1
w_0
0 INPUTC
0 INPUTCF
0 INPUTD
0 INPUTDF
En
1-to-2 decoder
w_0
0 SHIFTL
0 SHIFTR
En
2-to-4 decoder
w_1
w_0
0 BRE/BRZ
0 BRNE/BRNZ
1 BRG
0 BRGE
1 1 1 1 ? ? 1 0 ? ? ? ? ? ? ? ?
BRG

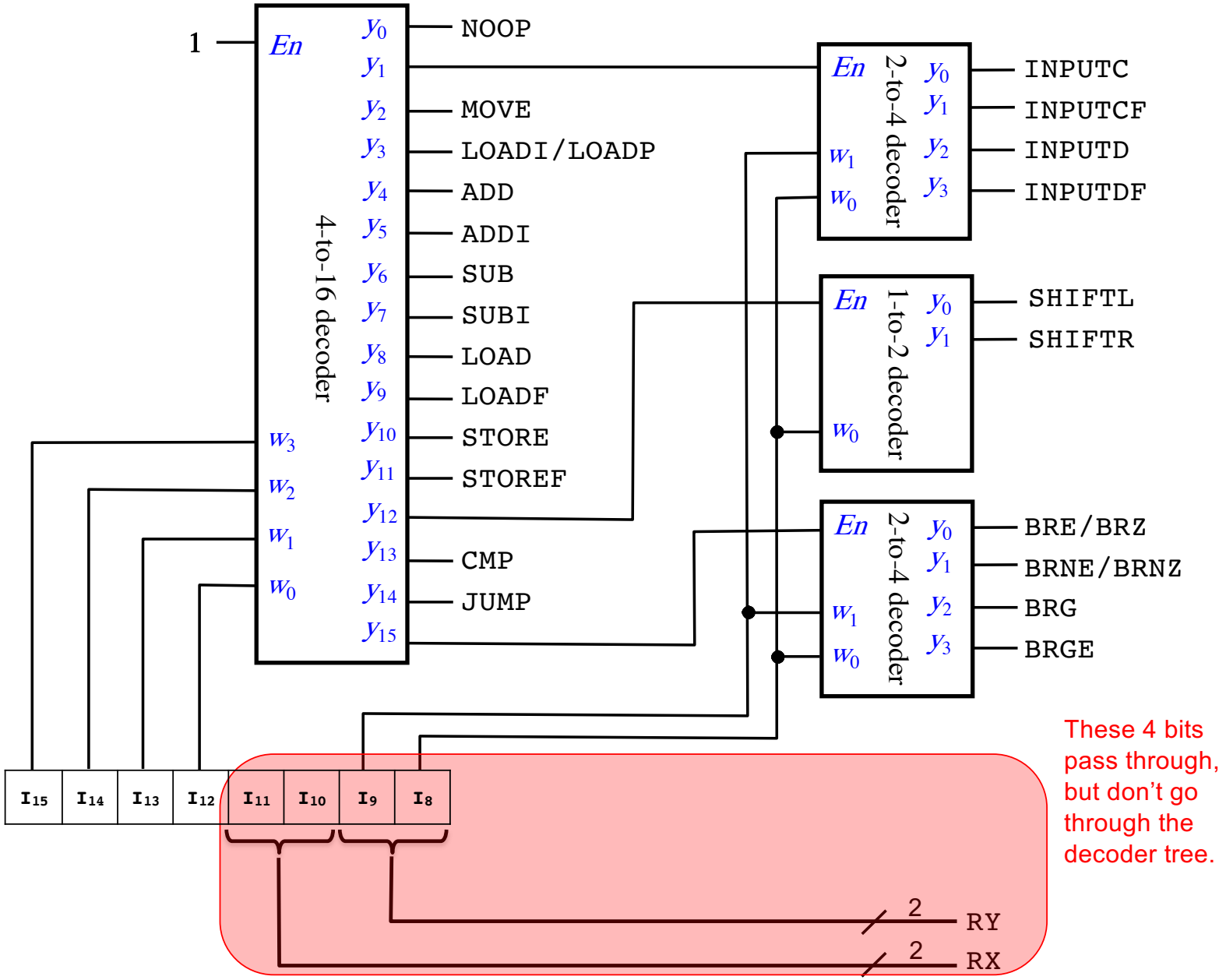

1
En
4-to-16 decoder
y_0
y_1
y_2
y_3
y_4
y_5
y_6
y_7
y_8
y_9
y_{10}
y_{11}
y_{12}
y_{13}
y_{14}
y_{15}
w_3
w_2
w_1
w_0
0 NOOP
0
0 MOVE
0 LOADI/LOADP
0 ADD
0 ADDI
0 SUB
0 SUBI
0 LOAD
0 LOADF
0 STORE
0 STOREF
0
0 CMP
0 JUMP
1
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
0 INPUTC
0 INPUTCF
0 INPUTD
0 INPUTDF
En
1-to-2 decoder
y_0
y_1
w_0
0 SHIFTL
0 SHIFTR
En
2-to-4 decoder
y_0
y_1
y_2
y_3
w_1
w_0
0 BRE/BRZ
0 BRNE/BRNZ
0 BRG
1 BRGE
1 1 1 1 ? ? 1 1 ? ? ? ? ? ? ? ?
BRGE

