ECE M116C - CS M151B Computer Architecture Systems - UCLA

Quiz 3

(Upload it to Gradescope)

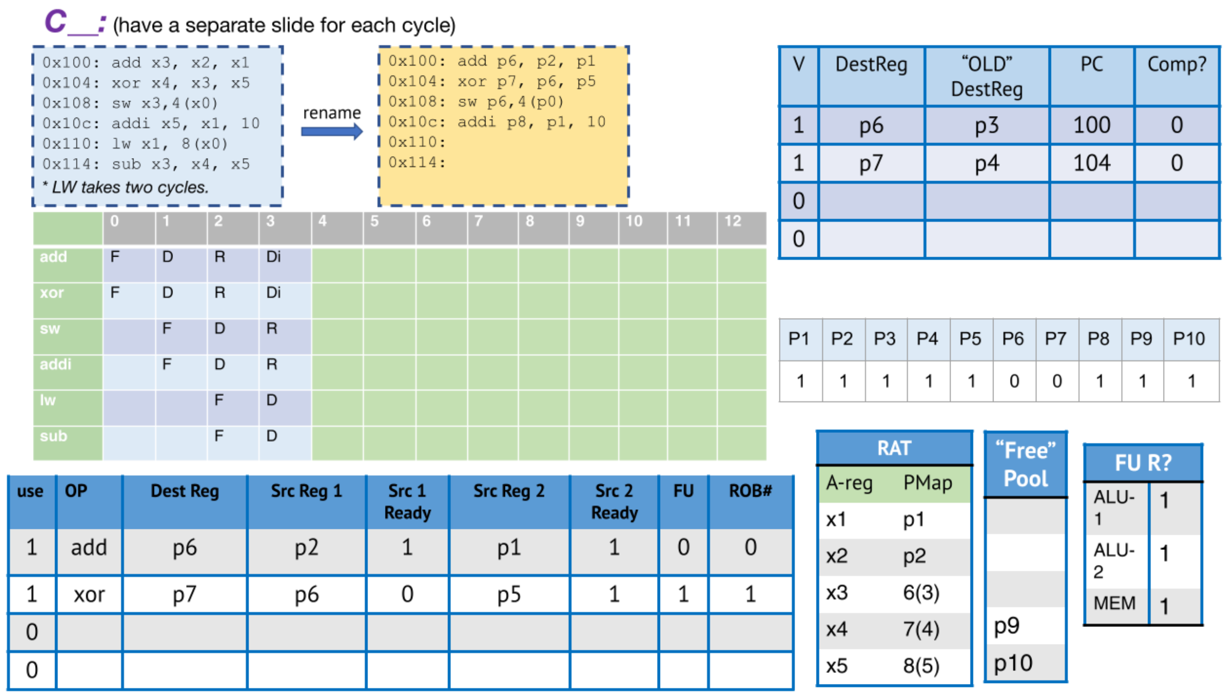
Fall 2021

Don't forget to sign in when you are leaving.

Q1. We have the following set of instructions (every instruction takes one cycle to complete except LW):

```
0x100: add x3, x2, x1
0x104: xor x4, x3, x5
0x108: sw x3,4(x0)
0x10c: addi x5, x1, 10
0x110: lw x1, 8(x0)
0x114: sub x3, x4, x5
```

We have a 2-issue out-of-order processor with the following units: (you can access the template by selecting the image and clicking on the link that appears). (link to ppt).



C__: (have a separate slide for each cycle)

```
0x100: add x3, x2, x1
0x104: xor x4, x3, x5
0x108: sw x3,4(x0)
0x10c: addi x5, x1, 10
0x110: lw x1, 8(x0)
0x114: sub x3, x4, x5
```

* LW takes two cycles.

rename

```
0x100: add p6, p2, p1
0x104: xor p7, p6, p5
0x108: sw p6,4(p0)
0x10c: addi p8, p1, 10
0x110:
0x114:
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
add	F	D	R	Di									
xor	F	D	R	Di									
sw		F	D	R									
addi		F	D	R									
lw			F	D									
sub			F	D									

V	DestReg	"OLD" DestReg	PC	Comp?
1	p6	p3	100	0
1	p7	p4	104	0
0				
0				

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
1	1	1	1	1	0	0	1	1	1

use	OP	Dest Reg	Src Reg 1	Src 1 Ready	Src Reg 2	Src 2 Ready	FU	ROB#
1	add	p6	p2	1	p1	1	0	0
1	xor	p7	p6	0	p5	1	1	1
0								
0								

RAT	
A-reg	PMap
x1	p1
x2	p2
x3	6(3)
x4	7(4)
x5	8(5)

"Free" Pool
p9
p10

FU R?	
ALU-1	1
ALU-2	1
MEM	1

Start from cycle 4, with Rename, and fill all the tables. In the green table, you should put R, Di (for dispatch), I (for issue, you should not use E), C, and Rt (for retire). If an instruction stays in one stage for multiple cycles (e.g., if the sources are not ready, or if RS is full), repeat the same character for it (e.g., F, D, R, Di, Di, I, C, Rt). Comp in ROB shows whether the instruction is completed. If an instruction can’t be retired, it should stay in C. Also, assume that you can retire a maximum of <u>two</u> instructions per cycle.

SOLUTION:

Cycle 4:
"add": issue the instruction (src1 ready, src2 ready, fu ready). We set ALU-1 to "not ready".

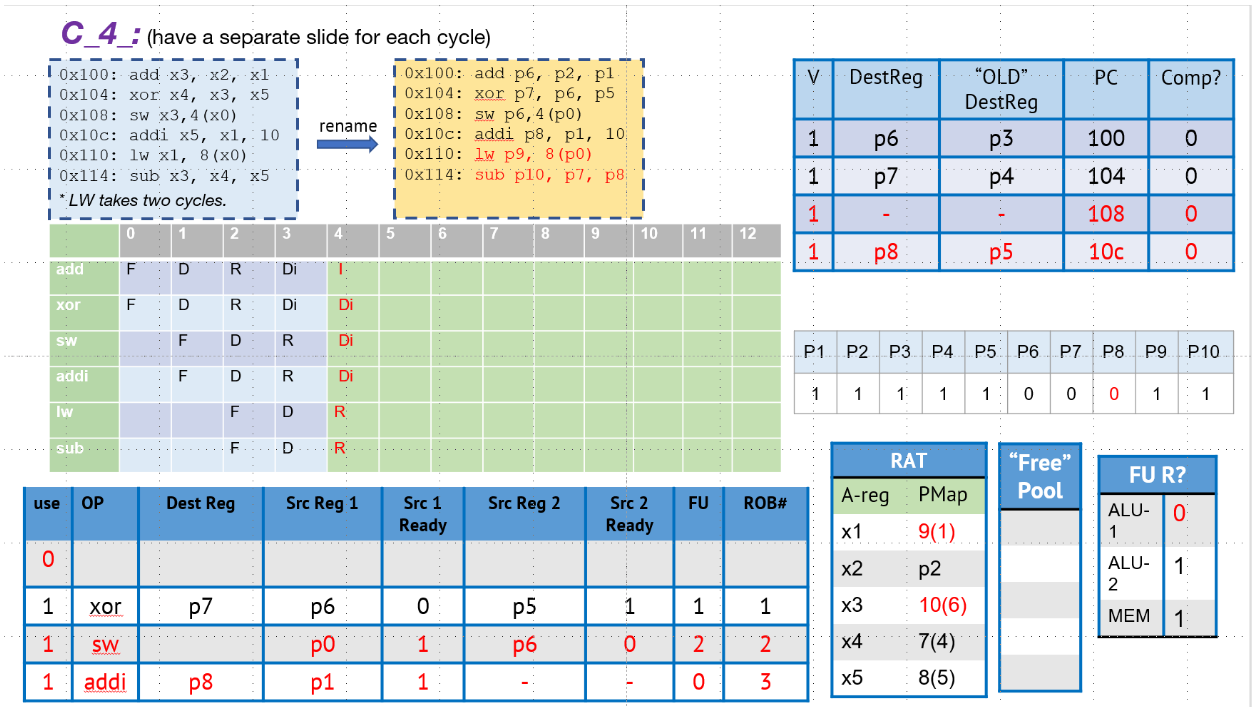
"xor": since "xor" depends on p6, we can't issue "xor".

"sw": we dispatch "sw". We place "sw" in the reservation station and the ROB (remember, "sw" doesn't have a destination register)

"addi": we dispatch "addi". We place "addi" in the reservation station and the ROB. We mark p8 as "not ready".

"lw": we rename "lw". We map the destination registers to a new physical register. For "lw", x1 maps to p9.

"sub": we rename "sub". For "sub", x3 maps to p10.



C_4_: (have a separate slide for each cycle)

