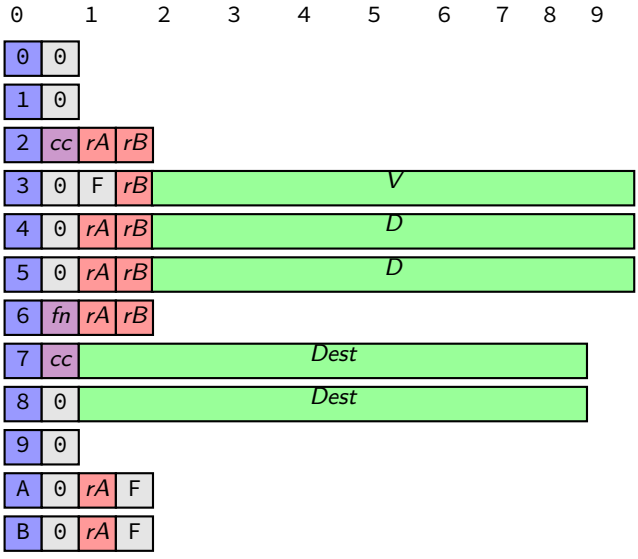
Y86-64 instruction formats



Y86-64 encoding (2)

```
addOne:
  irmovq  $1,   %rax
  addq    %rdi, %rax
  ret
```

Y86-64 state

`%r`*XX* — 15 registers

- ~~`%r15`~~ missing
- smaller parts of registers missing

ZF (zero), SF (sign), ~~OF (overflow)~~

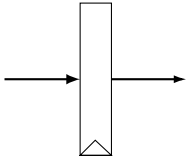
- book has `OF`, we'll not use it
- ~~CF~~ (carry) missing

`Stat` — processor status — halted?

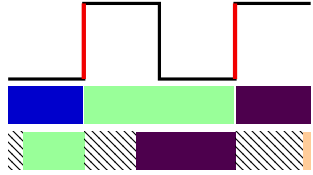
PC — program counter (AKA instruction pointer)

main memory

registers



updates every clock cycle



Y86-64 state

`%r`*XX* — 15 registers

- ~~`%r15`~~ missing
- smaller parts of registers missing

ZF (zero), SF (sign), ~~OF (overflow)~~

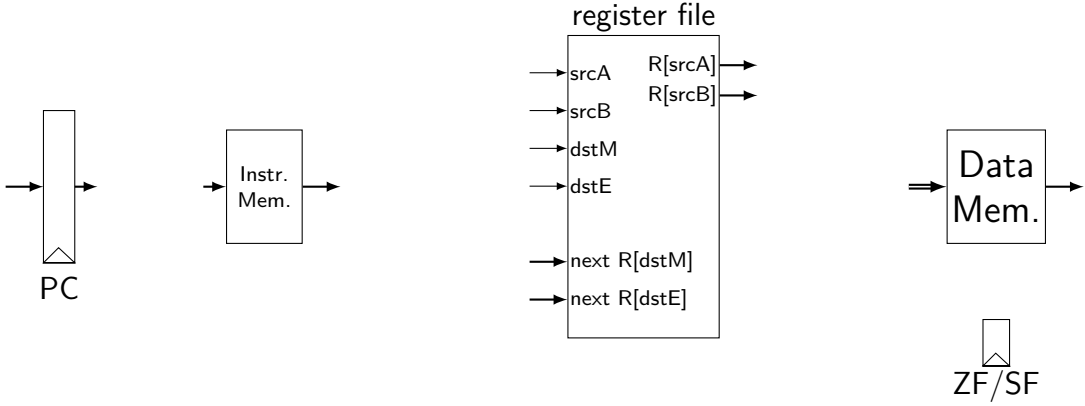
- book has `OF`, we'll not use it
- ~~CF~~ (carry) missing

`Stat` — processor status — halted?

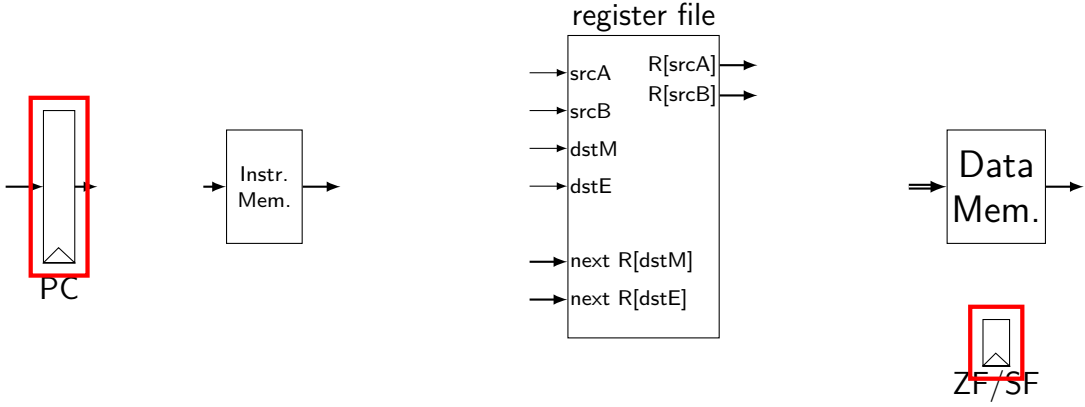
PC — program counter (AKA instruction pointer)