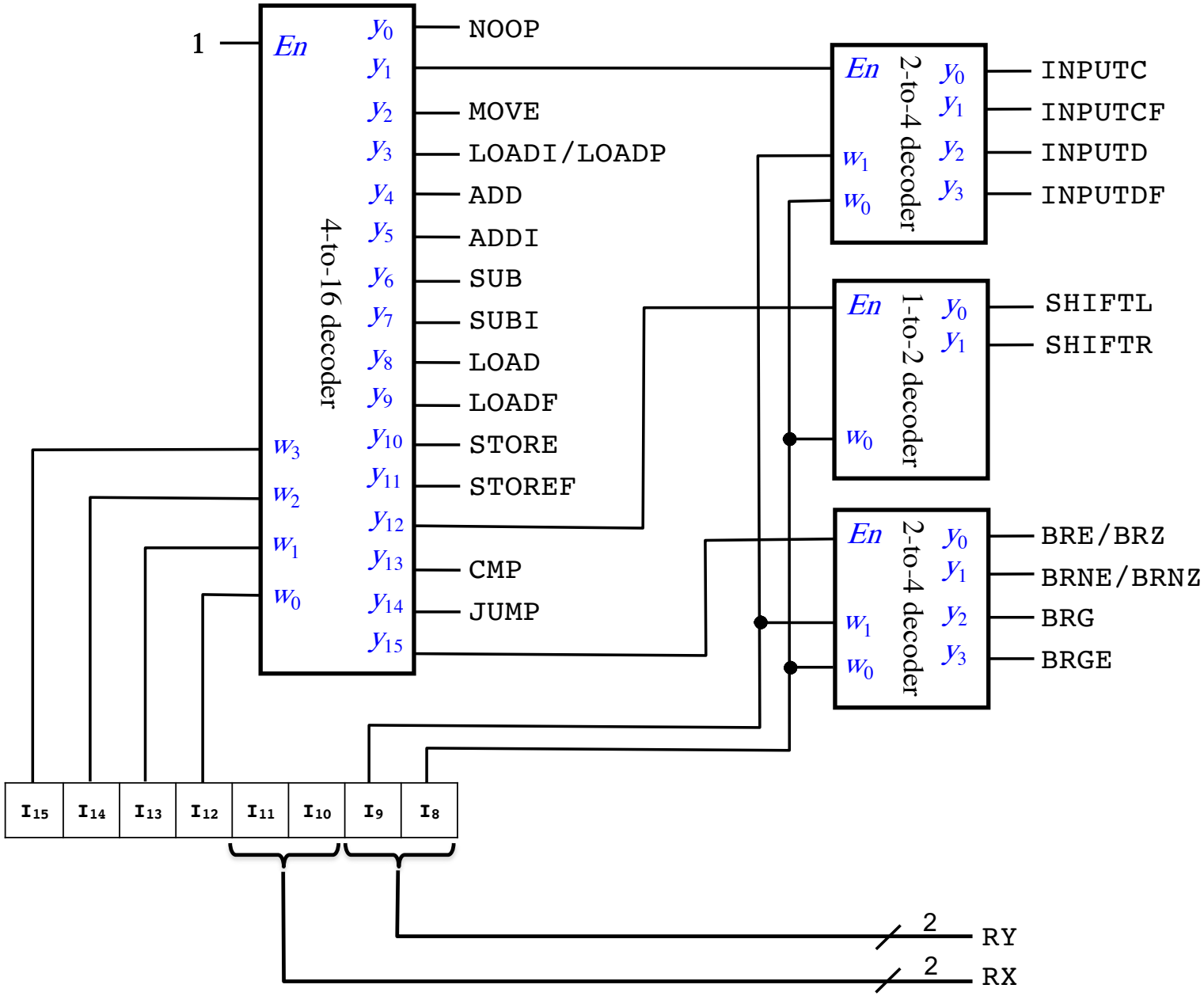

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 NOOP
y_1
y_2 MOVE
y_3 LOADI/LOADP
y_4 ADD
y_5 ADDI
y_6 SUB
y_7 SUBI
y_8 LOAD
y_9 LOADF
y_{10} STORE
y_{11} STOREF
y_{12}
y_{13} CMP
y_{14} JUMP
y_{15}
En
2-to-4 decoder
w_1
w_0
y_0 INPUTC
y_1 INPUTCF
y_2 INPUTD
y_3 INPUTDF
En
1-to-2 decoder
w_0
y_0 SHIFTL
y_1 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 BRE/BRZ
y_1 BRNE/BRNZ
y_2 BRG
y_3 BRGE
I15 I14 I13 I12 I11 I10 I9 I8 I7 I6 I5 I4 I3 I2 I1 I0
8 ADDR/VAL/OFFSET
2 RY
2 RX

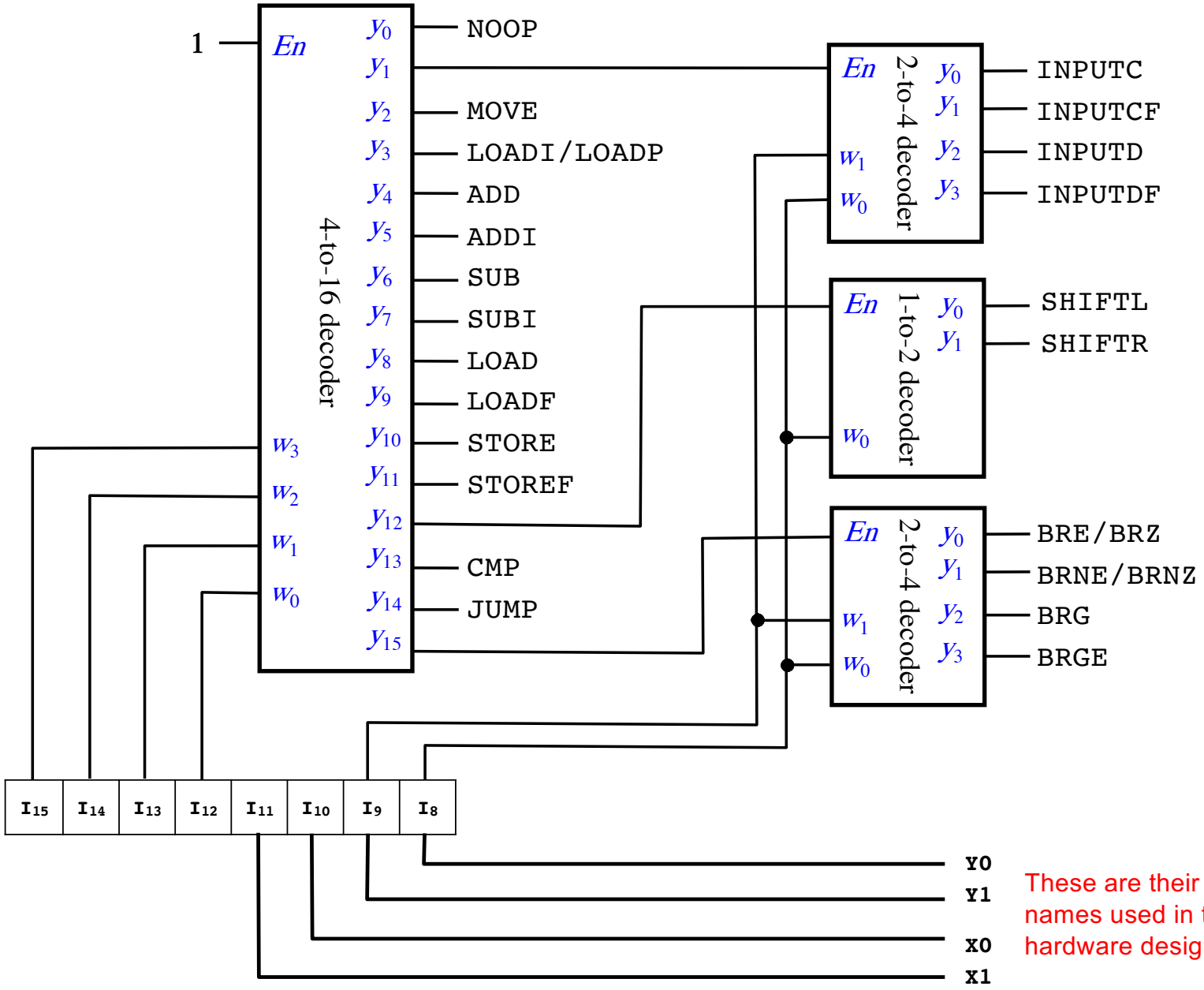

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 NOOP
y_1
y_2 MOVE
y_3 LOADI/LOADP
y_4 ADD
y_5 ADDI
y_6 SUB
y_7 SUBI
y_8 LOAD
y_9 LOADF
y_{10} STORE
y_{11} STOREF
y_{12}
y_{13} CMP
y_{14} JUMP
y_{15}
En
2-to-4 decoder
w_1
w_0
y_0 INPUTC
y_1 INPUTCF
y_2 INPUTD
y_3 INPUTDF
En
1-to-2 decoder
w_0
y_0 SHIFTL
y_1 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 BRE/BRZ
y_1 BRNE/BRNZ
y_2 BRG
y_3 BRGE
I15 I14 I13 I12 I11 I10 I9 I8 I7 I6 I5 I4 I3 I2 I1 I0
The second instruction byte is NOT routed through the OPCODE decoder.
8 ADDR/VAL/OFFSET
2 RY
2 RX