```
0x100: add x3, x2, x1
0x104: xor x4, x3, x5
0x108: sw x3,4(x0)
0x10c: addi x5, x1, 10
0x110: lw x1, 8(x0)
0x114: sub x3, x4, x5
```
** LW takes two cycles.*

rename

```
0x100: add p6, p2, p1
0x104: xor p7, p6, p5
0x108: sw p6,4(p0)
0x10c: addi p8, p1, 10
0x110: lw p9, 8(p0)
0x114: sub p10, p7, p8
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
add	F	D	R	Di	I								
xor	F	D	R	Di	Di								
sw		F	D	R	Di								
addi		F	D	R	Di								
lw			F	D	R								
sub			F	D	R								

V	DestReg	"OLD" DestReg	PC	Comp?
1	p6	p3	100	0
1	p7	p4	104	0
1	-	-	108	0
1	p8	p5	10c	0

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
1	1	1	1	1	0	0	0	1	1

use	OP	Dest Reg	Src Reg 1	Src 1 Ready	Src Reg 2	Src 2 Ready	FU	ROB#
0								
1	xor	p7	p6	0	p5	1	1	1
1	sw		p0	1	p6	0	2	2
1	addi	p8	p1	1	-	-	0	3

RAT	
A-reg	PMap
x1	9(1)
x2	p2
x3	10(6)
x4	7(4)
x5	8(5)

"Free" Pool

FU R?	
ALU-1	0
ALU-2	1
MEM	1

Cycle 5:

"add": this instruction has completed, so we mark the ROB for this instruction as complete, and we mark p6 as "ready". We also release the ALU-1 functional unit.

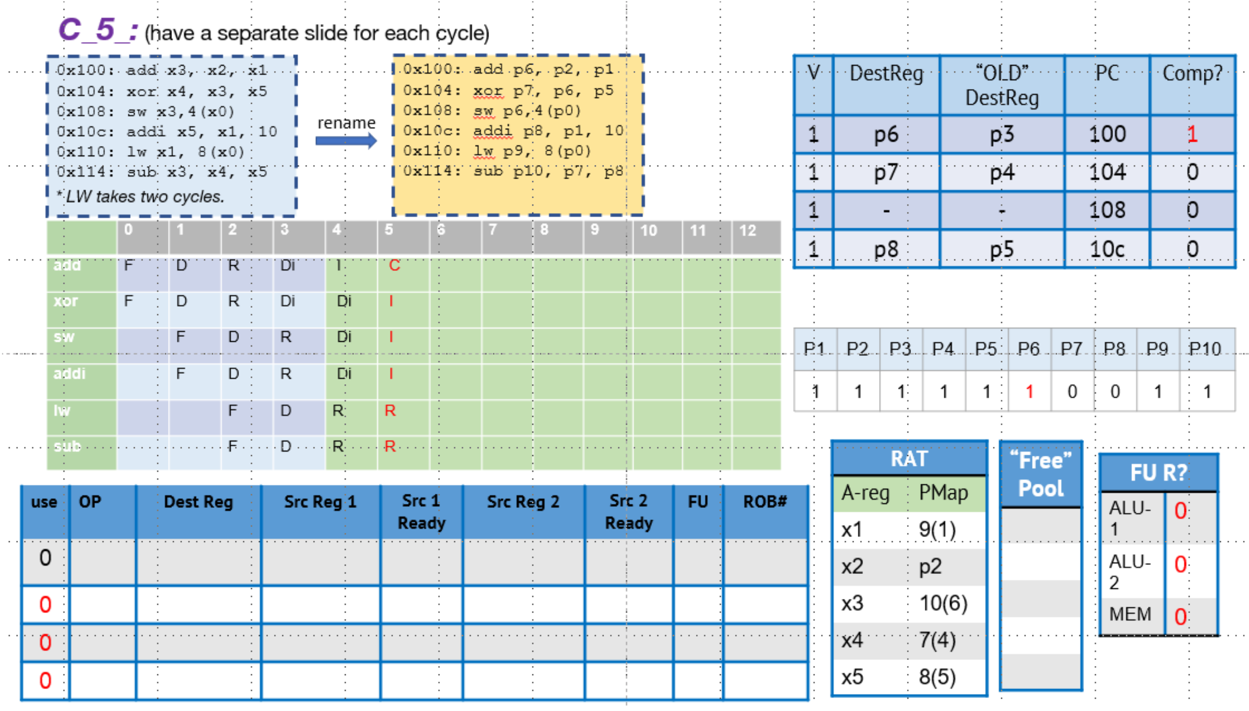
"xor": p6 has become ready, thus we can issue. We mark ALU-2 as "not ready", and clear "xor" from the reservation station.

"sw": we issue "sw". We mark MEM as "not ready" and clear "sw" from the reservation station.

"addi": we issue "addi". We mark ALU-1 as "not ready" and clear "addi" from the reservation station.

"lw": since there is no space in the ROB, we cannot dispatch "lw", thus "lw" remains in the rename stage (check alternate solution for other method)

"sub": we cannot dispatch "sub" for the same reason as "lw".



C_5_: (have a separate slide for each cycle)

```
0x100: add x3, x2, x1
0x104: xor x4, x3, x5
0x108: sw x3,4(x0)
0x10c: addi x5, x1, 10
0x110: lw x1, 8(x0)
0x114: sub x3, x4, x5
```

