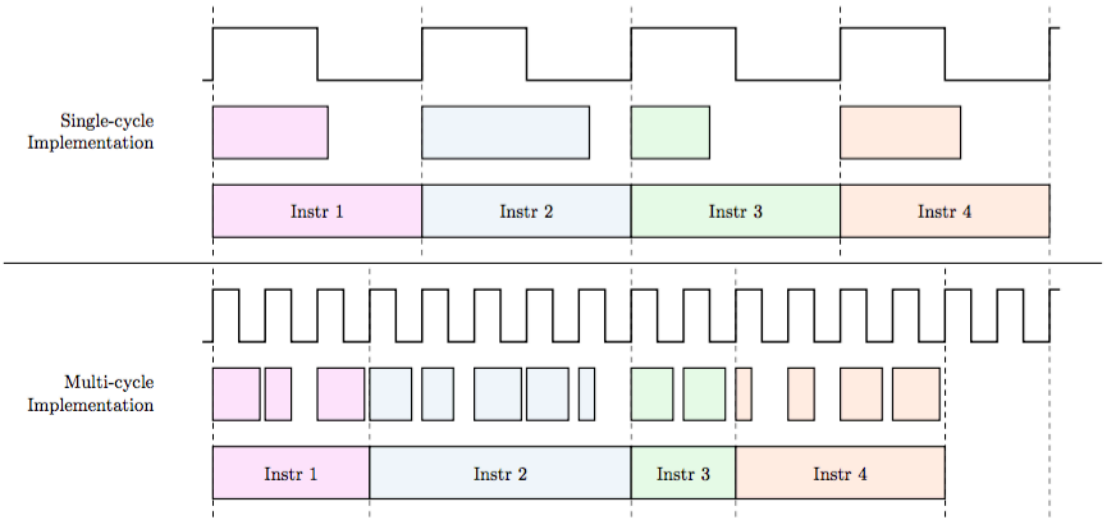
Lecture 4

Multi-Cycle Implementation and Microprogramming

We study another technique to implement Y86 instruction set architecture. This technique is called '*multi-cycle implementation*'[Par05]. Implied from its name, different types of instructions may require different number of cycles.

4.1 Limitation of Single-Cycle Implementation

Different types of instructions typically need different number of computation steps. However, the *single-cycle implementation* is designed to use only one cycle for every type of instructions. The clock period needs to cover all the computation steps, although many instructions need only subset of the steps. The performance of the processor can be improved if we can shorten the execution period for some types of instructions.



4.2 Multi-cycle Implementation

The multi-cycle implementation is designed to perform only subset of the computation steps in one cycle. Therefore, the clock period can be made shorter than the clock period of the single-cycle implementation. However, a single instruction needs multiple cycles to complete all the required steps.

Another benefit of the multi-cycle implementation is that all types of instructions do not need to use the same number of cycles for the execution. Simple instructions can be executed in a fewer number of cycles than sophisticated instructions. This therefore improves the performance of the processor.

4.2.1 Datapath

We need to introduce a number of temporary storages in the datapath. They are necessary to store the values through the multiple cycles that an instruction is executed. Here are the storages and their control signals.