main memory

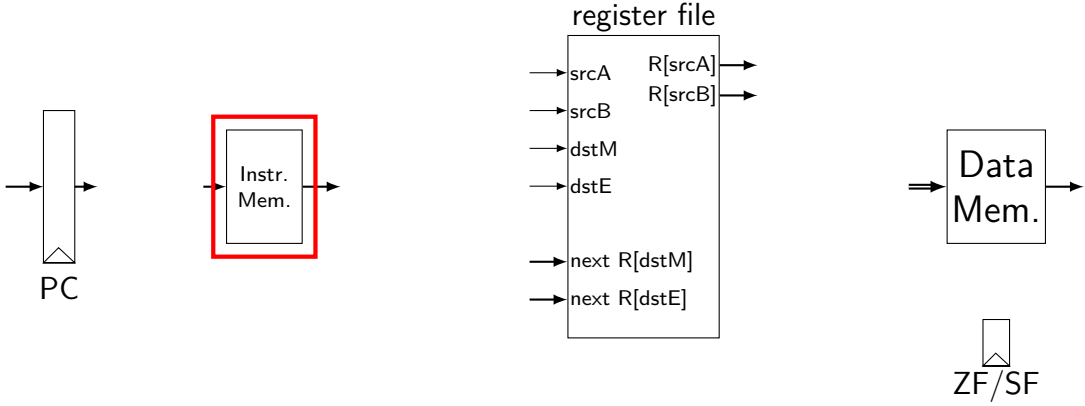
state in Y86-64



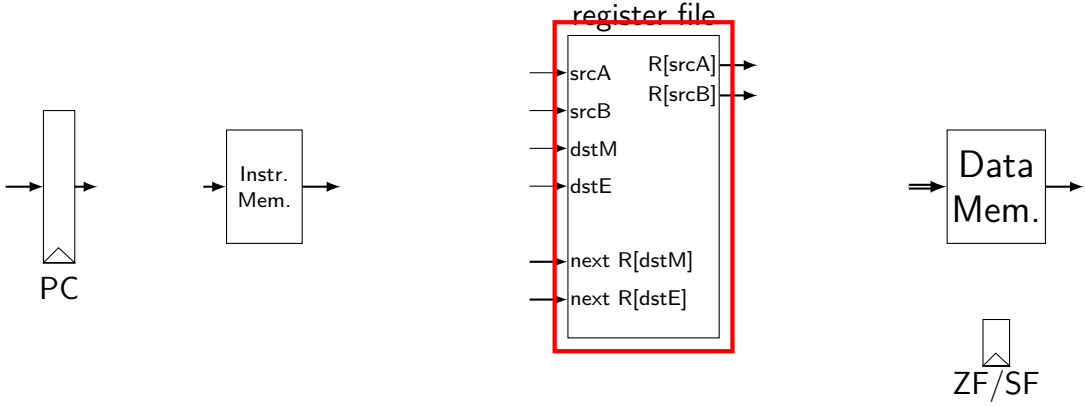
state in Y86-64



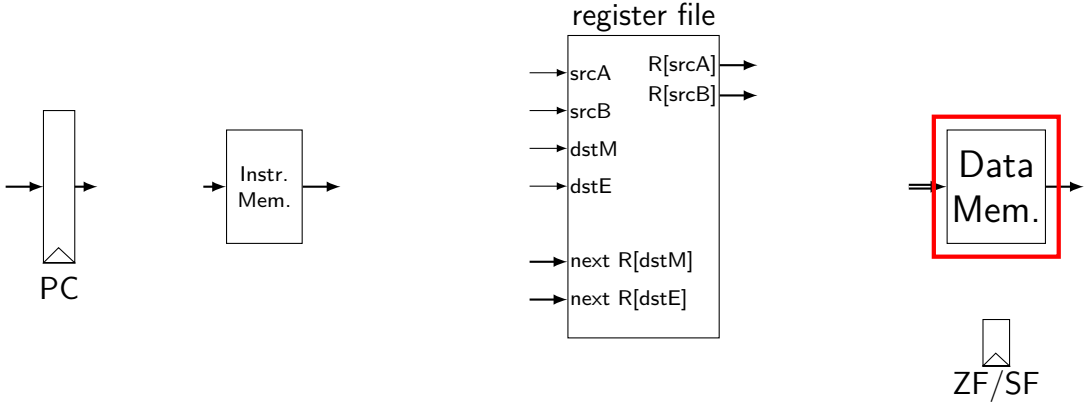
state in Y86-64



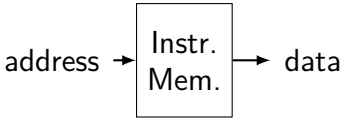
state in Y86-64

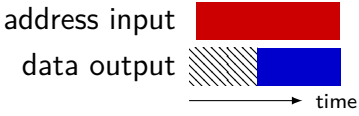


state in Y86-64

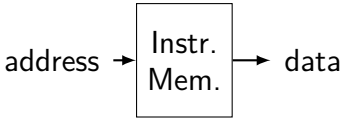


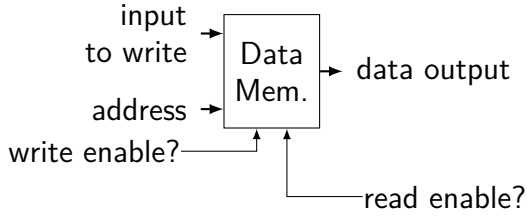
memories

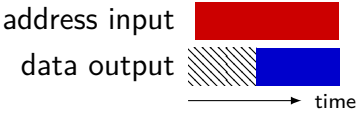




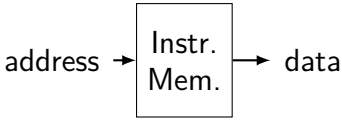
memories

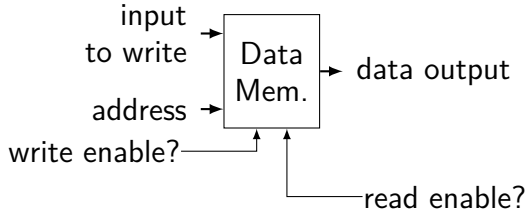

address
Instr.
Mem.
data

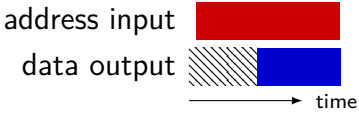

input
to write
address
write enable?
Data
Mem.
data output
read enable?

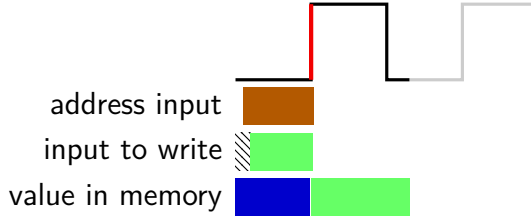

address input
data output
time

memories

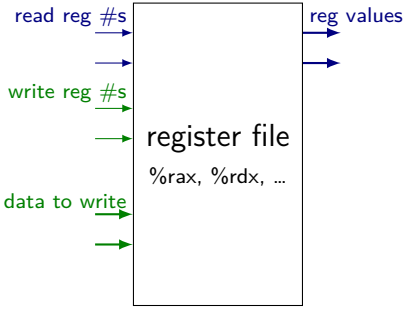

address
Instr.
Mem.
data


input
to write
address
write enable?
Data
Mem.
data output
read enable?