**LW takes two cycles.*

rename

```
0x100: add p6, p2, p1
0x104: xor p7, p6, p5
0x108: sw p6,4(p0)
0x10c: addi p8, p1, 10
0x110: lw p9, 8(p0)
0x114: sub p10, p7, p8
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
add	F	D	R	Di	I	C							
xor	F	D	R	Di	Di	I							
sw		F	D	R	Di	I							
addi		F	D	R	Di	I							
lw			F	D	R	R							
sub			F	D	R	R							

V	DestReg	"OLD" DestReg	PC	Comp?
1	p6	p3	100	1
1	p7	p4	104	0
1	-	-	108	0
1	p8	p5	10c	0

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
1	1	1	1	1	1	0	0	1	1

use	OP	Dest Reg	Src Reg 1	Src 1 Ready	Src Reg 2	Src 2 Ready	FU	ROB#
0								
0								
0								
0								

RAT	
A-reg	PMap
x1	9(1)
x2	p2
x3	10(6)
x4	7(4)
x5	8(5)

"Free" Pool

FU R?	
ALU-1	0
ALU-2	0
MEM	0

Cycle 6:

“add”: we retire “add”. We remove it from the ROB and free p3.

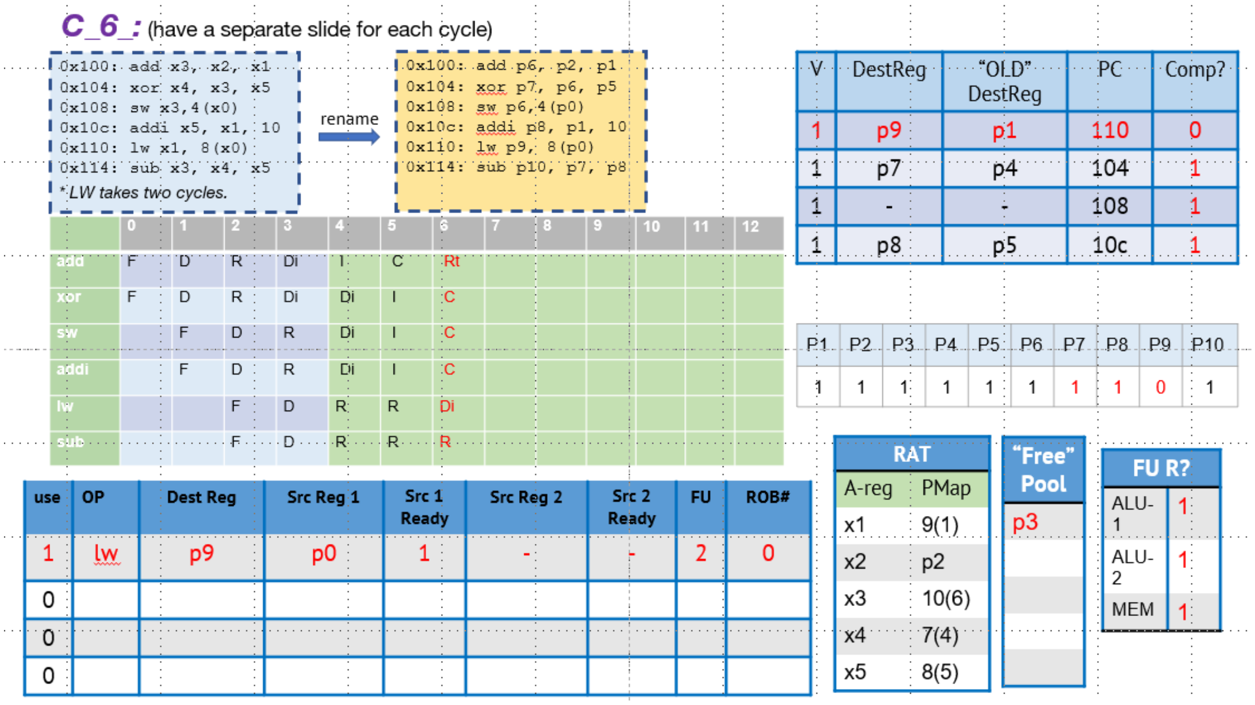
“xor”: we complete “xor”. We mark the ROB entry as “complete”. We mark p7 as “ready”. We release ALU-2.

“sw”: we complete “sw”. We mark the ROB entry as “complete”. We release MEM.

“addi”: we complete “addi”. We mark the ROB entry as “complete”. We release ALU-1.

“lw”: the ROB has an empty slot. Now we can dispatch “lw”. We place “lw” in the reservation station, mark p9 as “not ready”, and place “lw” in the ROB.

“sub”: “sub” is still stalled at the rename stage, as the ROB is full.



C_6_: (have a separate slide for each cycle)

```
0x100: add x3, x2, x1
0x104: xor x4, x3, x5
0x108: sw x3,4(x0)
0x10c: addi x5, x1, 10
0x110: lw x1, 8(x0)
0x114: sub x3, x4, x5
```

LW takes two cycles.

rename

```
0x100: add p6, p2, p1
0x104: xor p7, p6, p5
0x108: sw p6,4(p0)
0x10c: addi p8, p1, 10
0x110: lw p9, 8(p0)
0x114: sub p10, p7, p8
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
add	F	D	R	Di	I	C	Rt						
xor	F	D	R	Di	Di	I	C						
sw		F	D	R	Di	I	C						
addi		F	D	R	Di	I	C						
lw			F	D	R	R	Di						
sub			F	D	R	R	R						

use	OP	Dest Reg	Src Reg 1	Src 1 Ready	Src Reg 2	Src 2 Ready	FU	ROB#
1	lw	p9	p0	1	-	-	2	0
0								
0								
0								

V	DestReg	“OLD” DestReg	PC	Comp?
1	p9	p1	110	0
1	p7	p4	104	1
1	-	-	108	1
1	p8	p5	10c	1

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
1	1	1	1	1	1	1	1	0	1

RAT	
A-reg	PMap
x1	9(1)
x2	p2
x3	10(6)
x4	7(4)
x5	8(5)

“Free” Pool
p3

FU R?	
ALU-1	1
ALU-2	1
MEM	1

Cycle 7:

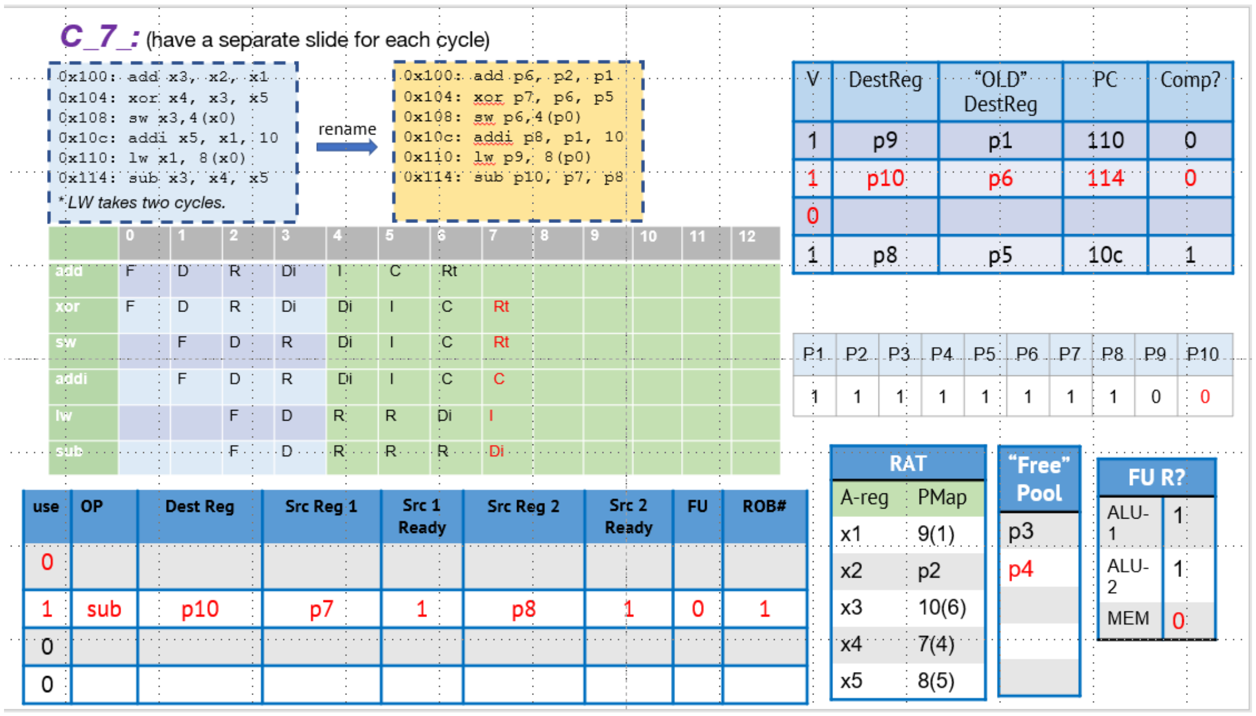
"xor": we retire "xor". We remove it from the ROB and free p4.