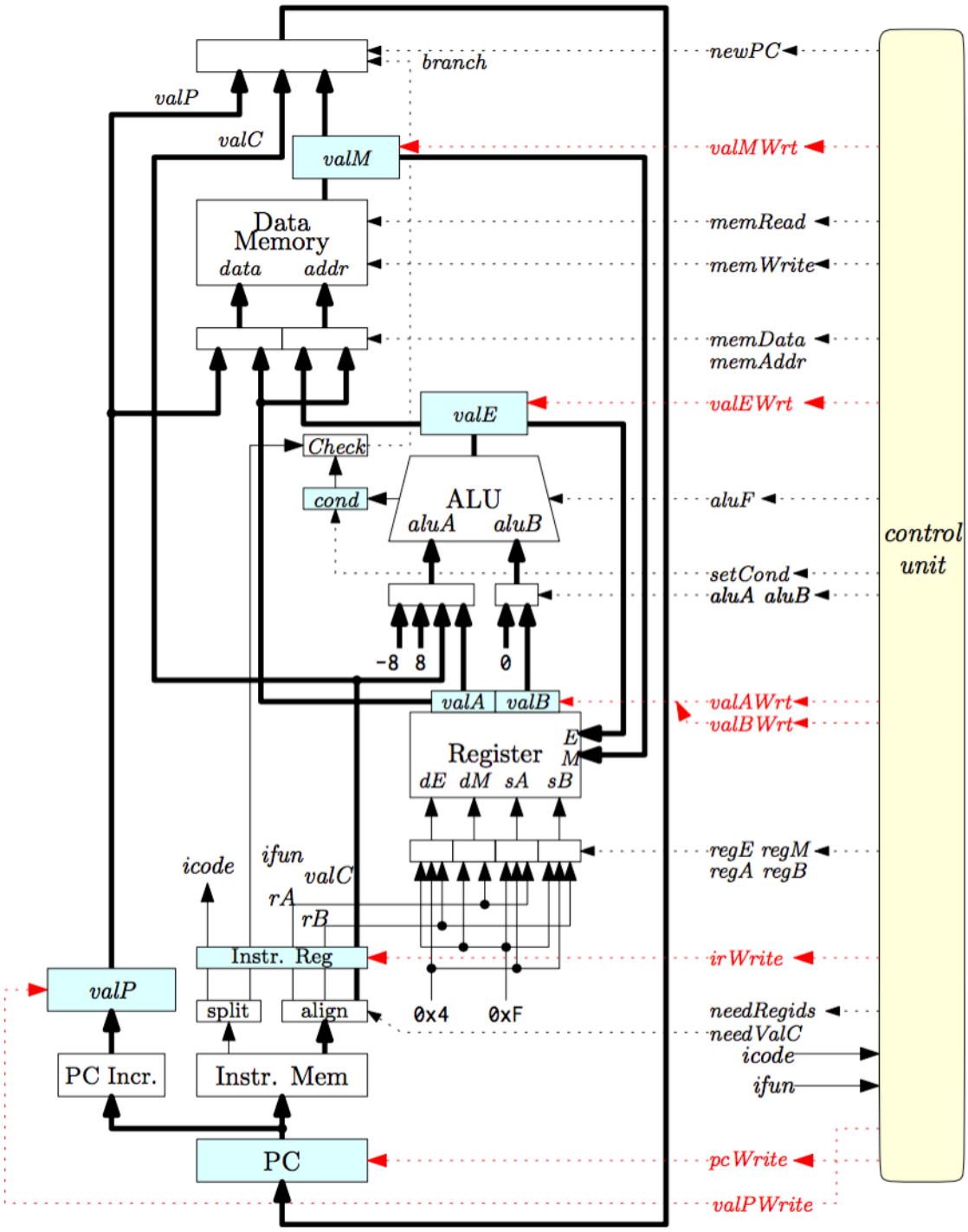
1. *Instruction Register* is used to stored the fetched instruction. The *irWrite* control signal can be set to 1 when the register needs updating.

2. *valP* and *valPWrite* are a storage and a control signal for the address of the next instruction.

3. *valA* and *valB* are storages from the values obtained from the register file. We use *valAWrite* and *valBWrite* to control the updating.