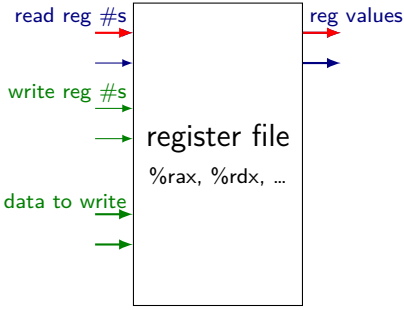

address input
data output
time

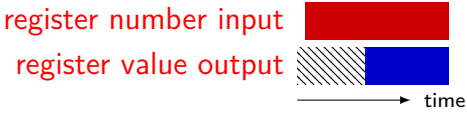

address input
input to write
value in memory

register file

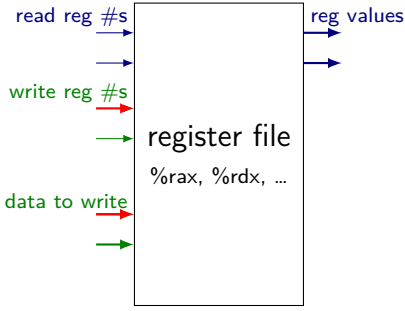


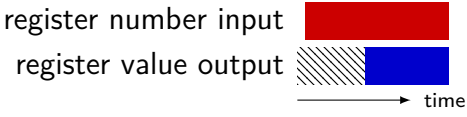
register file

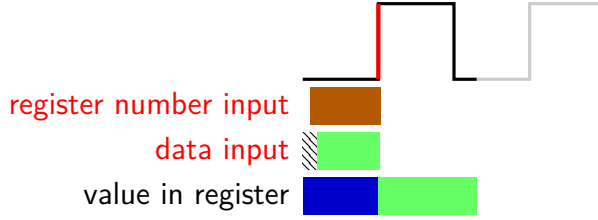




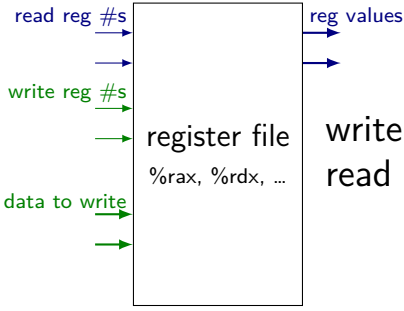
register file



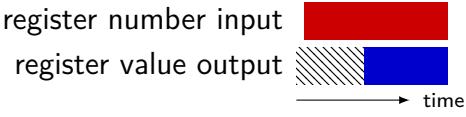


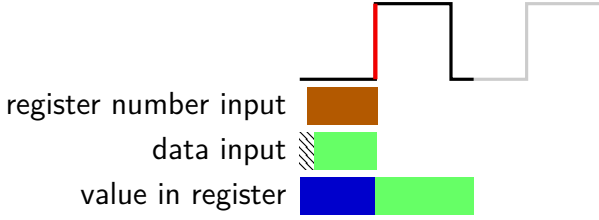


register file

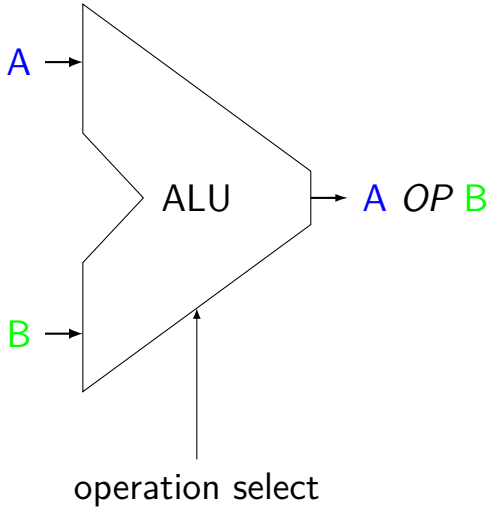


write register #15: write is ignored
read register #15: value is always 0





ALUs



Operations needed:
add — **addq**, addresses
sub — **subq**
xor — **xorq**
and — **andq**
more?

simple ISA 1: addq

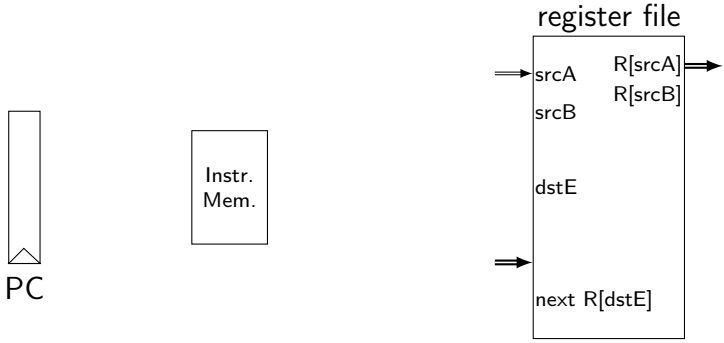
`addq %rXX, %rYY`

encoding: `%rXX` `%rYY` (two 4-bit register #s)

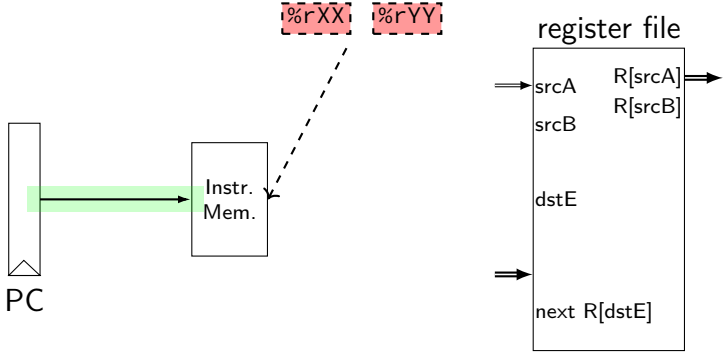
1 byte instructions, no opcode

no other instructions

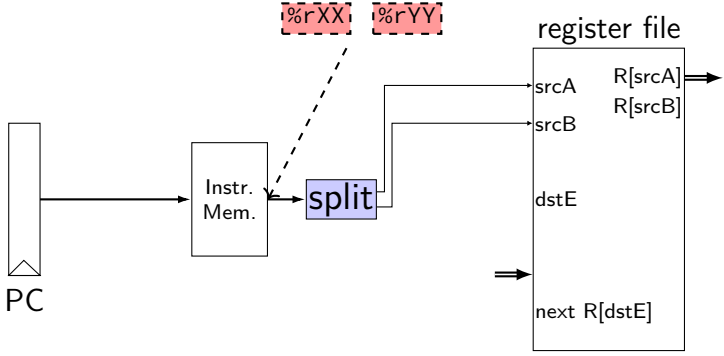
addq CPU


register file
srcA
R[srcA]
R[srcB]
srcB
dstE
next R[dstE]
Instr.
Mem.
PC

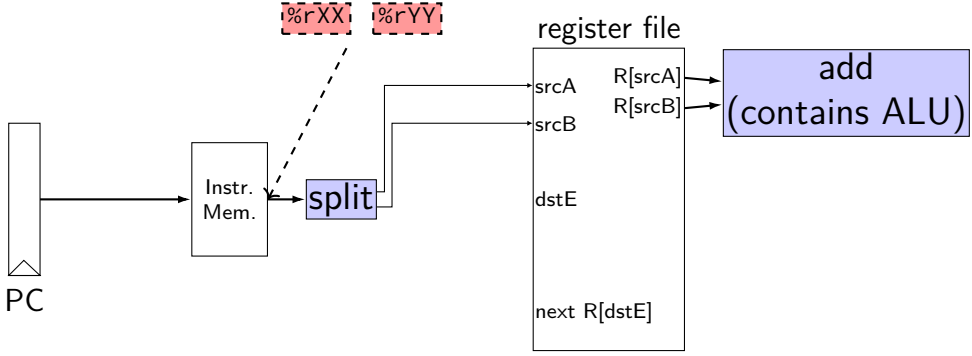
addq CPU


%rXX
%rYY
register file
srcA
srcB
dstE
R[srcA]
R[srcB]
next R[dstE]
Instr.
Mem.
PC

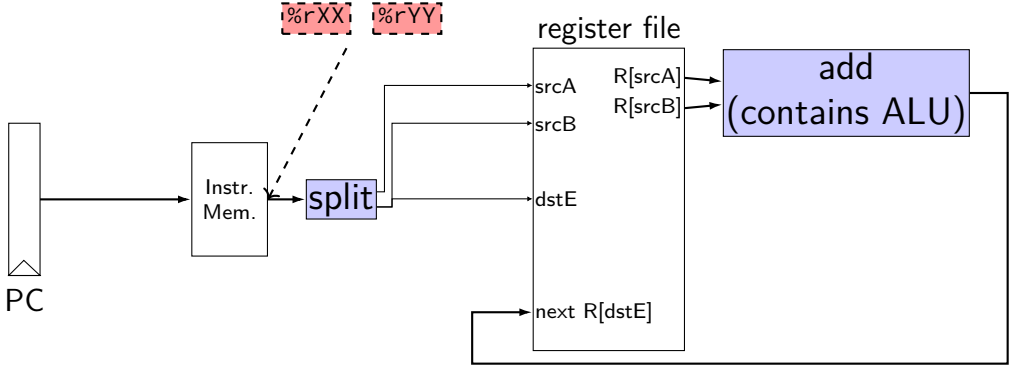
addq CPU


%rXX
%rYY
register file
srcA
srcB
R[srcA]
R[srcB]
dstE
next R[dstE]
Instr.
Mem.
split
PC