"sw": we retire "sw". We remove it from the ROB.

"addi": since we already retired 2 instructions (defined in problem), we cannot retire "addi". Thus, we need to stay at complete.

"lw": we issue "lw". Note that "lw" takes 2 cycles to finish in the issue stage. We set the functional unit MEM = 0 and clear "lw" from the reservation station.

"sub": the ROB has an empty slot. Now we can dispatch "sub". We place "sub" in the reservation station, mark p10 as "not ready", and place "sub" in the ROB.



C_7_: (have a separate slide for each cycle)

```
0x100: add x3, x2, x1
0x104: xor x4, x3, x5
0x108: sw x3,4(x0)
0x10c: addi x5, x1, 10
0x110: lw x1, 8(x0)
0x114: sub x3, x4, x5
*LW takes two cycles.
```

rename

```
0x100: add p6, p2, p1
0x104: xor p7, p6, p5
0x108: sw p6,4(p0)
0x10c: addi p8, p1, 10
0x110: lw p9, 8(p0)
0x114: sub p10, p7, p8
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
add	F	D	R	Di	I	C	Rt						
xor	F	D	R	Di	Di	I	C	Rt					
sw		F	D	R	Di	I	C	Rt					
addi		F	D	R	Di	I	C	C					
lw			F	D	R	R	Di	I					
sub			F	D	R	R	R	Di					

V	DestReg	"OLD" DestReg	PC	Comp?
1	p9	p1	110	0
1	p10	p6	114	0
0				
1	p8	p5	10c	1

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
1	1	1	1	1	1	1	1	0	0

use	OP	Dest Reg	Src Reg 1	Src 1 Ready	Src Reg 2	Src 2 Ready	FU	ROB#
0								
1	sub	p10	p7	1	p8	1	0	1
0								
0								

RAT

A-reg	PMap
x1	9(1)
x2	p2
x3	10(6)
x4	7(4)
x5	8(5)

"Free" Pool
p3
p4

FU R?

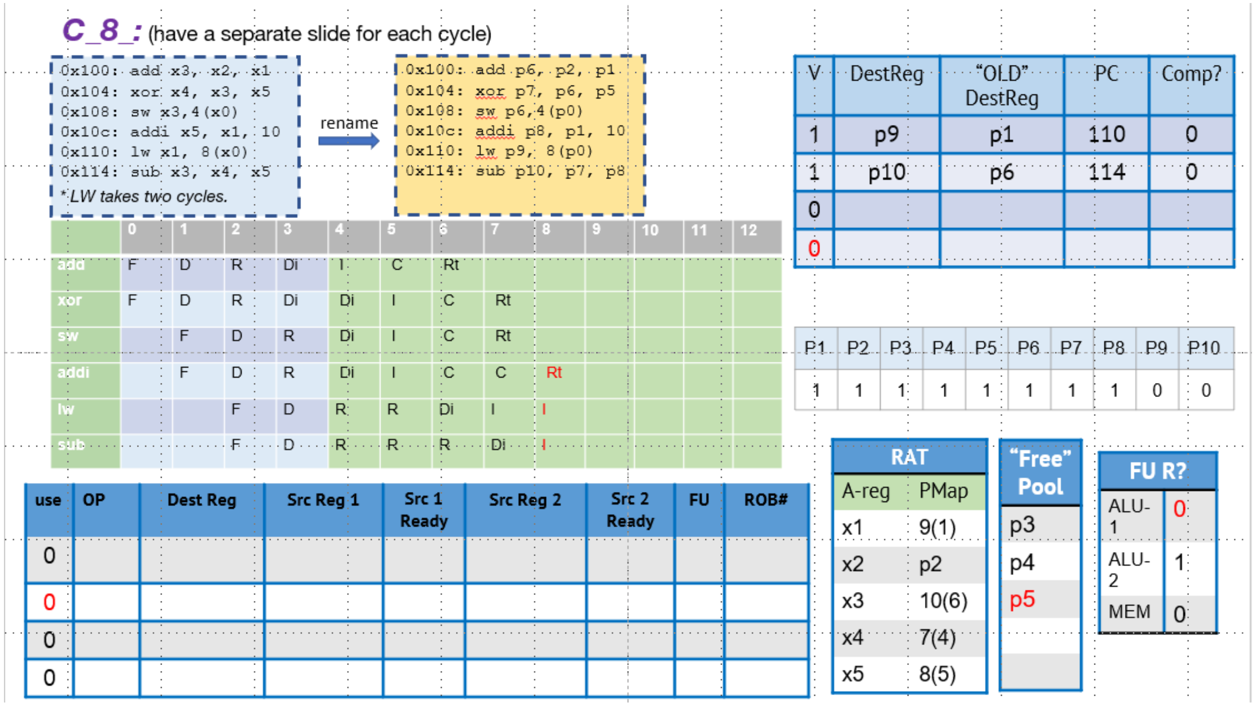
ALU-1	1
ALU-2	1
MEM	0

Cycle 8

"addi": we can now retire "addi". We remove it from the ROB and release p5.

"lw": "lw" needs one more cycle to complete.

"sub": we issue "sub", as the sources are ready and ALU-1 is ready. We set ALU-1 = 0 and clear "sub" from the reservation station.



C_8_: (have a separate slide for each cycle)

```
0x100: add x3, x2, x1
0x104: xor x4, x3, x5
0x108: sw x3,4(x0)
0x10c: addi x5, x1, 10
0x110: lw x1, 8(x0)
0x114: sub x3, x4, x5
```
**LW takes two cycles.*

rename

```
0x100: add p6, p2, p1
0x104: xor p7, p6, p5
0x108: sw p6,4(p0)
0x10c: addi p8, p1, 10
0x110: lw p9, 8(p0)
0x114: sub p10, p7, p8
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
add	F	D	R	Di	I	C	Rt						
xor	F	D	R	Di	Di	I	C	Rt					
sw		F	D	R	Di	I	C	Rt					
addi		F	D	R	Di	I	C	C	Rt				
lw			F	D	R	R	Di	I	I				
sub			F	D	R	R	R	Di	I				

V	DestReg	"OLD" DestReg	PC	Comp?
1	p9	p1	110	0
1	p10	p6	114	0
0				
0				

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
1	1	1	1	1	1	1	1	0	0

use	OP	Dest Reg	Src Reg 1	Src 1 Ready	Src Reg 2	Src 2 Ready	FU	ROB#
0								
0								
0								
0								

RAT	
A-reg	PMap
x1	9(1)
x2	p2
x3	10(6)
x4	7(4)
x5	8(5)