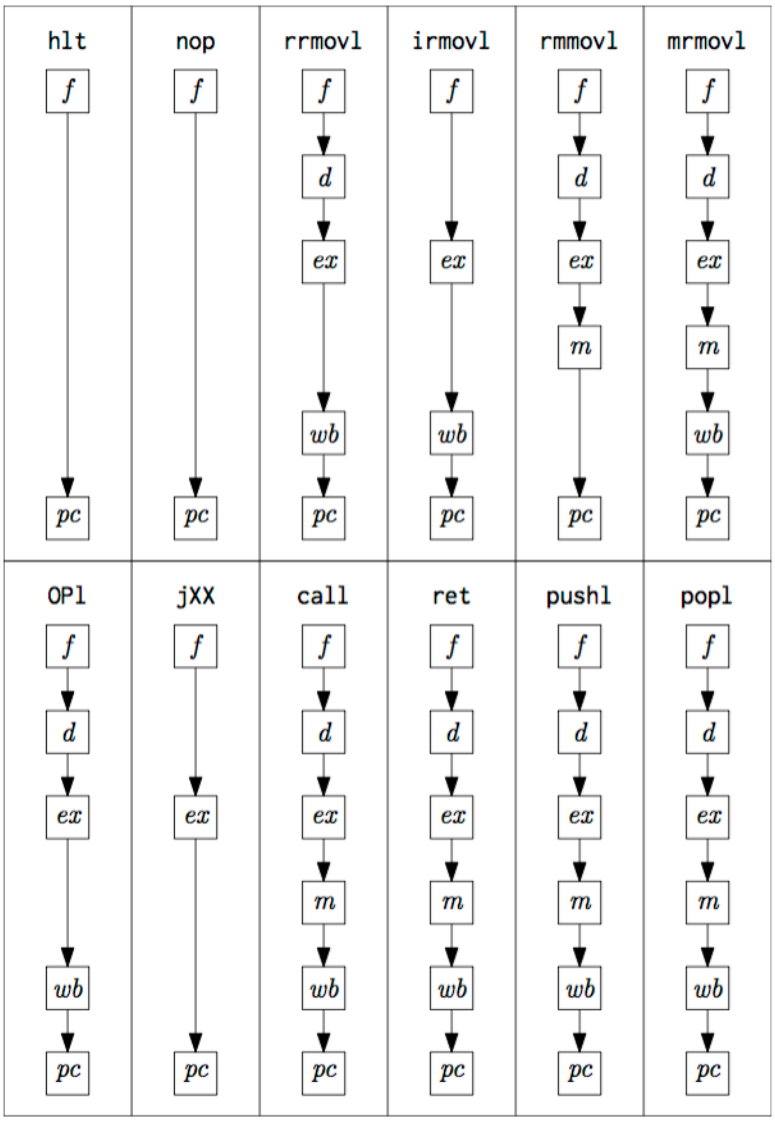
4. *valE* and *valM* are storages as well. We use *valEWrite* and *valMWrite* to control the updating.

5. *pcWrite* control signal is added to update the *PC* only the cycle performing *PC Update*.


branch
newPC
valP
valC
valM
valMWrt
Data Memory
data
addr
memRead
memWrite
memData
memAddr
valE
valEWrt
Check
cond
ALU
aluA
aluB
aluF
control unit
setCond
aluA aluB
-8 8
0
valA
valB
valAWrt
valBWrt
Register
E
M
dE dM sA sB
regE regM
regA regB
icode
ifun
valC
rA
rB
Instr. Reg
irWrite
valP
split
align
0x4
0xF
needRegids
needValC
icode
ifun
PC Incr.
Instr. Mem
PC
pcWrite
valPWrite

4.2.2 Control Unit

To design a control unit for this multi-cycle implementation, we first need to check how each type of instruction is executed. The following figure summarizes the execution stages used in each type of instruction.



The control unit can be then designed as a *state machine* that emits the control signals based on the current state and the instruction code (*icode*).

Example 4.1 Show how a **nop** instruction is executed.

- *Cycle 1*: the instruction is read from the instruction memory.

nRegid	*nValC*	*regA*	*regB*	*regE*	*regM*	*aluA*	*aluB*
0	1	XX	XX	00	00	XX	X
aluF	*setCnd*	*mAddr*	*mData*	*mRd*	*mWrt*	*newPC*	*vPWrt*
XX	0	X	X	X	0	XX	1
irWrt	*vAWrt*	*vBWrt*	*vEWrt*	*vMWrt*	*pcWrt*		
1	0	0	0	0	0		

- *Cycle 2*: update *PC* to the address of the next instruction (*valP*).

nRegid	*nValC*	*regA*	*regB*	*regE*	*regM*	*aluA*	*aluB*
X	X	XX	XX	00	00	XX	X
aluF	*setCnd*	*mAddr*	*mData*	*mRd*	*mWrt*	*newPC*	*vPWrt*
XX	0	X	X	X	0	00	0
irWrt	*vAWrt*	*vBWrt*	*vEWrt*	*vMWrt*	*pcWrt*		
0	0	0	0	0	1		

Example 4.2 Show how a `irmov V, rB` instruction is executed.