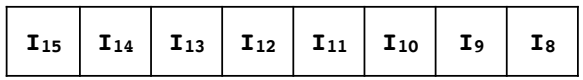

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 NOOP
y_1
y_2 MOVE
y_3 LOADI/LOADP
y_4 ADD
y_5 ADDI
y_6 SUB
y_7 SUBI
y_8 LOAD
y_9 LOADF
y_{10} STORE
y_{11} STOREF
y_{12}
y_{13} CMP
y_{14} JUMP
y_{15}
En
2-to-4 decoder
w_1
w_0
y_0 INPUTC
y_1 INPUTCF
y_2 INPUTD
y_3 INPUTDF
En
1-to-2 decoder
w_0
y_0 SHIFTL
y_1 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 BRE/BRZ
y_1 BRNE/BRNZ
y_2 BRG
y_3 BRGE
I15 I14 I13 I12 I11 I10 I9 I8
2 RY
2 RX

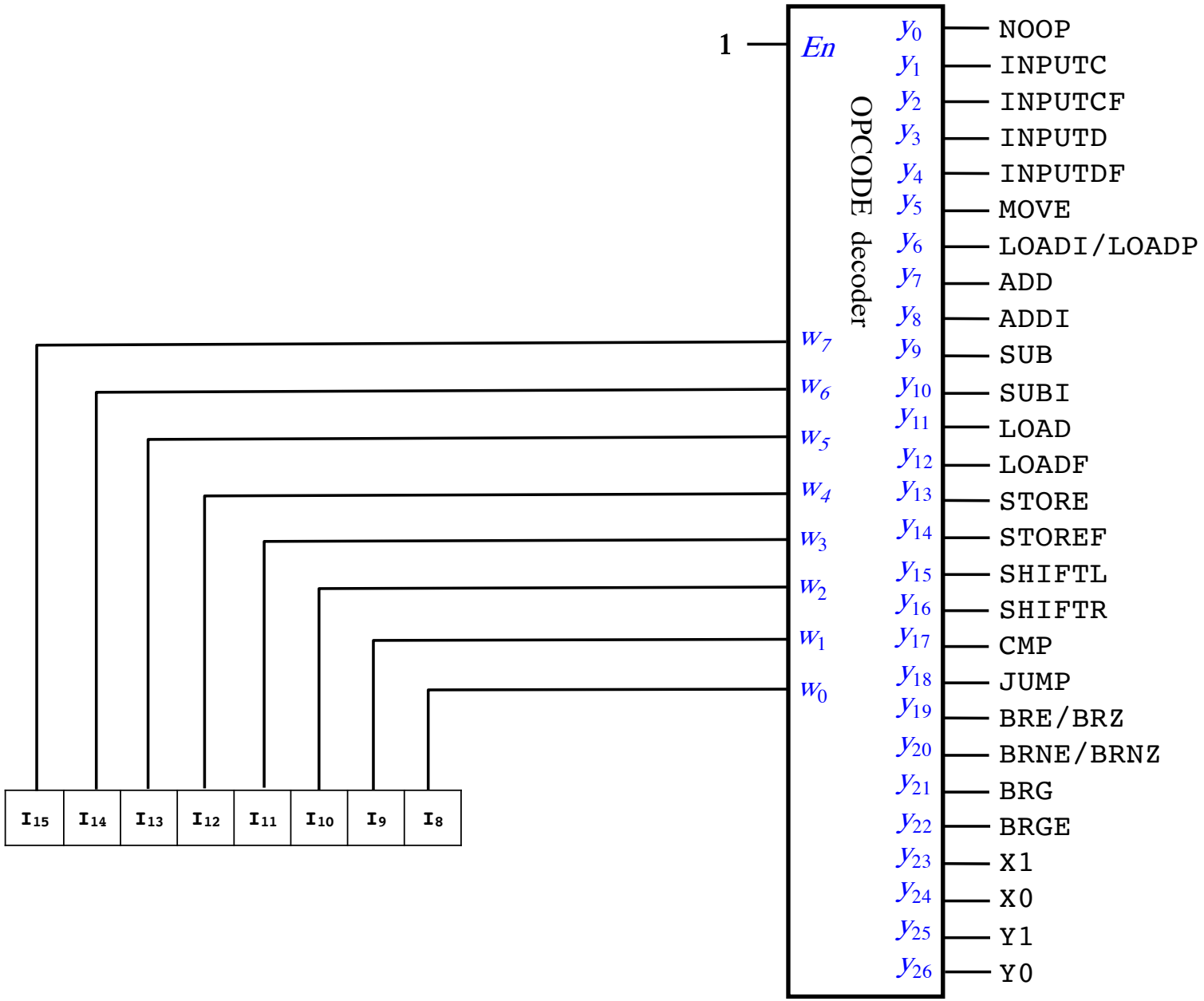

c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
6
16
6
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
8
c12 c13
ALU
c14
Flags
0 0 0 0
4
c15
c11
c16
c17
c18
6 low
4 low
8 low
Switches
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
c2
c3
PC Update Logic
PC
100000
i281 CPU