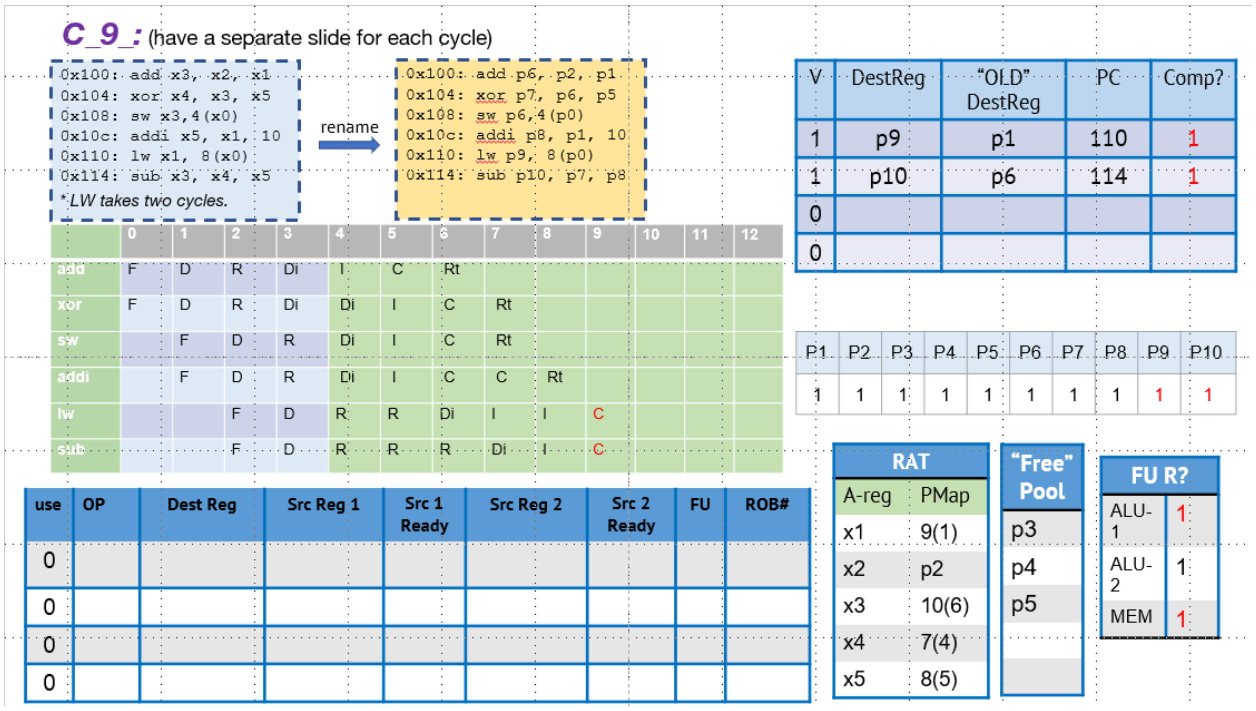
"Free" Pool
p3
p4
p5

FU R?	
ALU-1	0
ALU-2	1
MEM	0

Cycle 9:

“lw”: “lw” is complete after 2 cycles in issue. We mark “lw” as complete in the ROB. We mark p9 as “ready”, and release Mem.

“sub”: “sub” is complete. We mark “sub” as complete in the ROB. We mark p10 as “ready”, and release ALU-1.



C_9_: (have a separate slide for each cycle)

```
0x100: add x3, x2, x1
0x104: xor x4, x3, x5
0x108: sw x3,4(x0)
0x10c: addi x5, x1, 10
0x110: lw x1, 8(x0)
0x114: sub x3, x4, x5
```

**LW takes two cycles.*

rename

```
0x100: add p6, p2, p1
0x104: xor p7, p6, p5
0x108: sw p6,4(p0)
0x10c: addi p8, p1, 10
0x110: lw p9, 8(p0)
0x114: sub p10, p7, p8
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
add	F	D	R	Di	I	C	Rt						
xor	F	D	R	Di	Di	I	C	Rt					
sw		F	D	R	Di	I	C	Rt					
addi		F	D	R	Di	I	C	C	Rt				
lw			F	D	R	R	Di	I	I	C			
sub			F	D	R	R	R	Di	I	C			

V	DestReg	“OLD” DestReg	PC	Comp?
1	p9	p1	110	1
1	p10	p6	114	1
0				
0				

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
1	1	1	1	1	1	1	1	1	1

use	OP	Dest Reg	Src Reg 1	Src 1 Ready	Src Reg 2	Src 2 Ready	FU	ROB#
0								
0								
0								
0								

RAT

A-reg	PMap
x1	9(1)
x2	p2
x3	10(6)
x4	7(4)
x5	8(5)

“Free” Pool

p3
p4
p5

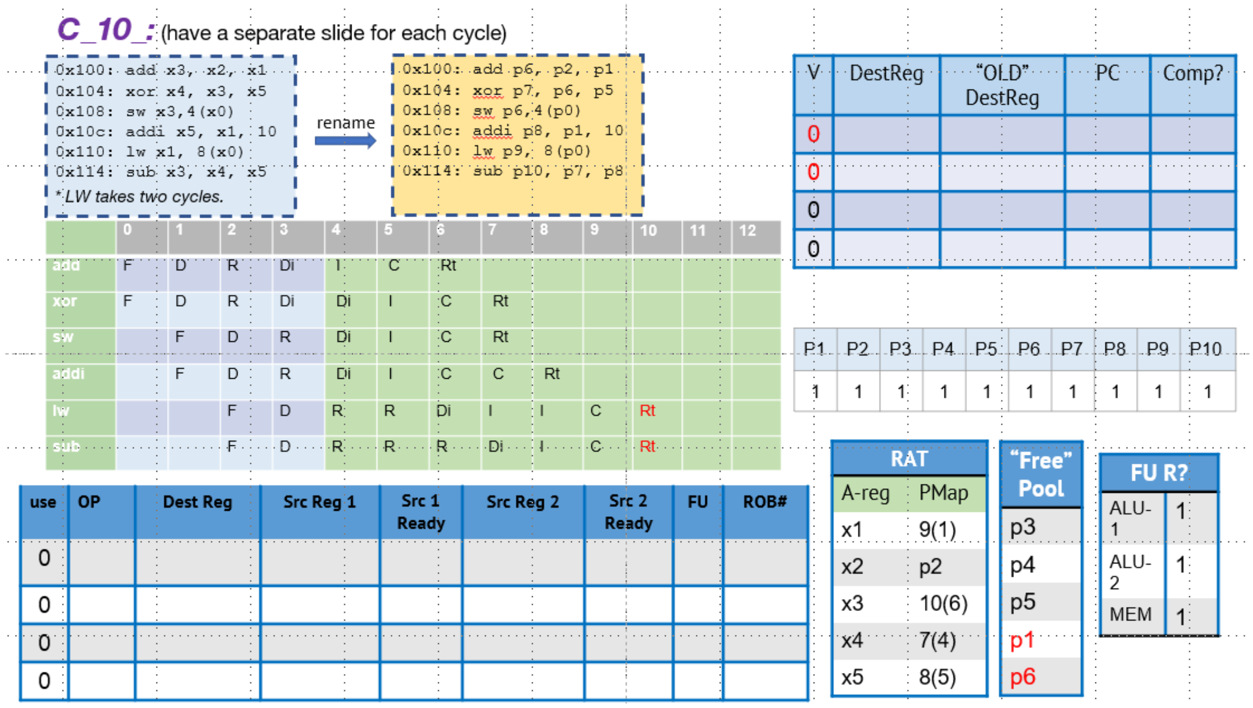
FU R?

ALU-1	1
ALU-2	1
MEM	1

Cycle 10:

“lw”: we retire “lw”. We remove “lw” from the ROB and release p1.

“sub”: we retire “sub”. We remove “sub” from the ROB and release p6.



C_10_: (have a separate slide for each cycle)

```
0x100: add x3, x2, x1
0x104: xor x4, x3, x5
0x108: sw x3,4(x0)
0x10c: addi x5, x1, 10
0x110: lw x1, 8(x0)
0x114: sub x3, x4, x5
```

**LW takes two cycles.*

rename

```
0x100: add p6, p2, p1
0x104: xor p7, p6, p5
0x108: sw p6,4(p0)
0x10c: addi p8, p1, 10
0x110: lw p9, 8(p0)
0x114: sub p10, p7, p8
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
add	F	D	R	Di	I	C	Rt						
xor	F	D	R	Di	Di	I	C	Rt					
sw		F	D	R	Di	I	C	Rt					
addi		F	D	R	Di	I	C	C	Rt				
lw			F	D	R	R	Di	I	I	C	Rt		
sub			F	D	R	R	R	Di	I	C	Rt		

V	DestReg	“OLD” DestReg	PC	Comp?
0				
0				
0				
0				

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
1	1	1	1	1	1	1	1	1	1

use	OP	Dest Reg	Src Reg 1	Src 1 Ready	Src Reg 2	Src 2 Ready	FU	ROB#
0								
0								
0								
0								

RAT

A-reg	PMap
x1	9(1)
x2	p2
x3	10(6)
x4	7(4)
x5	8(5)

“Free” Pool
p3
p4
p5
p1
p6

FU R?	
ALU-1	1
ALU-2	1
MEM	1

<u>ALTERNATE SOLUTION</u>

In this alternate solution, we dispatch "lw" and "sub", even though there is no space in the ROB. We wait to issue these instructions until a space in the ROB opens. The previous solution is the "correct" solution but this is a design choice.