- *Cycle 1*: the instruction is read from the instruction memory.

nRegid	*nValC*	*regA*	*regB*	*regE*	*regM*	*aluA*	*aluB*
1	1	XX	XX	00	00	XX	X
aluF	*setCnd*	*mAddr*	*mData*	*mRd*	*mWrt*	*newPC*	*vPWrt*
XX	0	X	X	X	0	XX	1
irWrt	*vAWrt*	*vBWrt*	*vEWrt*	*vMWrt*	*pcWrt*		
1	0	0	0	0	0		

- *Cycle 2*: perform $0 + valC$ by the ALU.

nRegid	*nValC*	*regA*	*regB*	*regE*	*regM*	*aluA*	*aluB*
X	X	XX	XX	00	00	10	0
aluF	*setCnd*	*mAddr*	*mData*	*mRd*	*mWrt*	*newPC*	*vPWrt*
00	0	X	X	X	0	XX	0
irWrt	*vAWrt*	*vBWrt*	*vEWrt*	*vMWrt*	*pcWrt*		
0	0	0	1	0	0		

- *Cycle 3*: write *valE* to register *rB*.

nRegid	*nValC*	*regA*	*regB*	*regE*	*regM*	*aluA*	*aluB*
X	X	XX	XX	10	00	XX	X
aluF	*setCnd*	*mAddr*	*mData*	*mRd*	*mWrt*	*newPC*	*vPWrt*
XX	0	X	X	X	0	XX	0
irWrt	*vAWrt*	*vBWrt*	*vEWrt*	*vMWrt*	*pcWrt*		
0	0	0	0	0	0		

- *Cycle 4:* update *PC* to *valP*.

nRegid	*nValC*	*regA*	*regB*	*regE*	*regM*	*aluA*	*aluB*
X	X	XX	XX	00	00	XX	X
aluF	*setCnd*	*mAddr*	*mData*	*mRd*	*mWrt*	*newPC*	*vPWrt*
XX	0	X	X	X	0	00	0
irWrt	*vAWrt*	*vBWrt*	*vEWrt*	*vMWrt*	*pcWrt*		
0	0	0	0	0	1		

Exercise 4.1 Write the control signal when `mrmov D(rB), rA` is executed.

- *Cycle 1*:

nRegid	*nValC*	*regA*	*regB*	*regE*	*regM*	*aluA*	*aluB*
aluF	*setCnd*	*mAddr*	*mData*	*mRd*	*mWrt*	*newPC*	*vPWrt*
irWrt	*vAWrt*	*vBWrt*	*vEWrt*	*vMWrt*	*pcWrt*		

- *Cycle 2*:

nRegid	*nValC*	*regA*	*regB*	*regE*	*regM*	*aluA*	*aluB*
aluF	*setCnd*	*mAddr*	*mData*	*mRd*	*mWrt*	*newPC*	*vPWrt*
irWrt	*vAWrt*	*vBWrt*	*vEWrt*	*vMWrt*	*pcWrt*		

- *Cycle 3*:

nRegid	*nValC*	*regA*	*regB*	*regE*	*regM*	*aluA*	*aluB*
aluF	*setCnd*	*mAddr*	*mData*	*mRd*	*mWrt*	*newPC*	*vPWrt*
irWrt	*vAWrt*	*vBWrt*	*vEWrt*	*vMWrt*	*pcWrt*		

- *Cycle 4:*

nRegid	*nValC*	*regA*	*regB*	*regE*	*regM*	*aluA*	*aluB*
aluF	*setCnd*	*mAddr*	*mData*	*mRd*	*mWrt*	*newPC*	*vPWrt*
irWrt	*vAWrt*	*vBWrt*	*vEWrt*	*vMWrt*	*pcWrt*		

- *Cycle 5:*

nRegid	*nValC*	*regA*	*regB*	*regE*	*regM*	*aluA*	*aluB*
aluF	*setCnd*	*mAddr*	*mData*	*mRd*	*mWrt*	*newPC*	*vPWrt*
irWrt	*vAWrt*	*vBWrt*	*vEWrt*	*vMWrt*	*pcWrt*		

- *Cycle 6:*

nRegid	*nValC*	*regA*	*regB*	*regE*	*regM*	*aluA*	*aluB*
aluF	*setCnd*	*mAddr*	*mData*	*mRd*	*mWrt*	*newPC*	*vPWrt*
irWrt	*vAWrt*	*vBWrt*	*vEWrt*	*vMWrt*	*pcWrt*		

Exercise 4.2 Write the control signal when `rrmov rA, rB` is executed.