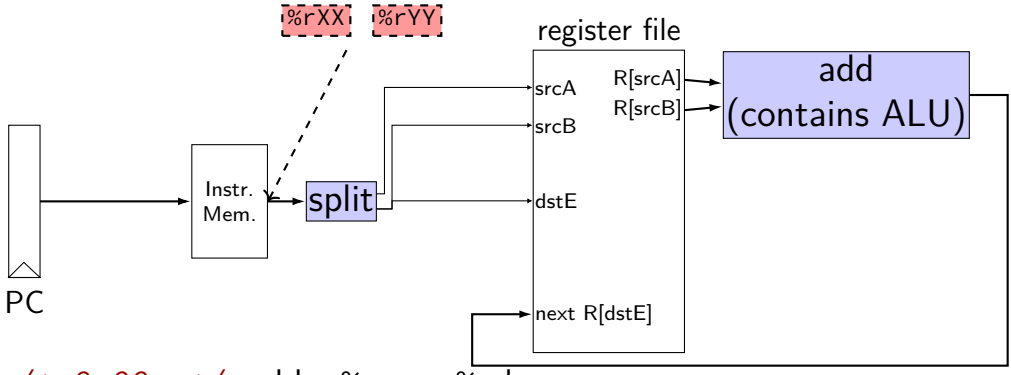
addq CPU


%rXX
%rYY
register file
srcA
srcB
dstE
next R[dstE]
R[srcA]
R[srcB]
add
(contains ALU)
Instr.
Mem.
split
PC

addq CPU


%rXX
%rYY
register file
srcA
srcB
dstE
next R[dstE]
R[srcA]
R[srcB]
add
(contains ALU)
Instr.
Mem.
split
PC

addq CPU



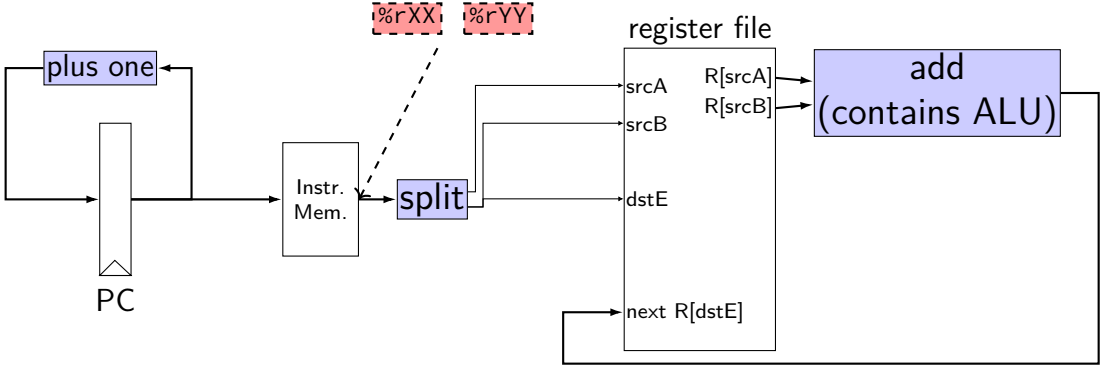

```
/* 0x00: */ addq %rax, %rdx
/* 0x01: */ addq %rbx, %rdx
```

initially: PC = 0x00, rax = 1, rbx = 2, rdx = 3

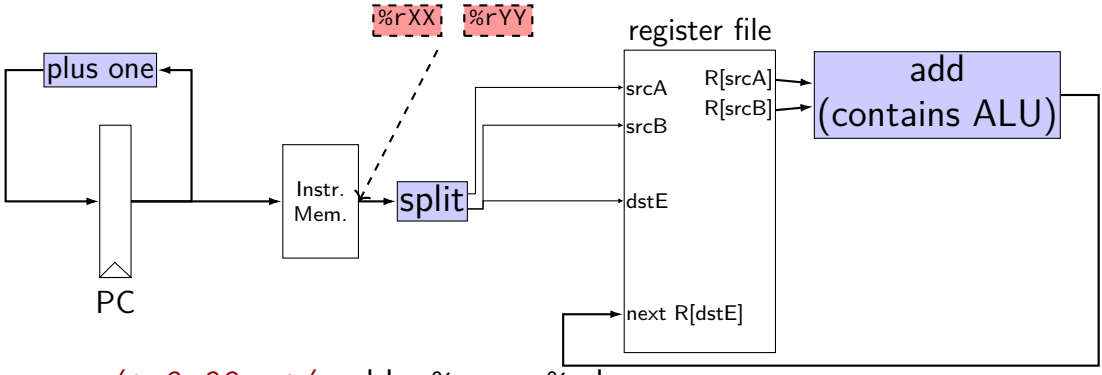
after cycle 1: PC = ????, rax = 1, rbx = 2, rdx = 4

after cycle 2: PC = ????, rax = ??, rbx = ??, rdx = ??

addq CPU


%rXX
%rYY
register file
plus one
srcA
R[srcA]
R[srcB]
add
(contains ALU)
srcB
Instr.
Mem.
split
dstE
PC
next R[dstE]

addq CPU




```
/* 0x00: */ addq %rax, %rdx
/* 0x01: */ addq %rbx, %rdx
```

initially: PC = 0x00, rax = 1, rbx = 2, rdx = 3

after cycle 1: PC = 0x01, rax = 1, rbx = 2, rdx = 4

after cycle 2: PC = 0x02, rax = 1, rbx = 2, rdx = 6

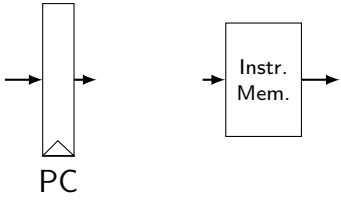
Simple ISA 2: jmp

`jmp label`

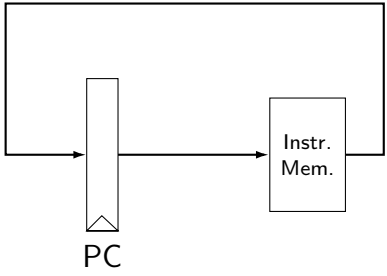
encoding: *8-byte little-endian address*

- 8 byte instructions, no opcode

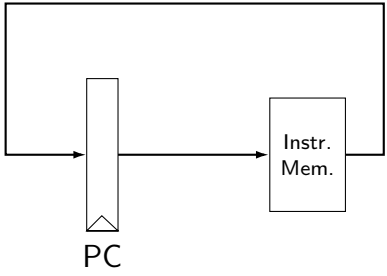
jmp CPU



jmp CPU



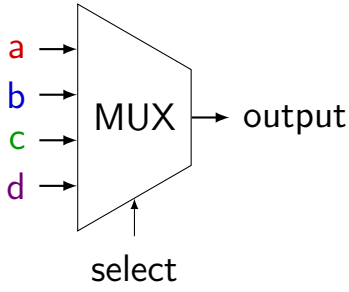
jmp CPU



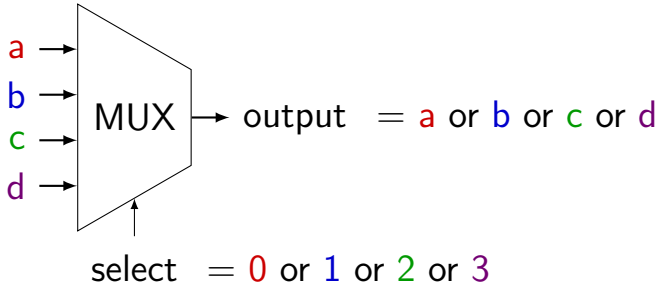

```
/* 0x00: */ jmp 0x10
/* 0x08: */ jmp 0x00
/* 0x10: */ jmp 0x08
```

initially: PC = 0x00
after cycle 1: PC = 0x10
after cycle 2: PC = 0x08
after cycle 3: PC = 0x00