Cycle 4:
"add": issue the instruction (src1 ready, src2 ready, fu ready). We set ALU-1 to "not ready".

"xor": since "xor" depends on p6, we can't issue "xor".

"sw": we dispatch "sw". We place "sw" in the reservation station and the ROB (remember, "sw" doesn't have a destination register)

"addi": we dispatch "addi". We place "addi" in the reservation station and the ROB. We mark p8 as "not ready".

"lw": we rename "lw". We map the destination registers to a new physical register. For "lw", x1 maps to p9.

"sub": we rename "sub". For "sub", x3 maps to p10.

C_4_: (have a separate slide for each cycle)

```
0x100: add x3, x2, x1
0x104: xor x4, x3, x5
0x108: sw x3,4(x0)
0x10c: addi x5, x1, 10
0x110: lw x1, 8(x0)
0x114: sub x3, x4, x5
```
** LW takes two cycles.*

rename

```
0x100: add p6, p2, p1
0x104: xor p7, p6, p5
0x108: sw p6,4(p0)
0x10c: addi p8, p1, 10
0x110: lw p9, 8(p0)
0x114: sub p10, p7, p8
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
add	F	D	R	Di	I								
xor	F	D	R	Di	Di								
sw		F	D	R	Di								
addi		F	D	R	Di								
lw			F	D	R								
sub			F	D	R								

V	DestReg	"OLD" DestReg	PC	Comp?
1	p6	p3	100	0
1	p7	p4	104	0
1	-	-	108	0
1	p8	p5	10c	0

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
1	1	1	1	1	0	0	0	1	1

use	OP	Dest Reg	Src Reg 1	Src 1 Ready	Src Reg 2	Src 2 Ready	FU	ROB#
0								
1	xor	p7	p6	0	p5	1	1	1
1	sw		p0	1	p6	0	2	2
1	addi	p8	p1	1	-	-	0	3

RAT	
A-reg	PMap
x1	9(1)
x2	p2
x3	10(6)
x4	7(4)
x5	8(5)

"Free" Pool

FU R?	
ALU-1	0
ALU-2	1
MEM	1

Cycle 5:

"add": this instruction has completed, so we mark the ROB for this instruction as complete, and we mark p6 as "ready". We also release the ALU-1 functional unit.

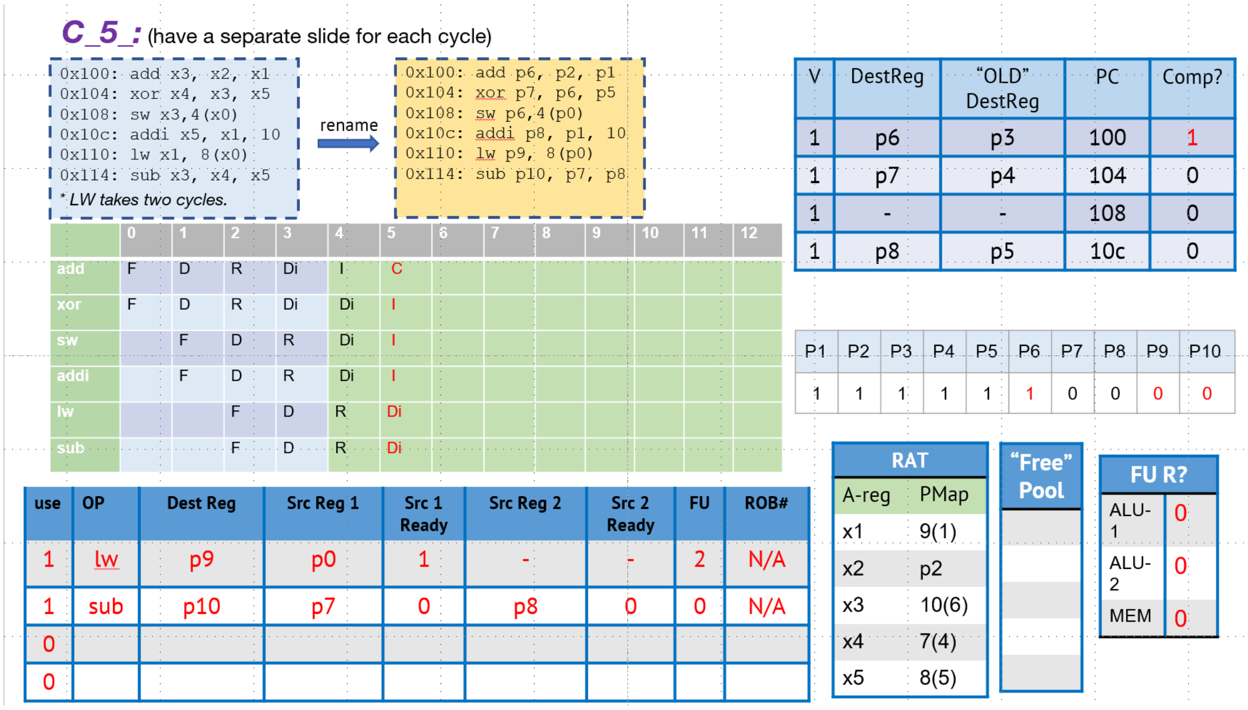
"xor": p6 has become ready, thus we can issue. We mark ALU-2 as "not ready" and clear "xor" from the reservation station.

"sw": we issue "sw". We mark MEM as "not ready" and clear "sw" from the reservation station.

"addi": we issue "addi". We mark ALU-1 as "not ready" and clear "addi" from the reservation station.

"lw": we dispatch "lw", placing it in the reservation station. Since the ROB is currently full, we need to stall "lw" until "lw" can be placed in the ROB. We mark p9 as "not ready".

"sub": we dispatch "sub", placing it in the reservation station (can't place in ROB, same reason as for "lw"). We mark p10 as "not ready"



C_5_: (have a separate slide for each cycle)

```
0x100: add x3, x2, x1
0x104: xor x4, x3, x5
0x108: sw x3,4(x0)
0x10c: addi x5, x1, 10
0x110: lw x1, 8(x0)
0x114: sub x3, x4, x5
```

* *LW takes two cycles.*

rename

```
0x100: add p6, p2, p1
0x104: xor p7, p6, p5
0x108: sw p6,4(p0)
0x10c: addi p8, p1, 10
0x110: lw p9, 8(p0)
0x114: sub p10, p7, p8
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
add	F	D	R	Di	I	C							
xor	F	D	R	Di	Di	I							
sw		F	D	R	Di	I							
addi		F	D	R	Di	I							
lw			F	D	R	Di							
sub			F	D	R	Di							

V	DestReg	"OLD" DestReg	PC	Comp?
1	p6	p3	100	1
1	p7	p4	104	0
1	-	-	108	0
1	p8	p5	10c	0

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
1	1	1	1	1	1	0	0	0	0

use	OP	Dest Reg	Src Reg 1	Src 1 Ready	Src Reg 2	Src 2 Ready	FU	ROB#
1	lw	p9	p0	1	-	-	2	N/A
1	sub	p10	p7	0	p8	0	0	N/A
0								
0								

RAT	
A-reg	PMap
x1	9(1)
x2	p2
x3	10(6)
x4	7(4)
x5	8(5)

"Free" Pool

FU R?	
ALU-1	0
ALU-2	0
MEM	0

Cycle 6:

“add”: we retire “add”. We remove it from the ROB and free p3.