- *Cycle 1*:

nRegid	*nValC*	*regA*	*regB*	*regE*	*regM*	*aluA*	*aluB*
aluF	*setCnd*	*mAddr*	*mData*	*mRd*	*mWrt*	*newPC*	*vPWrt*
irWrt	*vAWrt*	*vBWrt*	*vEWrt*	*vMWrt*	*pcWrt*		

- *Cycle 2*:

nRegid	*nValC*	*regA*	*regB*	*regE*	*regM*	*aluA*	*aluB*
aluF	*setCnd*	*mAddr*	*mData*	*mRd*	*mWrt*	*newPC*	*vPWrt*
irWrt	*vAWrt*	*vBWrt*	*vEWrt*	*vMWrt*	*pcWrt*		

- *Cycle 3*:

nRegid	*nValC*	*regA*	*regB*	*regE*	*regM*	*aluA*	*aluB*
aluF	*setCnd*	*mAddr*	*mData*	*mRd*	*mWrt*	*newPC*	*vPWrt*
irWrt	*vAWrt*	*vBWrt*	*vEWrt*	*vMWrt*	*pcWrt*		

- *Cycle 4:*

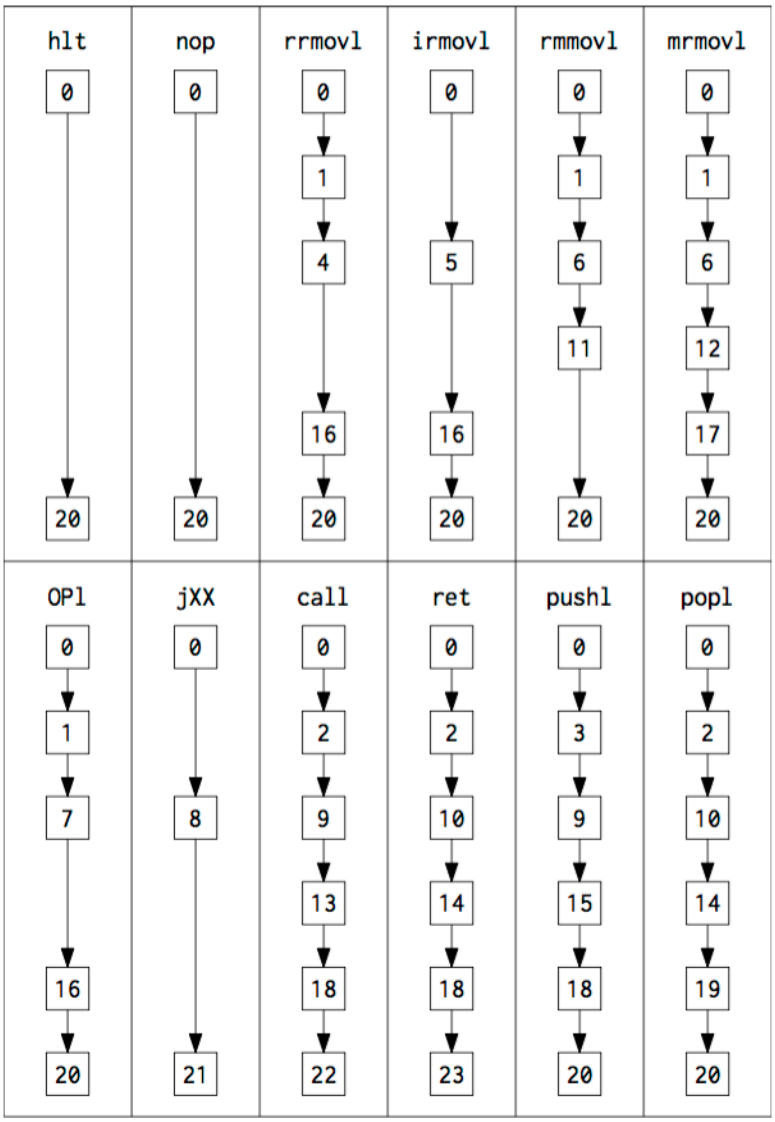
nRegid	*nValC*	*regA*	*regB*	*regE*	*regM*	*aluA*	*aluB*
aluF	*setCnd*	*mAddr*	*mData*	*mRd*	*mWrt*	*newPC*	*vPWrt*
irWrt	*vAWrt*	*vBWrt*	*vEWrt*	*vMWrt*	*pcWrt*		