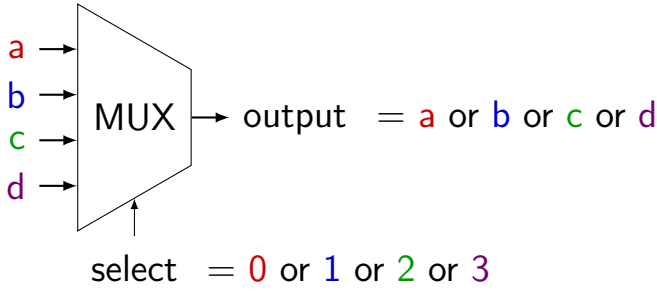
multiplexers



multiplexers



multiplexers



truth table:

select bit 1	select bit 0	output (many bits)
0	0	a
0	1	b
1	0	c
1	1	d

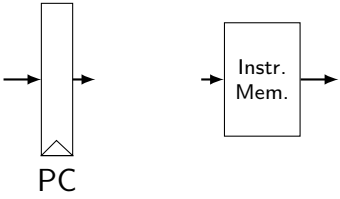
Simple ISA 3: Jmp or No-Op

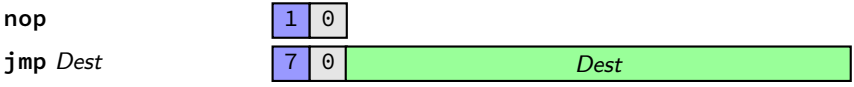
actual subset of Y86-64

`jmp LABEL` — encoded as `0x70` + address

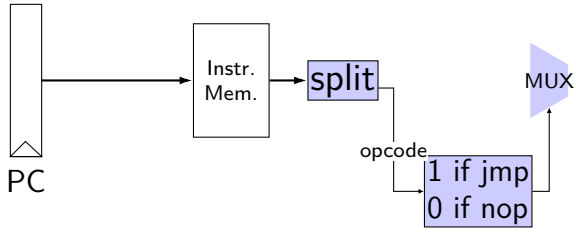
`nop` — encoded as `0x10`

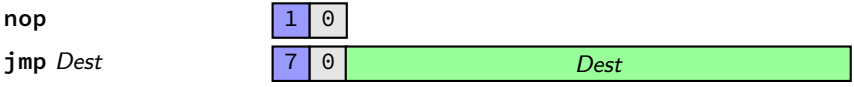
jmp+nop CPU



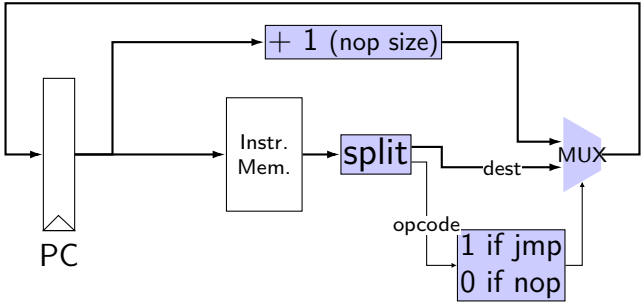


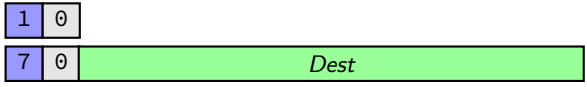
jmp+nop CPU


Instr.
Mem.
split
MUX
PC
opcode
1 if jmp
0 if nop

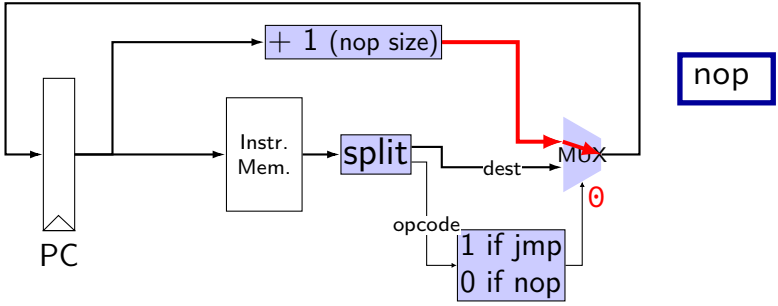

nop
jmp Dest
1
0
7
0
Dest

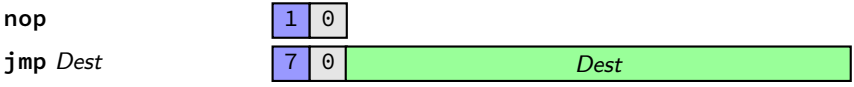
jmp+nop CPU


+ 1 (nop size)
Instr.
Mem.
split
dest
MUX
opcode
1 if jmp
0 if nop
PC

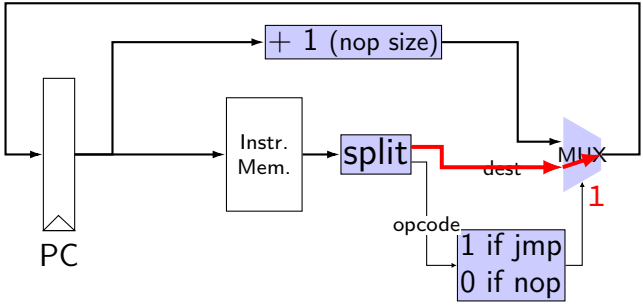

nop
1
0
jmp Dest
7
0
Dest

jmp+nop CPU


+ 1 (nop size)
nop
Instr.
Mem.
split
dest
MUX
0
opcode
1 if jmp
0 if nop
PC


nop
jmp Dest
1
0
7
0
Dest

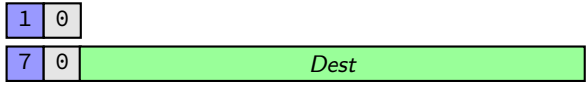
jmp+nop CPU




```
jmp dest
```

nop

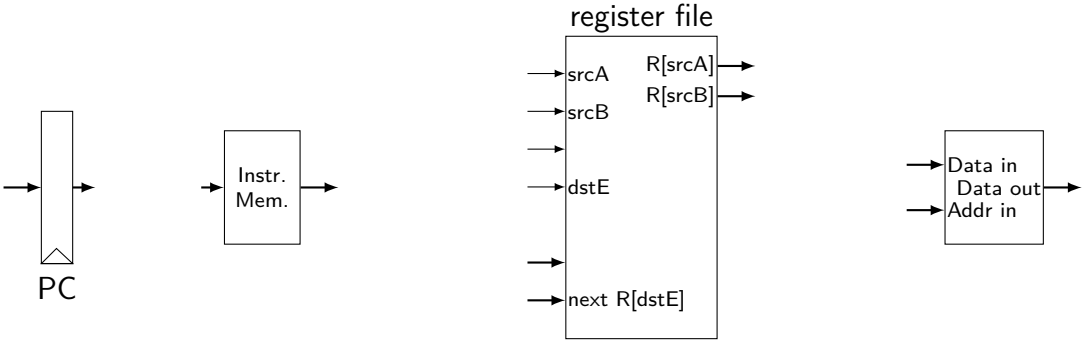
jmp *Dest*



simple ISA 4: mov-to-register

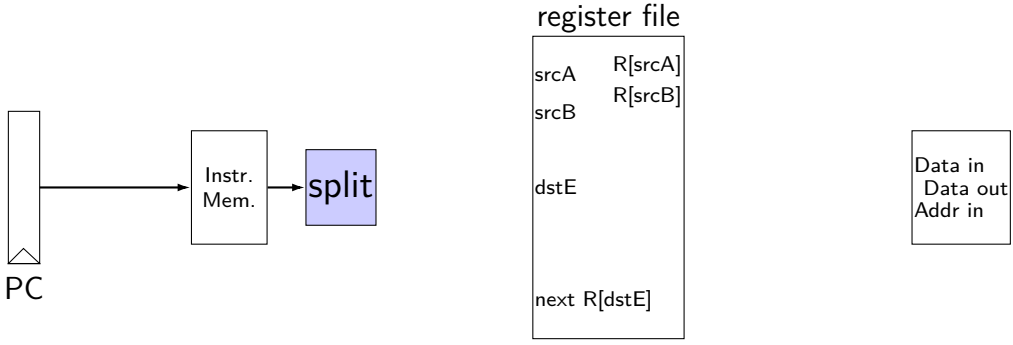
```
irmovq $constant, %rYY

rrmovq %rXX, %rYY

mrmovq 10(%rXX), %rYY
```