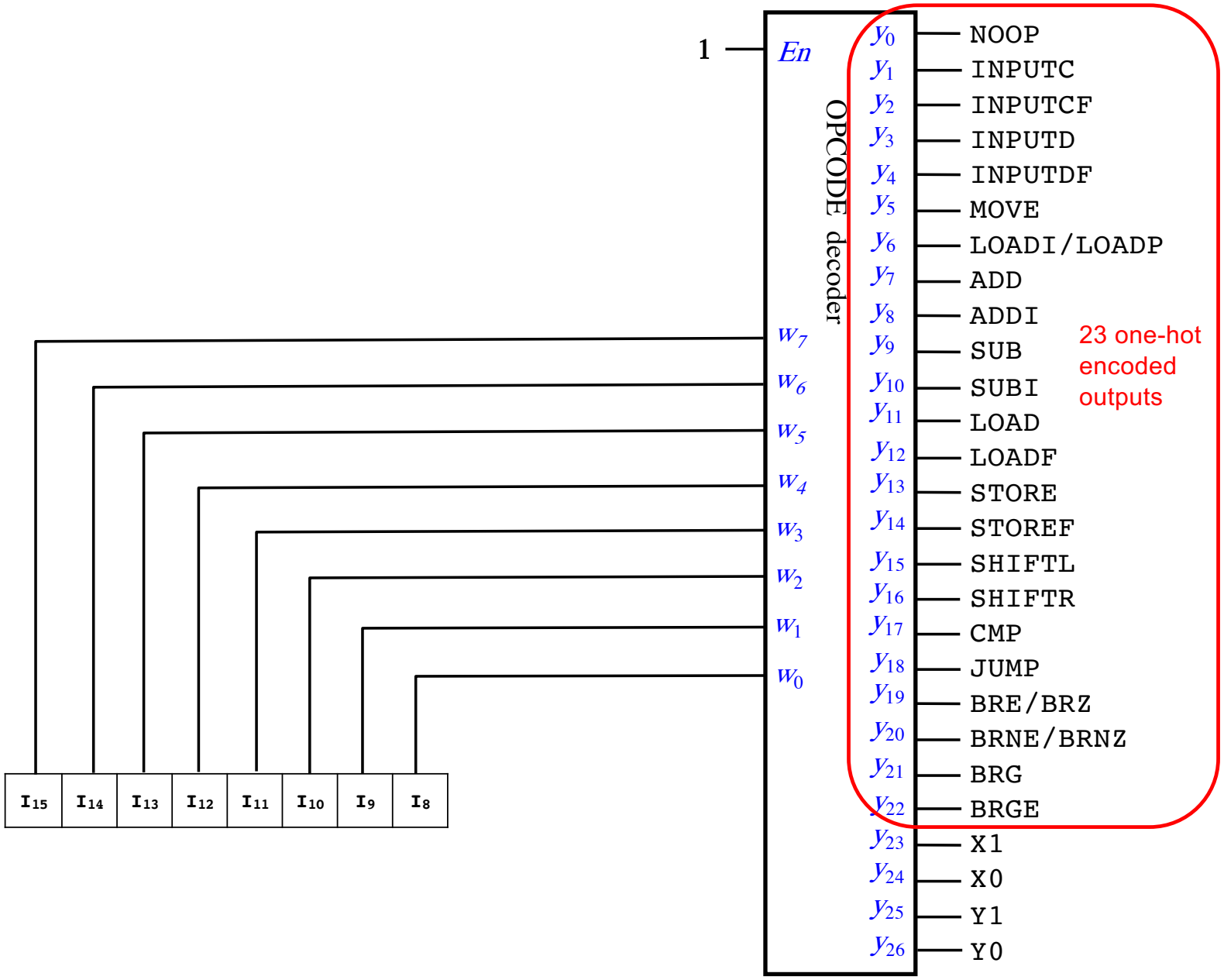

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 NOOP
y_1
y_2 MOVE
y_3 LOADI/LOADP
y_4 ADD
y_5 ADDI
y_6 SUB
y_7 SUBI
y_8 LOAD
y_9 LOADF
y_{10} STORE
y_{11} STOREF
y_{12}
y_{13} CMP
y_{14} JUMP
y_{15}
En
2-to-4 decoder
w_1
w_0
y_0 INPUTC
y_1 INPUTCF
y_2 INPUTD
y_3 INPUTDF
En
1-to-2 decoder
w_0
y_0 SHIFTL
y_1 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 BRE/BRZ
y_1 BRNE/BRNZ
y_2 BRG
y_3 BRGE
I15 I14 I13 I12 I11 I10 I9 I8
2 RY
2 RX
These 4 bits pass through, but don't go through the decoder tree.

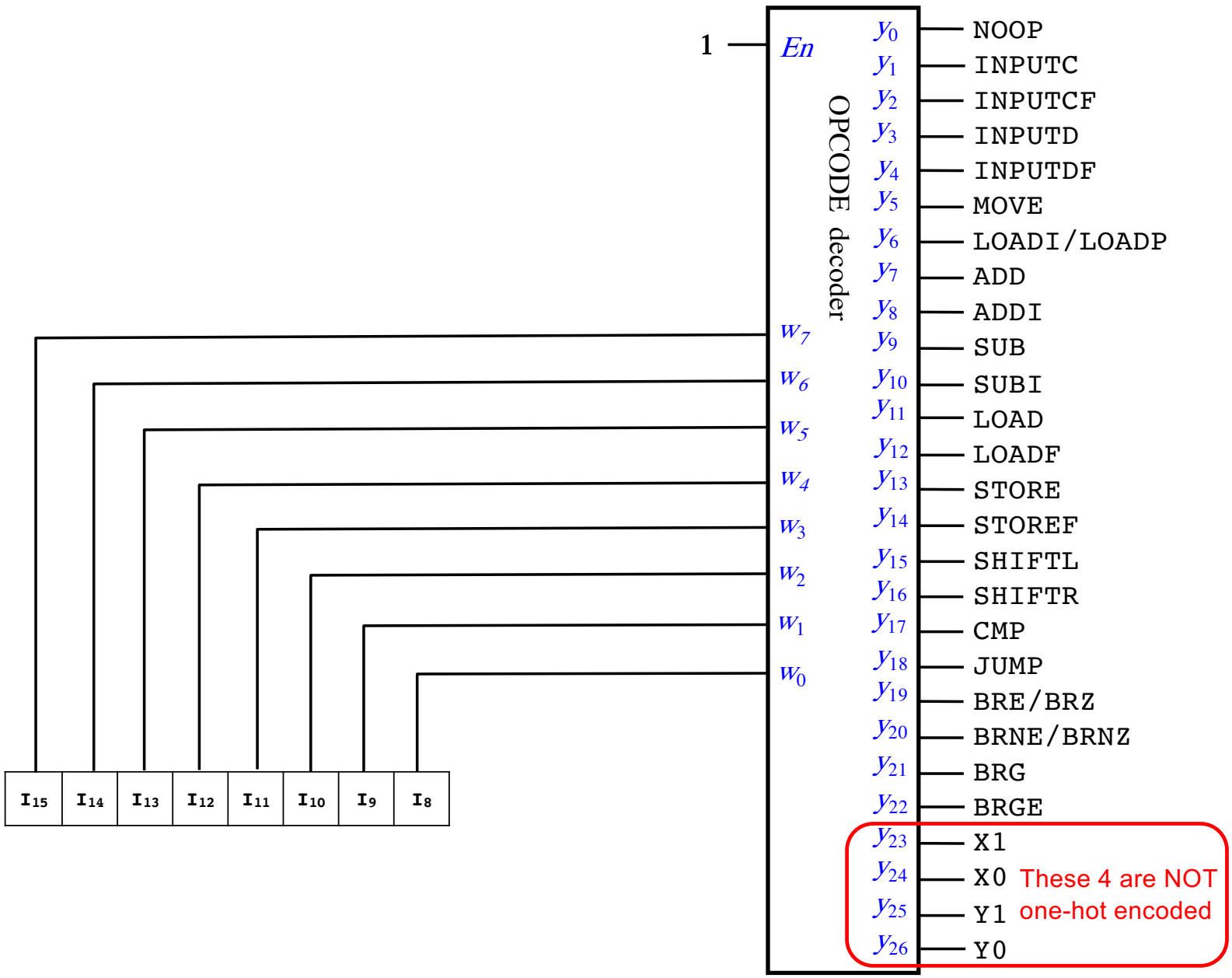

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 NOOP
y_1
y_2 MOVE
y_3 LOADI/LOADP
y_4 ADD
y_5 ADDI
y_6 SUB
y_7 SUBI
y_8 LOAD
y_9 LOADF
y_{10} STORE
y_{11} STOREF
y_{12}
y_{13} CMP
y_{14} JUMP
y_{15}
En
2-to-4 decoder
w_1
w_0
y_0 INPUTC
y_1 INPUTCF
y_2 INPUTD
y_3 INPUTDF
En
1-to-2 decoder
w_0
y_0 SHIFTL
y_1 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 BRE/BRZ
y_1 BRNE/BRNZ
y_2 BRG
y_3 BRGE
I15 I14 I13 I12 I11 I10 I9 I8
2 RY
2 RX