- *Cycle 5:*

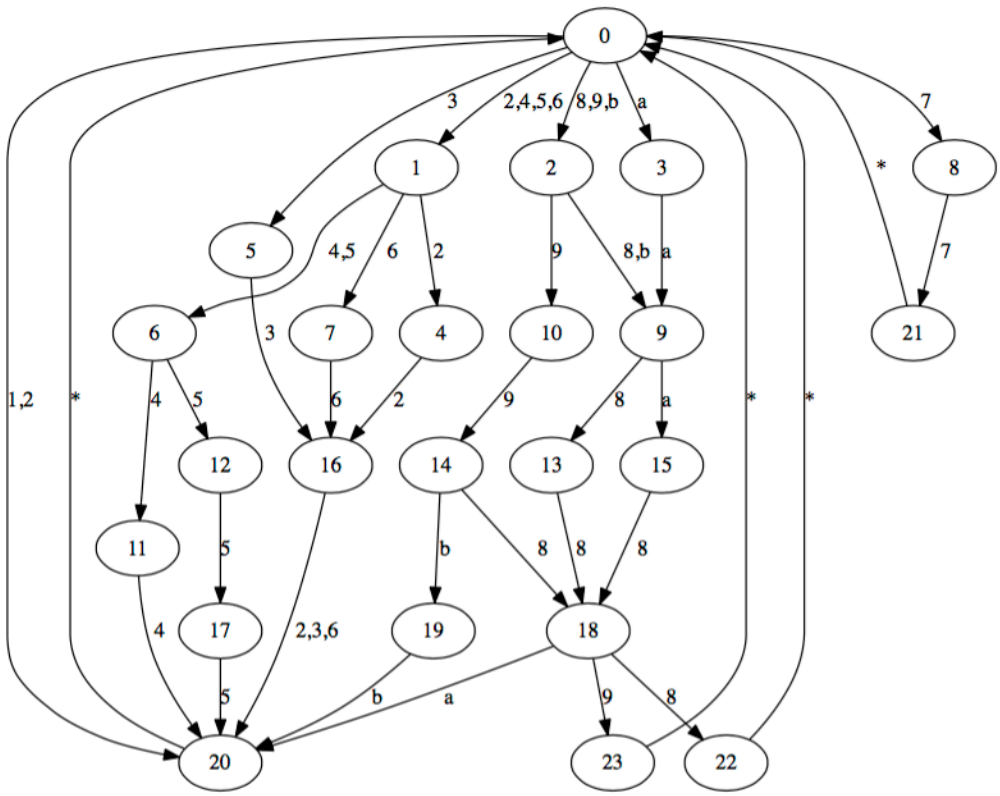
nRegid	*nValC*	*regA*	*regB*	*regE*	*regM*	*aluA*	*aluB*
aluF	*setCnd*	*mAddr*	*mData*	*mRd*	*mWrt*	*newPC*	*vPWrt*
irWrt	*vAWrt*	*vBWrt*	*vEWrt*	*vMWrt*	*pcWrt*		

To design the control unit, we first assign a *state number* to each computation steps. We use the same number for the steps performing the same or similar task. For example, the *Decode* stage of **rrmovl** and **rmmovl** can be assigned the same state number since they are compatible.

Stage	**rrmovl**	**rmmovl**
Decode	$valA \leftarrow R[rA]$	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$



The assigned state numbers can be combined to create a state machine. This can be later transformed to a *synchronous digital circuit* for the control unit.



4.2.3 Determining Clock Period

The *clock period* needs to be long enough to complete the longest stage in the datapath.

$$P = \max\{P_f, P_d, P_e, P_m, P_{wb}, P_{pc}\} \tag{4.1}$$

Exercise 4.3 From the following required time period for each computation stage, what are the clock period for the single-cycle and multi-cycle implementation of Y86?

Stage	Required Time Period
Fetch	0.3 ns
Decode	0.2 ns
Execute	0.3 ns
Memory	0.4 ns
Write back	0.2 ns
PC Update	0.1 ns

Exercise 4.4 What is the execution time and CPI of the following program when the clock period is set to 0.5 ns?