mov-to-register CPU



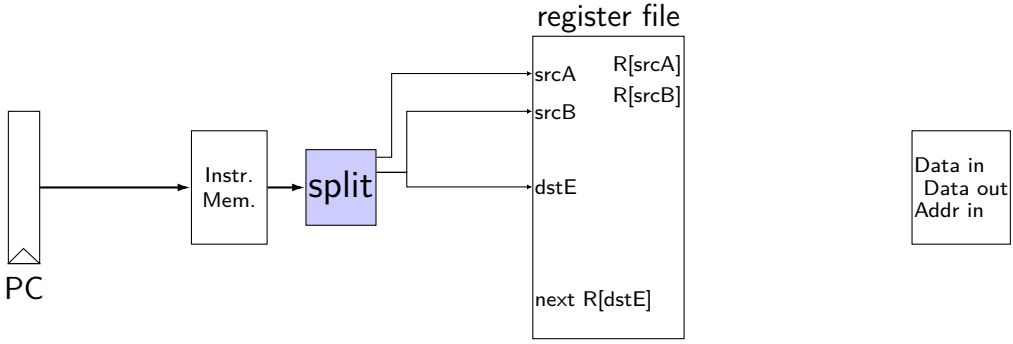
rrmovq *rA*, *rB*	2	0	*rA*	*rB*	
irmovq *V*, *rB*	3	0	F	*rB*	*V*
mrmovq *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

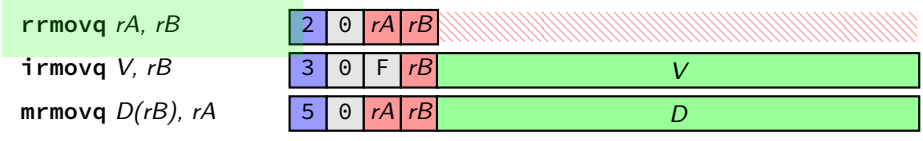
mov-to-register CPU



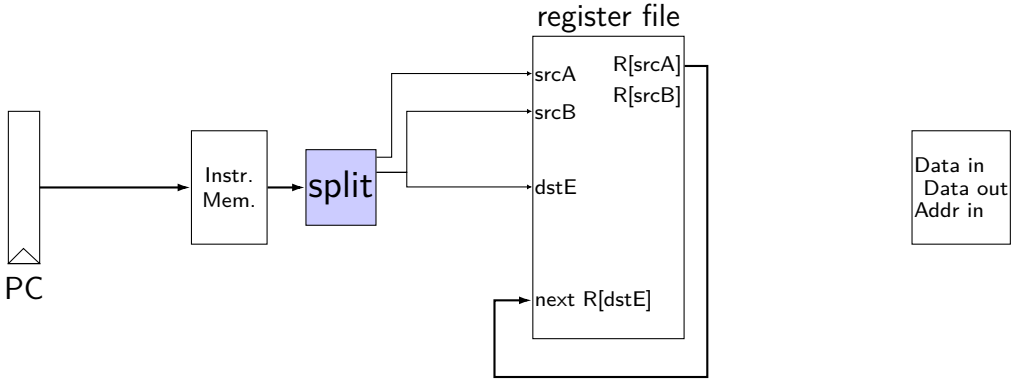
rrmovq *rA*, *rB*	2	0	*rA*	*rB*	
irmovq *V*, *rB*	3	0	F	*rB*	*V*
mrmovq *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

mov-to-register CPU



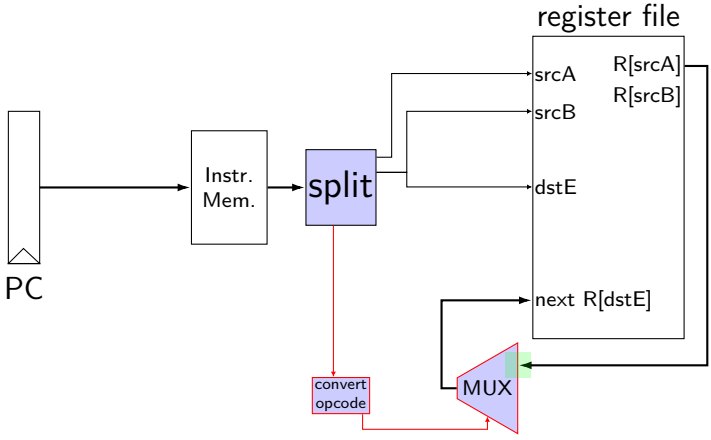


mov-to-register CPU



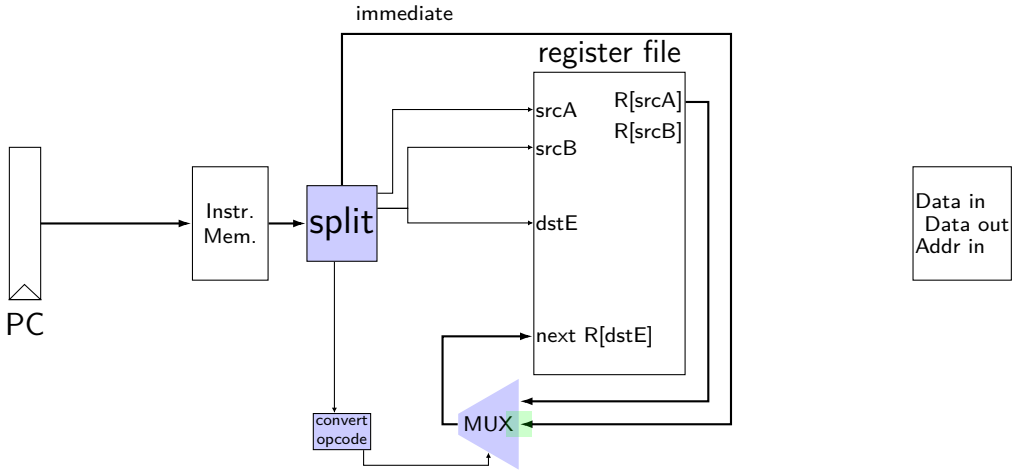
rrmovq *rA*, *rB*	2	0	*rA*	*rB*	
irmovq *V*, *rB*	3	0	F	*rB*	*V*
mrmovq *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