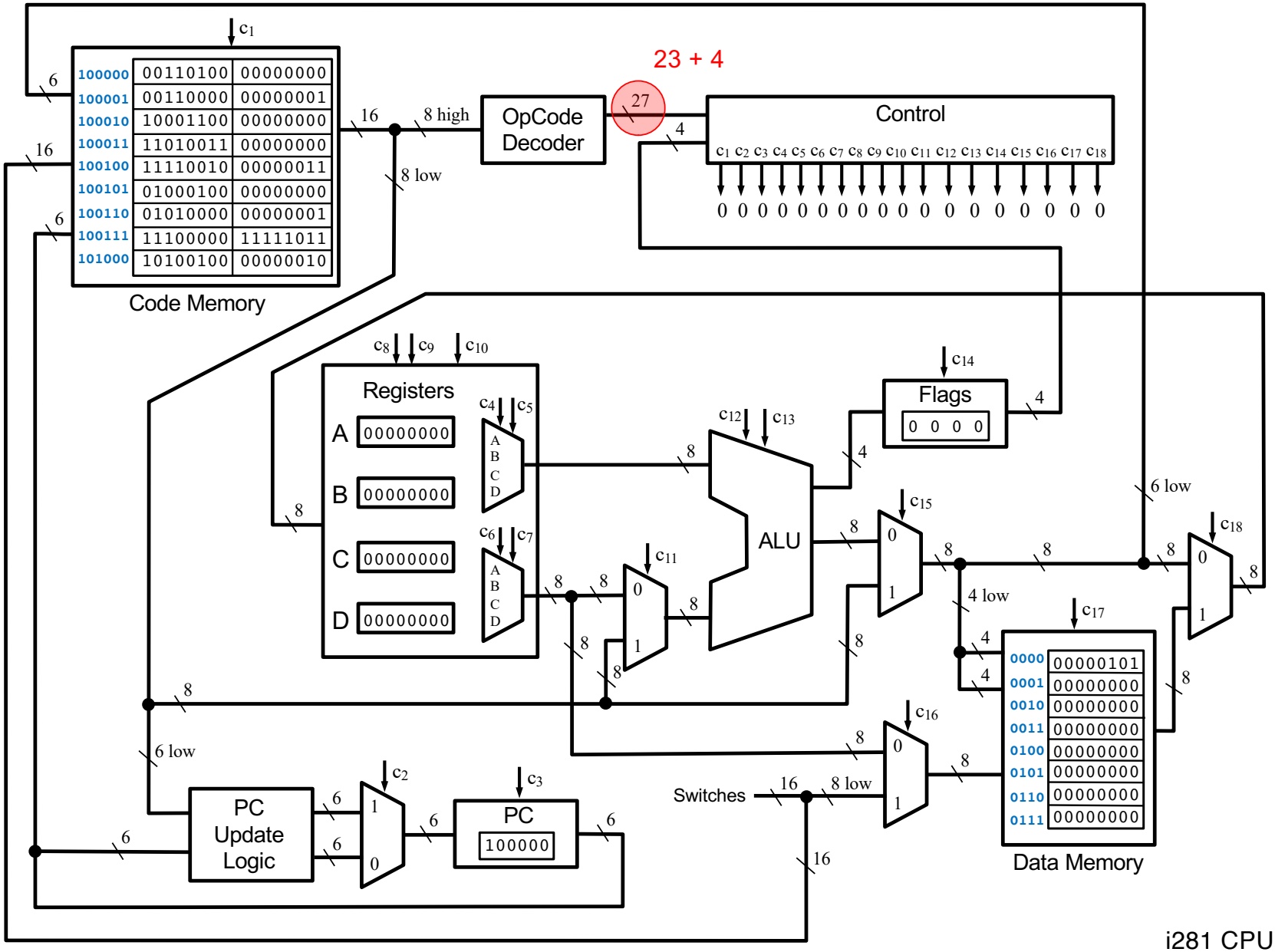

1
En
4-to-16 decoder
w_3
w_2
w_1
w_0
y_0 NOOP
y_1
y_2 MOVE
y_3 LOADI/LOADP
y_4 ADD
y_5 ADDI
y_6 SUB
y_7 SUBI
y_8 LOAD
y_9 LOADF
y_{10} STORE
y_{11} STOREF
y_{12}
y_{13} CMP
y_{14} JUMP
y_{15}
En
2-to-4 decoder
w_1
w_0
y_0 INPUTC
y_1 INPUTCF
y_2 INPUTD
y_3 INPUTDF
En
1-to-2 decoder
w_0
y_0 SHIFTL
y_1 SHIFTR
En
2-to-4 decoder
w_1
w_0
y_0 BRE/BRZ
y_1 BRNE/BRNZ
y_2 BRG
y_3 BRGE
I15 I14 I13 I12 I11 I10 I9 I8
Y0
Y1
X0
X1
These are their names used in the hardware design.


I15
I14
I13
I12
I11
I10
I9
I8


1
En
OPCODE decoder
w_7
w_6
w_5
w_4
w_3
w_2
w_1
w_0
y_0 NOOP
y_1 INPUTC
y_2 INPUTCF
y_3 INPUTD
y_4 INPUTDF
y_5 MOVE
y_6 LOADI/LOADP
y_7 ADD
y_8 ADDI
y_9 SUB
y_{10} SUBI
y_{11} LOAD
y_{12} LOADF
y_{13} STORE
y_{14} STOREF
y_{15} SHIFTL
y_{16} SHIFTR
y_{17} CMP
y_{18} JUMP
y_{19} BRE/BRZ
y_{20} BRNE/BRNZ
y_{21} BRG
y_{22} BRGE
y_{23} X1
y_{24} X0
y_{25} Y1
y_{26} Y0
I_{15} I_{14} I_{13} I_{12} I_{11} I_{10} I_9 I_8


1
En
OPCODE decoder
w_7
w_6
w_5
w_4
w_3
w_2
w_1
w_0
I_{15} I_{14} I_{13} I_{12} I_{11} I_{10} I_9 I_8
y_0 NOOP
y_1 INPUTC
y_2 INPUTCF
y_3 INPUTD
y_4 INPUTDF
y_5 MOVE
y_6 LOADI/LOADP
y_7 ADD
y_8 ADDI
y_9 SUB
y_{10} SUBI
y_{11} LOAD
y_{12} LOADF
y_{13} STORE
y_{14} STOREF
y_{15} SHIFTL
y_{16} SHIFTR
y_{17} CMP
y_{18} JUMP
y_{19} BRE/BRZ
y_{20} BRNE/BRNZ
y_{21} BRG
y_{22} BRGE
y_{23} X1
y_{24} X0
y_{25} Y1
y_{26} Y0
23 one-hot encoded outputs


1
En
OPCODE decoder
w_7
w_6
w_5
w_4
w_3
w_2
w_1
w_0
y_0 NOOP
y_1 INPUTC
y_2 INPUTCF
y_3 INPUTD
y_4 INPUTDF
y_5 MOVE
y_6 LOADI/LOADP
y_7 ADD
y_8 ADDI
y_9 SUB
y_{10} SUBI
y_{11} LOAD
y_{12} LOADF
y_{13} STORE
y_{14} STOREF
y_{15} SHIFTL
y_{16} SHIFTR
y_{17} CMP
y_{18} JUMP
y_{19} BRE/BRZ
y_{20} BRNE/BRNZ
y_{21} BRG
y_{22} BRGE
y_{23} X1
y_{24} X0
y_{25} Y1
y_{26} Y0
These 4 are NOT one-hot encoded
I_{15} I_{14} I_{13} I_{12} I_{11} I_{10} I_9 I_8