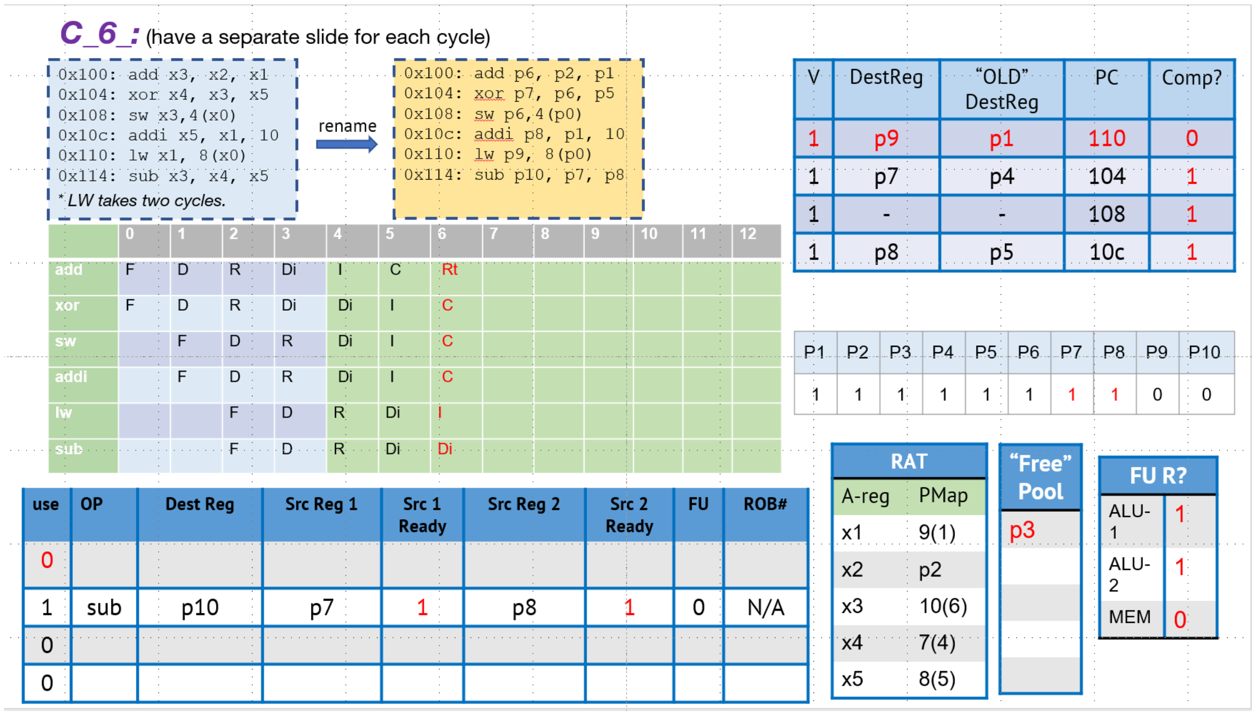
“xor”: we complete “xor”. We mark the ROB entry as “complete”. We mark p7 as “ready”. We release ALU-2.

“sw”: we complete “sw”. We mark the ROB entry as “complete”. We release MEM.

“addi”: we complete “addi”. We mark the ROB entry as “complete”. We release ALU-1.

“lw”: we place “lw” in the ROB now that there is an empty space. We then issue “lw”. We mark MEM as “not ready” and clear “lw” from the reservation station.

“sub”: we need to stall in “Di” as there is no space in the ROB. The sources for “sub” are now ready though.



C_6_: (have a separate slide for each cycle)

```
0x100: add x3, x2, x1
0x104: xor x4, x3, x5
0x108: sw x3,4(x0)
0x10c: addi x5, x1, 10
0x110: lw x1, 8(x0)
0x114: sub x3, x4, x5
```

** LW takes two cycles.*

rename

```
0x100: add p6, p2, p1
0x104: xor p7, p6, p5
0x108: sw p6,4(p0)
0x10c: addi p8, p1, 10
0x110: lw p9, 8(p0)
0x114: sub p10, p7, p8
```

V	DestReg	“OLD” DestReg	PC	Comp?
1	p9	p1	110	0
1	p7	p4	104	1
1	-	-	108	1
1	p8	p5	10c	1

	0	1	2	3	4	5	6	7	8	9	10	11	12
add	F	D	R	Di	I	C	Rt						
xor	F	D	R	Di	Di	I	C						
sw		F	D	R	Di	I	C						
addi		F	D	R	Di	I	C						
lw			F	D	R	Di	I						
sub			F	D	R	Di	Di						

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
1	1	1	1	1	1	1	1	0	0

use	OP	Dest Reg	Src Reg 1	Src 1 Ready	Src Reg 2	Src 2 Ready	FU	ROB#
0								
1	sub	p10	p7	1	p8	1	0	N/A
0								
0								

RAT

A-reg	PMap
x1	9(1)
x2	p2
x3	10(6)
x4	7(4)
x5	8(5)

“Free” Pool
p3

FU R?

ALU-1	1
ALU-2	1
MEM	0

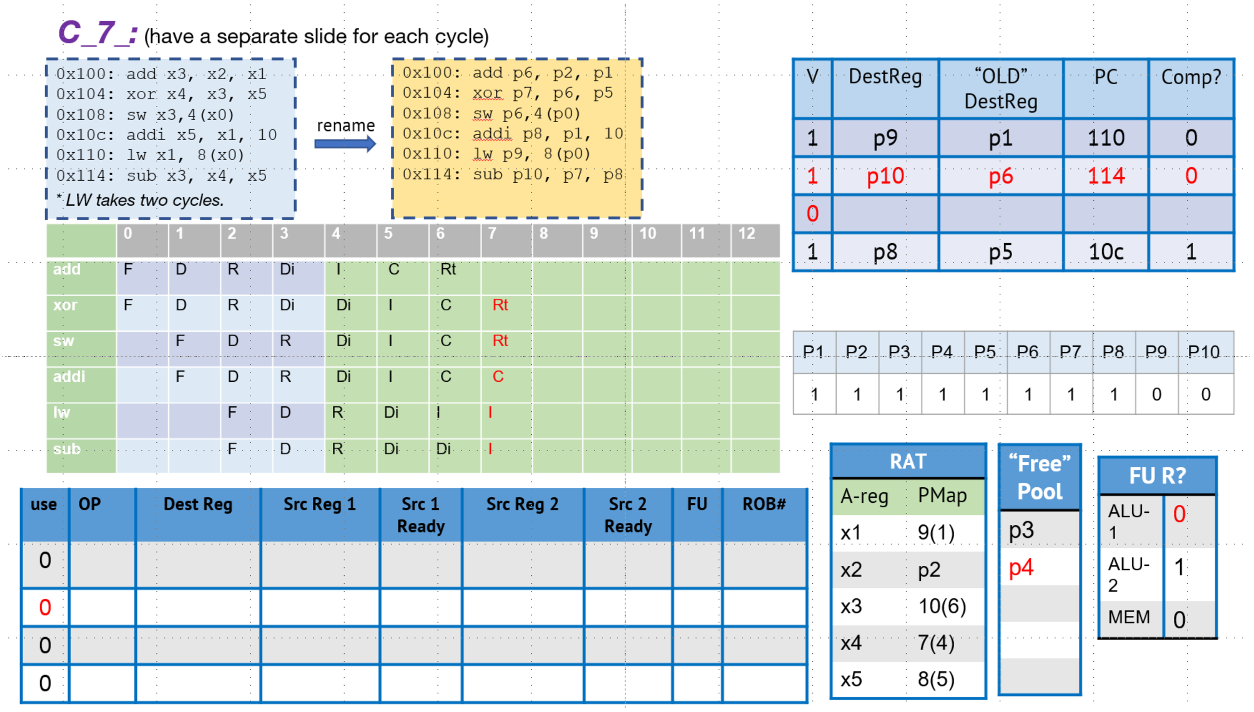
Cycle 7:
"xor": we retire "xor". We remove it from the ROB and free p4.