mov-to-register CPU



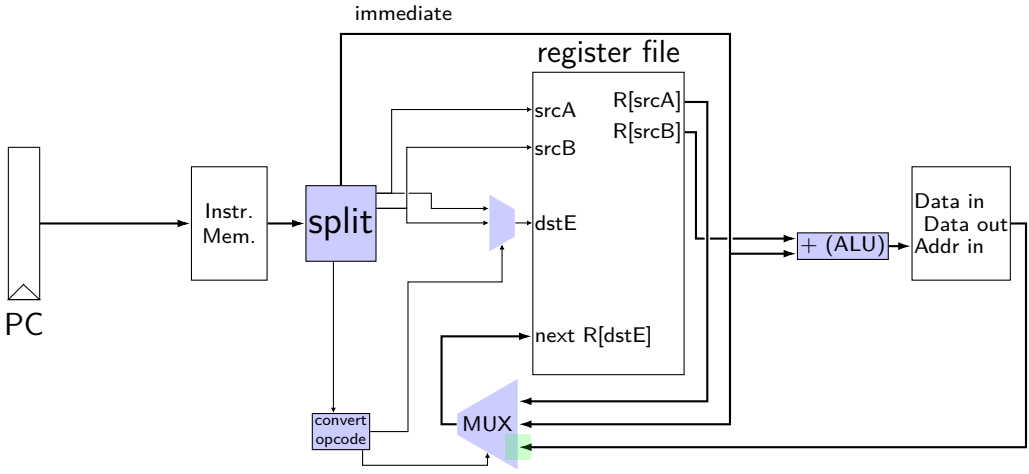
rrmovq *rA*, *rB*	2	0	*rA*	*rB*	
irmovq *V*, *rB*	3	0	F	*rB*	*V*
mrmovq *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

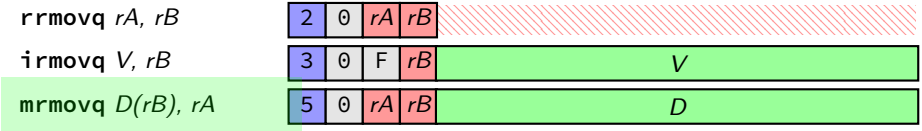
mov-to-register CPU



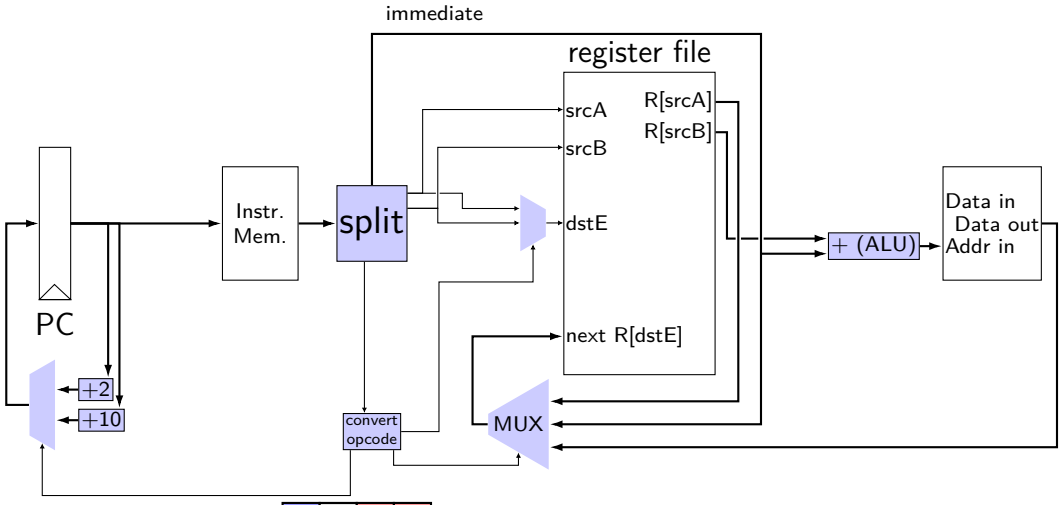
rrmovq *rA*, *rB*	2	0	*rA*	*rB*	
irmovq *V*, *rB*	3	0	F	*rB*	*V*
mrmovq *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

mov-to-register CPU





mov-to-register CPU

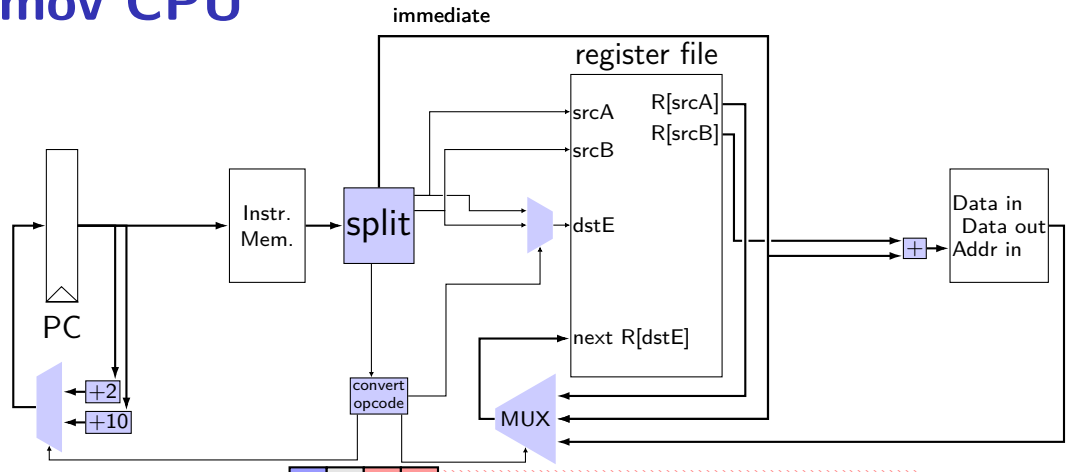


rrmovq *rA*, *rB*	2	0	*rA*	*rB*	
irmovq *V*, *rB*	3	0	F	*rB*	*V*
mrmovq *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

simple ISA 4B: mov

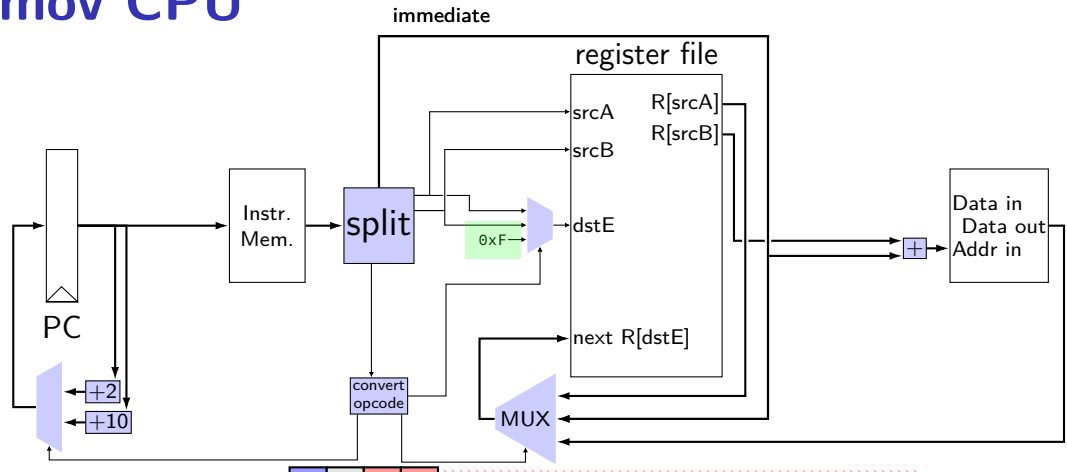
```
irmovq $constant, %rYY

rrmovq %rXX, %rYY

mrmovq 10(%rXX), %rYY

rmmovq %rXX, 10(%rYY)
```