23 + 4
c1
100000 00110100 00000000
100001 00110000 00000001
100010 10001100 00000000
100011 11010011 00000000
100100 11110010 00000011
100101 01000100 00000000
100110 01010000 00000001
100111 11100000 11111011
101000 10100100 00000010
Code Memory
6
16
6
16
8 high
8 low
OpCode Decoder
27
4
Control
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c8 c9 c10
Registers
A 00000000
B 00000000
C 00000000
D 00000000
c4 c5
c6 c7
A B C D
8
c12 c13
ALU
c14
Flags
0 0 0 0
4
c15
c11
c16
c17
c18
6 low
4 low
8 low
0000 00000101
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
Data Memory
Switches
16
c2
c3
PC Update Logic
PC
100000
i281 CPU

The Control Table

	IMEM_WRITE_ENABLE	PROGRAM_COUNTER_MUX	PROGRAM_COUNTER_WRITE_ENABLE	REGISTERS_PORT0_SELECT1	REGISTERS_PORT0_SELECT0	REGISTERS_PORT1_SELECT1	REGISTERS_PORT1_SELECT0	REGISTERS_WRITE_SELECT1	REGISTERS_WRITE_SELECT0	REGISTERS_WRITE_ENABLE	ALU_SOURCE_MUX	ALU_SELECT1	ALU_SELECT0	FLAGS_WRITE_ENABLE	ALU_RESUT_MUX	DMEM_INPUT_MUX	DMEM_WRITE_ENABLE	REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

23 one-hot encoded OPCODEs

	IMEM_WRITE_ENABLE	PROGRAM_COUNTER_MUX	PROGRAM_COUNTER_WRITE_ENABLE	REGISTERS_PORT0_SELECT1	REGISTERS_PORT0_SELECT0	REGISTERS_PORT1_SELECT1	REGISTERS_PORT1_SELECT0	REGISTERS_WRITE_SELECT1	REGISTERS_WRITE_SELECT0	REGISTERS_WRITE_ENABLE	ALU_SOURCE_MUX	ALU_SELECT1	ALU_SELECT0	FLAGS_WRITE_ENABLE	ALU_RESUT_MUX	DMEM_INPUT_MUX	DMEM_WRITE_ENABLE	REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

18 control lines

23 one-hot encoded OPCODEs

	IMEM_WRITE_ENABLE	PROGRAM_COUNTER_MUX	PROGRAM_COUNTER_WRITE_ENABLE	REGISTERS_PORT0_SELECT1	REGISTERS_PORT0_SELECT0	REGISTERS_PORT1_SELECT1	REGISTERS_PORT1_SELECT0	REGISTERS_WRITE_SELECT1	REGISTERS_WRITE_SELECT0	REGISTERS_WRITE_ENABLE	ALU_SOURCE_MUX	ALU_SELECT1	ALU_SELECT0	FLAGS_WRITE_ENABLE	ALU_RESUT_MUX	DMEM_INPUT_MUX	DMEM_WRITE_ENABLE	REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

	IMEM_WRITE_ENABLE	PROGRAM_COUNTER_MUX	PROGRAM_COUNTER_WRITE_ENABLE	REGISTERS_PORT0_SELECT1	REGISTERS_PORT0_SELECT0	REGISTERS_PORT1_SELECT1	REGISTERS_PORT1_SELECT0	REGISTERS_WRITE_SELECT1	REGISTERS_WRITE_SELECT0	REGISTERS_WRITE_ENABLE	ALU_SOURCE_MUX	ALU_SELECT1	ALU_SELECT0	FLAGS_WRITE_ENABLE	ALU_RESUT_MUX	DMEM_INPUT_MUX	DMEM_WRITE_ENABLE	REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

Taken from these bits of the instruction

C_{15}	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0
				X_1	X_0										

	IMEM_WRITE_ENABLE	PROGRAM_COUNTER_MUX	PROGRAM_COUNTER_WRITE_ENABLE	REGISTERS_PORT0_SELECT1	REGISTERS_PORT0_SELECT0	REGISTERS_PORT1_SELECT1	REGISTERS_PORT1_SELECT0	REGISTERS_WRITE_SELECT1	REGISTERS_WRITE_SELECT0	REGISTERS_WRITE_ENABLE	ALU_SOURCE_MUX	ALU_SELECT1	ALU_SELECT0	FLAGS_WRITE_ENABLE	ALU_RESUT_MUX	DMEM_INPUT_MUX	DMEM_WRITE_ENABLE	REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

Taken from these bits of the instruction

C_{15}	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0
				X_1	X_0										

	IMEM_WRITE_ENABLE	PROGRAM_COUNTER_MUX	PROGRAM_COUNTER_WRITE_ENABLE	REGISTERS_PORT0_SELECT1	REGISTERS_PORT0_SELECT0	REGISTERS_PORT1_SELECT1	REGISTERS_PORT1_SELECT0	REGISTERS_WRITE_SELECT1	REGISTERS_WRITE_SELECT0	REGISTERS_WRITE_ENABLE	ALU_SOURCE_MUX	ALU_SELECT1	ALU_SELECT0	FLAGS_WRITE_ENABLE	ALU_RESUT_MUX	DMEM_INPUT_MUX	DMEM_WRITE_ENABLE	REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

Taken from these bits of the instruction

C_{15}	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0
						Y_1	Y_0								

	IMEM_WRITE_ENABLE	PROGRAM_COUNTER_MUX	PROGRAM_COUNTER_WRITE_ENABLE	REGISTERS_PORT0_SELECT1	REGISTERS_PORT0_SELECT0	REGISTERS_PORT1_SELECT1	REGISTERS_PORT1_SELECT0	REGISTERS_WRITE_SELECT1	REGISTERS_WRITE_SELECT0	REGISTERS_WRITE_ENABLE	ALU_SOURCE_MUX	ALU_SELECT1	ALU_SELECT0	FLAGS_WRITE_ENABLE	ALU_RESUT_MUX	DMEM_INPUT_MUX	DMEM_WRITE_ENABLE	REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

Taken from these bits of the instruction

C_{15}	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0
						Y_1	Y_0								

	IMEM_WRITE_ENABLE	PROGRAM_COUNTER_MUX	PROGRAM_COUNTER_WRITE_ENABLE	REGISTERS_PORT0_SELECT1	REGISTERS_PORT0_SELECT0	REGISTERS_PORT1_SELECT1	REGISTERS_PORT1_SELECT0	REGISTERS_WRITE_SELECT1	REGISTERS_WRITE_SELECT0	REGISTERS_WRITE_ENABLE	ALU_SOURCE_MUX	ALU_SELECT1	ALU_SELECT0	FLAGS_WRITE_ENABLE	ALU_RESUT_MUX	DMEM_INPUT_MUX	DMEM_WRITE_ENABLE	REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