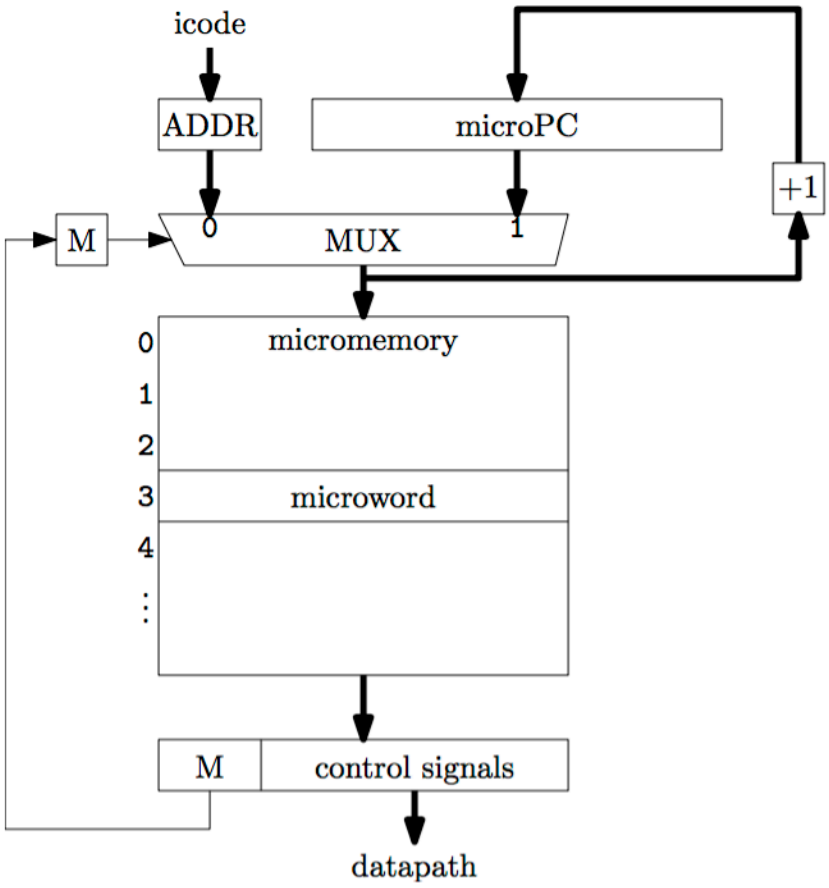
```
push    %rbx
push    %rsi
irmov   $0, %rbx
irmov   $5, %rsi
add     %rbx, %rsi
jmp     0x10900
pop     %rsi
pop     %rbx
ret
```

4.3 Microprogramming

A simple technique used to implement the control units is called '*hardwired control implementation*'. This technique has a limitation that changing a part in the datapath causes the circuit to be redesigned.

Microprogramming is a technique to implement the control units by using a ROM or RAM called "*micromemory*" to store the collection of control signals corresponding to the states, as well as the next state. Then, we can have a simple processor to read the micromemory and send the control signals to the datapath before moving to the next state.

Therefore, modifying the control unit become equivalent to editing the values in the micromemory The control unit can then be changed without redesigning the entire circuit.



For example, here are microwords for **nop** instruction.

```
Addr MUX  Control Signals
000:  0   00000000000000000000001100000  HLT Fetch
100:  1   00000000000000000000001100000  NOP Fetch
101:  0   00000000000000000000000000001  NOP PC Update
200:  1   10000000000000000000001100000  RRMOV Fetch
201:  1   00100000000000000000000010000  RRMOV Decode
202:  1   xxxxxxxxxxxxxxxxxxxxxxxxxxxxx  RRMOV Execute
203:  1   xxxxxxxxxxxxxxxxxxxxxxxxxxxxx  RRMOV Write back
204:  0   xxxxxxxxxxxxxxxxxxxxxxxxxxxxx  RRMOV PC Update
300:  1   xxxxxxxxxxxxxxxxxxxxxxxxxxxxx  IRMOV Fetch
...
```

References

[Par05] B. Parhami. *Computer Architecture: From Microprocessors To Supercomputers*. Oxford University Press, 2005.