mov CPU



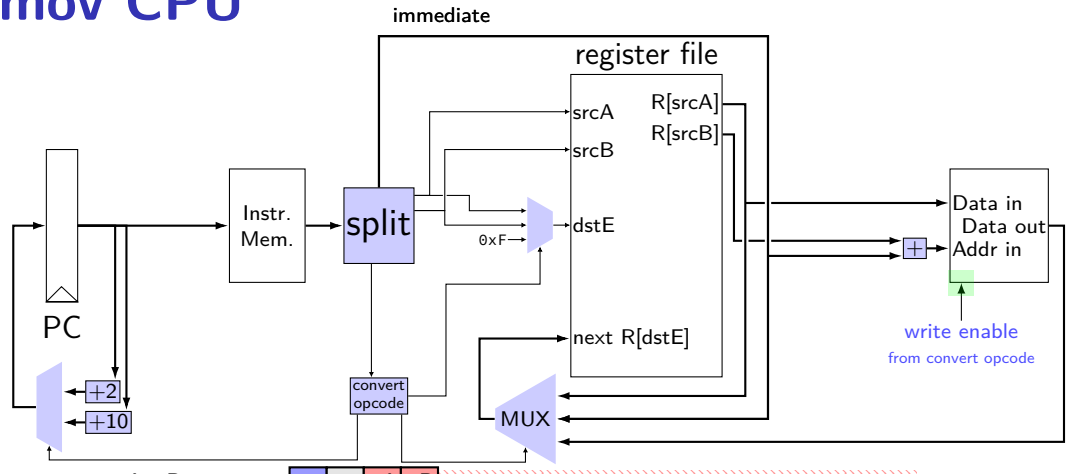
rrmovq *rA, rB*	2	0	*rA*	*rB*	
irmovq *V, rB*	3	0	F	*rB*	*V*
mrmovq *D(rB), rA*	5	0	*rA*	*rB*	*D*
rmmovq *rA, D(rB)*	4	0	*rA*	*rB*	*D*

mov CPU



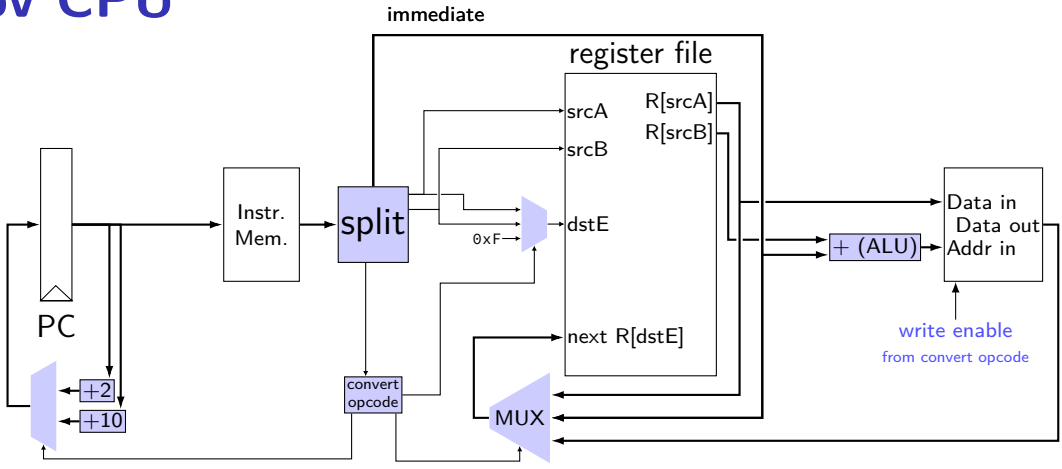
rrmovq *rA, rB*	2	0	*rA*	*rB*	
irmovq *V, rB*	3	0	F	*rB*	*V*
mrmovq *D(rB), rA*	5	0	*rA*	*rB*	*D*
rmmovq *rA, D(rB)*	4	0	*rA*	*rB*	*D*

mov CPU

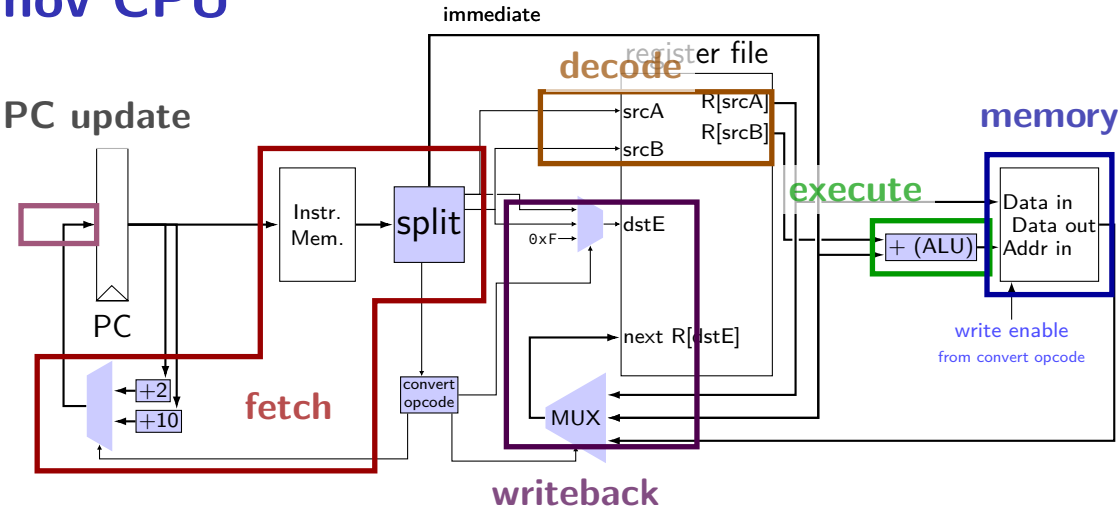


rrmovq *rA, rB*	2	0	*rA*	*rB*	
irmovq *V, rB*	3	0	F	*rB*	*V*
mrmovq *D(rB), rA*	5	0	*rA*	*rB*	*D*
rmmovq *rA, D(rB)*	4	0	*rA*	*rB*	*D*

mov CPU


immediate
register file
srcA
srcB
R[srcA]
R[srcB]
dstE
next R[dstE]
Instr.
Mem.
split
0xF
PC
+2
+10
convert
opcode
MUX
+ (ALU)
Data in
Data out
Addr in
write enable
from convert opcode

mov CPU


immediate
register file
decode
PC update
memory
srcA
srcB
R[srcA]
R[srcB]
execute
Instr.
Mem.
split
0xF
dstE
Data in
Data out
Addr in
+ (ALU)
PC
next R[dstE]
write enable
from convert opcode
+2
+10
fetch
convert
opcode
MUX
writeback

stages and time

fetch / decode / execute / memory / write back / PC update

Order when these events happen `pushq %rax` instruction:

1. instruction read
2. memory changes
3. `%rsp` changes
4. PC changes

Hint: recall how registers, register files, memory works

a. 1; then 2, 3, and 4 in any order
b. 1; then 2, 3, and 4 at almost the same time
c. 1; then 2; then 3; then 4
d. 1; then 3; then 2; then 4
e. 1; then 2; then 3 and 4 at almost the same time
f. something else

Summary

each instruction takes one cycle

divided into stages for design convenience

read values from previous cycle

send new values to state components

control what is sent with MUXes

Stages

conceptual division of instruction:

fetch — read instruction memory, split instruction, compute length

decode — read register file

execute — arithmetic (including of addresses)

memory — read or write data memory

write back — write to register file

PC update — compute next value of PC