	IMEM_WRITE_ENABLE	PROGRAM_COUNTER_MUX	PROGRAM_COUNTER_WRITE_ENABLE	REGISTERS_PORT0_SELECT1	REGISTERS_PORT0_SELECT0	REGISTERS_PORT1_SELECT1	REGISTERS_PORT1_SELECT0	REGISTERS_WRITE_SELECT1	REGISTERS_WRITE_SELECT0	REGISTERS_WRITE_ENABLE	ALU_SOURCE_MUX	ALU_SELECT1	ALU_SELECT0	FLAGS_WRITE_ENABLE	ALU_RESUT_MUX	DMEM_INPUT_MUX	DMEM_WRITE_ENABLE	REG_WRITEBACK_MUX
NOOP			1															
INPUTC	1		1												1			
INPUTCF	1		1	X1	X0						1	1						
INPUTD			1												1	1	1	
INPUTDF			1	X1	X0						1	1				1	1	
MOVE			1	Y1	Y0			X1	X0	1	1	1						
LOADI/LOADP			1					X1	X0	1					1			
ADD			1	X1	X0	Y1	Y0	X1	X0	1		1		1				
ADDI			1	X1	X0			X1	X0	1	1	1		1				
SUB			1	X1	X0	Y1	Y0	X1	X0	1		1	1	1				
SUBI			1	X1	X0			X1	X0	1	1	1	1	1				
LOAD			1					X1	X0	1					1			1
LOADF			1	Y1	Y0			X1	X0	1	1	1						1
STORE			1			X1	X0								1		1	
STOREF			1	Y1	Y0	X1	X0				1	1					1	
SHIFTL			1	X1	X0			X1	X0	1				1				
SHIFTR			1	X1	X0			X1	X0	1			1	1				
CMP			1	X1	X0	Y1	Y0					1	1	1				
JUMP		1	1															
BRE/BRZ		B1	1															
BRNE/BRNZ		B2	1															
BRG		B3	1															
BRGE		B4	1															

computed using
the flags register

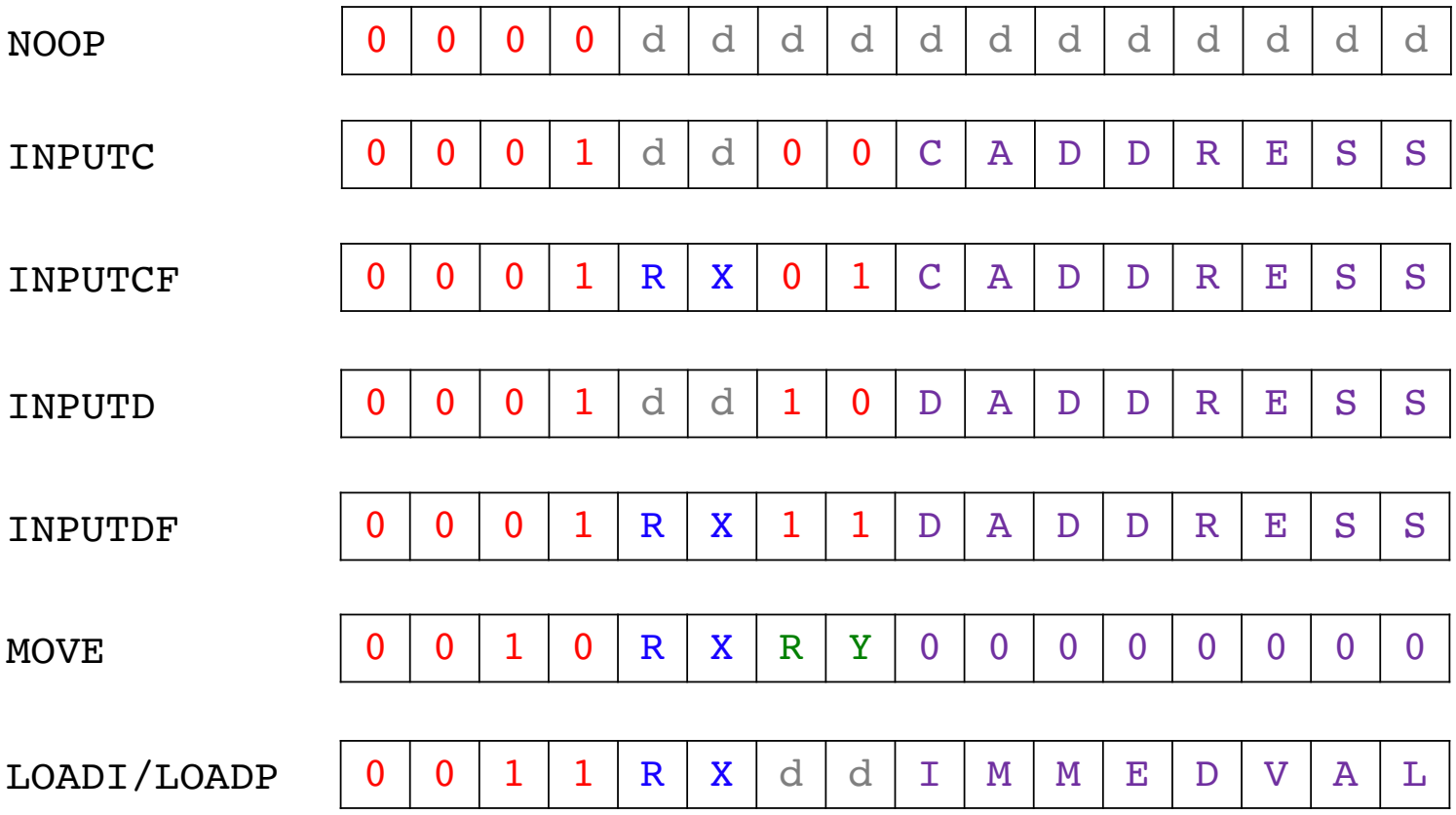
B1= ZF
B2= ~ZF
B3= AND(~ZF, XNOR(NF, OF))
B4= XNOR(NF, OF)

Zero Flag (ZF)
Negative Flag (NF)
Overflow Flag (OF)

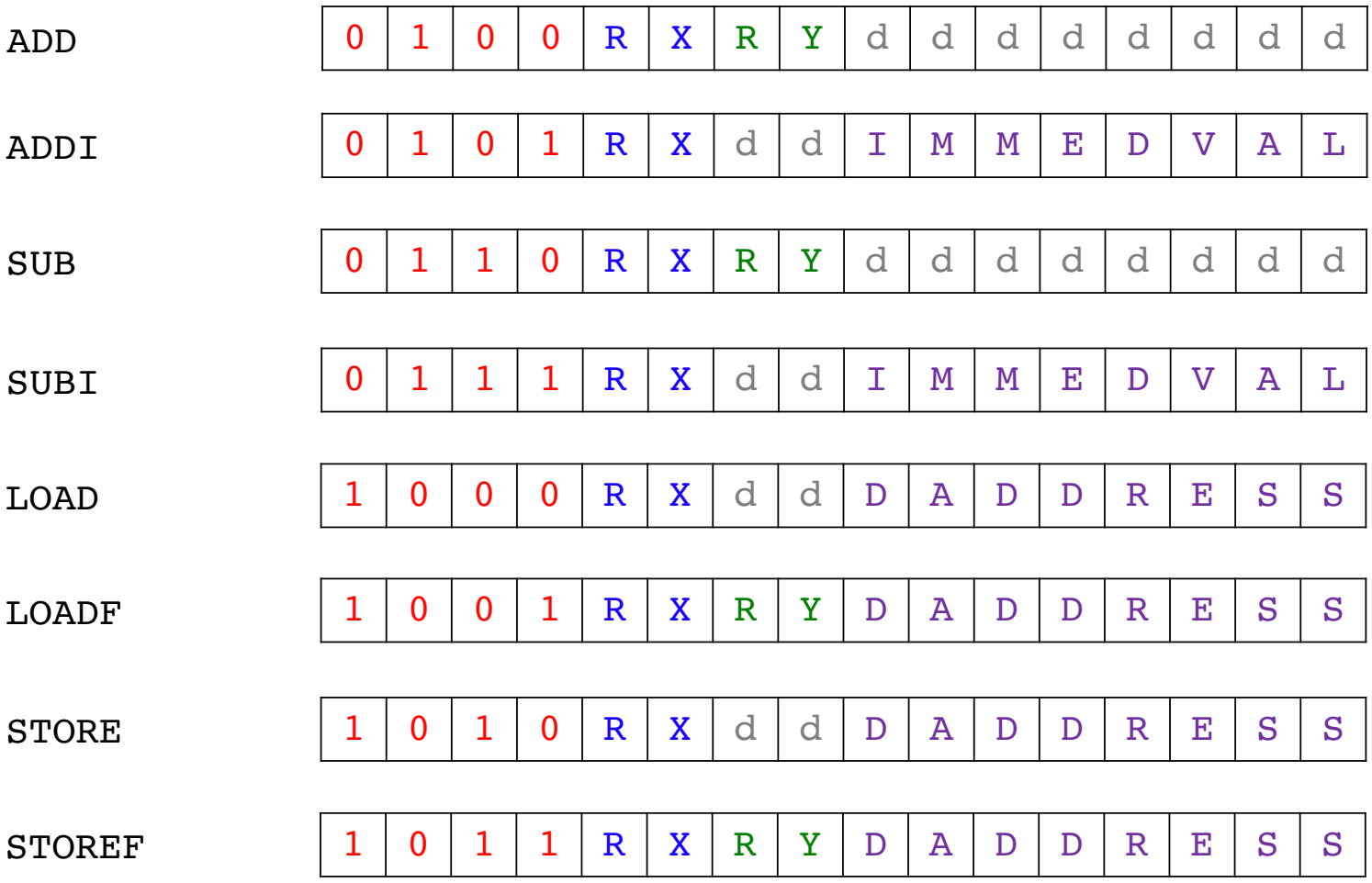
The OPCODEs

(Mapped to Machine Language)

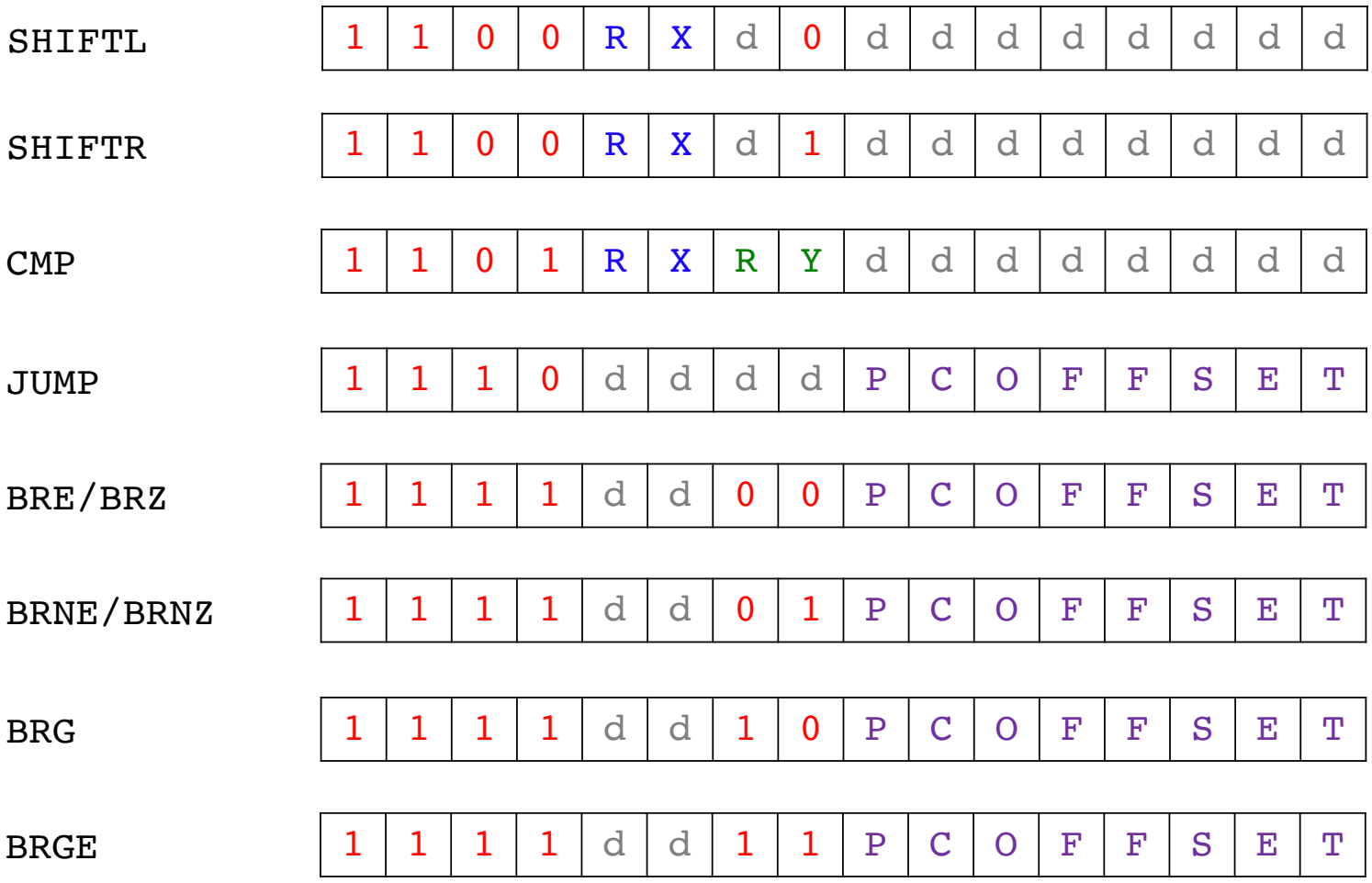
The OPCODEs


NOOP
0 0 0 0 d d d d d d d d d d d d
INPUTC
0 0 0 1 d d 0 0 C A D D R E S S
INPUTCF
0 0 0 1 R X 0 1 C A D D R E S S
INPUTD
0 0 0 1 d d 1 0 D A D D R E S S
INPUTDF
0 0 0 1 R X 1 1 D A D D R E S S
MOVE
0 0 1 0 R X R Y 0 0 0 0 0 0 0 0
LOADI/LOADP
0 0 1 1 R X d d I M M E D V A L

The OPCODEs


ADD
0 1 0 0 R X R Y d d d d d d d d
ADDI
0 1 0 1 R X d d I M M E D V A L
SUB
0 1 1 0 R X R Y d d d d d d d d
SUBI
0 1 1 1 R X d d I M M E D V A L
LOAD
1 0 0 0 R X d d D A D D R E S S
LOADF
1 0 0 1 R X R Y D A D D R E S S
STORE
1 0 1 0 R X d d D A D D R E S S
STOREF
1 0 1 1 R X R Y D A D D R E S S

The OPCODEs


SHIFTL
1 1 0 0 R X d 0 d d d d d d d d
SHIFTR
1 1 0 0 R X d 1 d d d d d d d d
CMP
1 1 0 1 R X R Y d d d d d d d d
JUMP
1 1 1 0 d d d d P C O F F S E T
BRE/BRZ
1 1 1 1 d d 0 0 P C O F F S E T
BRNE/BRNZ
1 1 1 1 d d 0 1 P C O F F S E T
BRG
1 1 1 1 d d 1 0 P C O F F S E T
BRGE
1 1 1 1 d d 1 1 P C O F F S E T

Questions?

THE END