"sw": we retire "sw". We remove it from the ROB.

"addi": since we already retired 2 instructions (defined in problem), we cannot retire "addi". Thus, we need to stay at complete.

"lw": since "lw" takes two cycles (defined in problem), we are still in the issue stage.

"sub": now that there is space in the ROB, we place "sub" in the ROB, then we issue "sub" since the sources are ready and the FU is ready. We mark ALU-1 as "not ready" and clear "sub" from the reservation station.



C_7_: (have a separate slide for each cycle)

```
0x100: add x3, x2, x1
0x104: xor x4, x3, x5
0x108: sw x3,4(x0)
0x10c: addi x5, x1, 10
0x110: lw x1, 8(x0)
0x114: sub x3, x4, x5
```
** LW takes two cycles.*

rename

```
0x100: add p6, p2, p1
0x104: xor p7, p6, p5
0x108: sw p6,4(p0)
0x10c: addi p8, p1, 10
0x110: lw p9, 8(p0)
0x114: sub p10, p7, p8
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
add	F	D	R	Di	I	C	Rt						
xor	F	D	R	Di	Di	I	C	Rt					
sw		F	D	R	Di	I	C	Rt					
addi		F	D	R	Di	I	C	C					
lw			F	D	R	Di	I	I					
sub			F	D	R	Di	Di	I					

V	DestReg	"OLD" DestReg	PC	Comp?
1	p9	p1	110	0
1	p10	p6	114	0
0				
1	p8	p5	10c	1

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
1	1	1	1	1	1	1	1	0	0

use	OP	Dest Reg	Src Reg 1	Src 1 Ready	Src Reg 2	Src 2 Ready	FU	ROB#
0								
0								
0								
0								

RAT	
A-reg	PMap
x1	9(1)
x2	p2
x3	10(6)
x4	7(4)
x5	8(5)

"Free" Pool
p3
p4

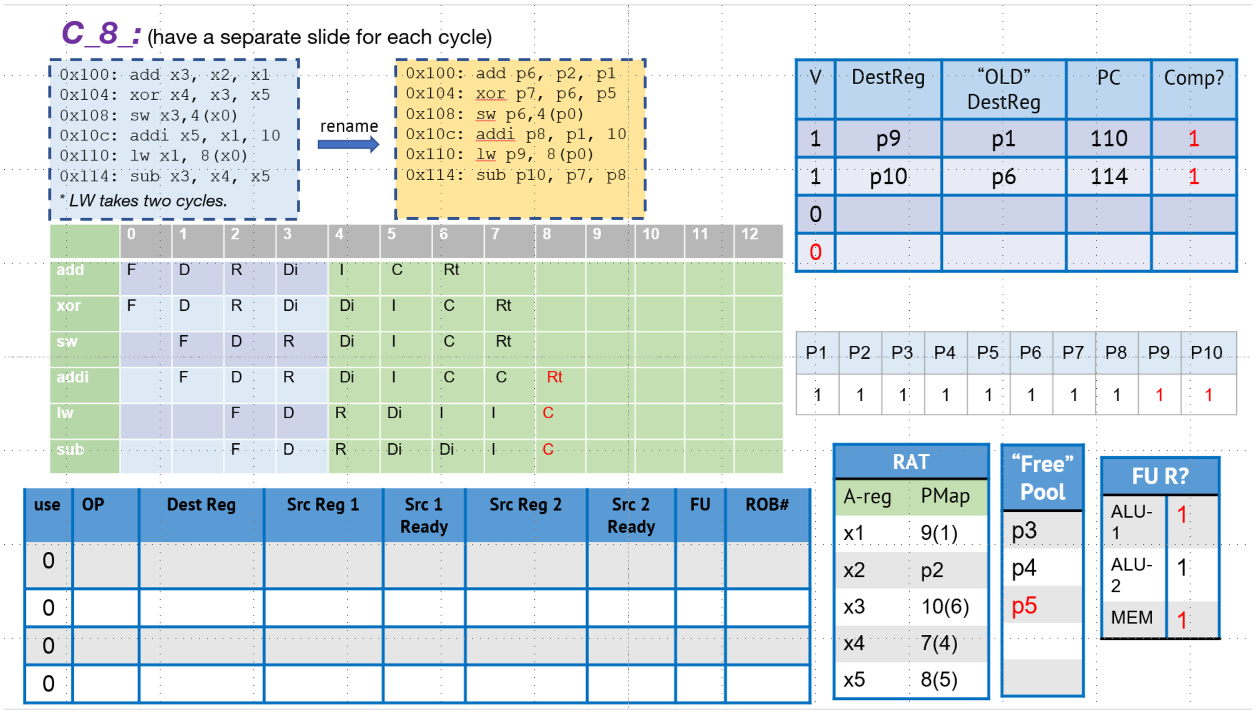
FU R?	
ALU-1	0
ALU-2	1
MEM	0

Cycle 8:

"addi": we can now retire "addi". We remove it from the ROB and release p5.

"lw": "lw" is complete after 2 cycles in issue. We mark "lw" as complete in the ROB. We mark p9 as "ready", and release Mem.

"sub": "sub" is complete. We mark "sub" as complete in the ROB. We mark p10 as "ready", and release ALU-1.



C_8_: (have a separate slide for each cycle)

```
0x100: add x3, x2, x1
0x104: xor x4, x3, x5
0x108: sw x3,4(x0)
0x10c: addi x5, x1, 10
0x110: lw x1, 8(x0)
0x114: sub x3, x4, x5
```

* *LW takes two cycles.*

rename

```
0x100: add p6, p2, p1
0x104: xor p7, p6, p5
0x108: sw p6,4(p0)
0x10c: addi p8, p1, 10
0x110: lw p9, 8(p0)
0x114: sub p10, p7, p8
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
add	F	D	R	Di	I	C	Rt						
xor	F	D	R	Di	Di	I	C	Rt					
sw		F	D	R	Di	I	C	Rt					
addi		F	D	R	Di	I	C	C	Rt				
lw			F	D	R	Di	I	I	C				
sub			F	D	R	Di	Di	I	C				

V	DestReg	"OLD" DestReg	PC	Comp?
1	p9	p1	110	1
1	p10	p6	114	1
0				
0				

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
1	1	1	1	1	1	1	1	1	1

use	OP	Dest Reg	Src Reg 1	Src 1 Ready	Src Reg 2	Src 2 Ready	FU	ROB#
0								
0								
0								
0								

RAT	
A-reg	PMap
x1	9(1)
x2	p2
x3	10(6)
x4	7(4)
x5	8(5)

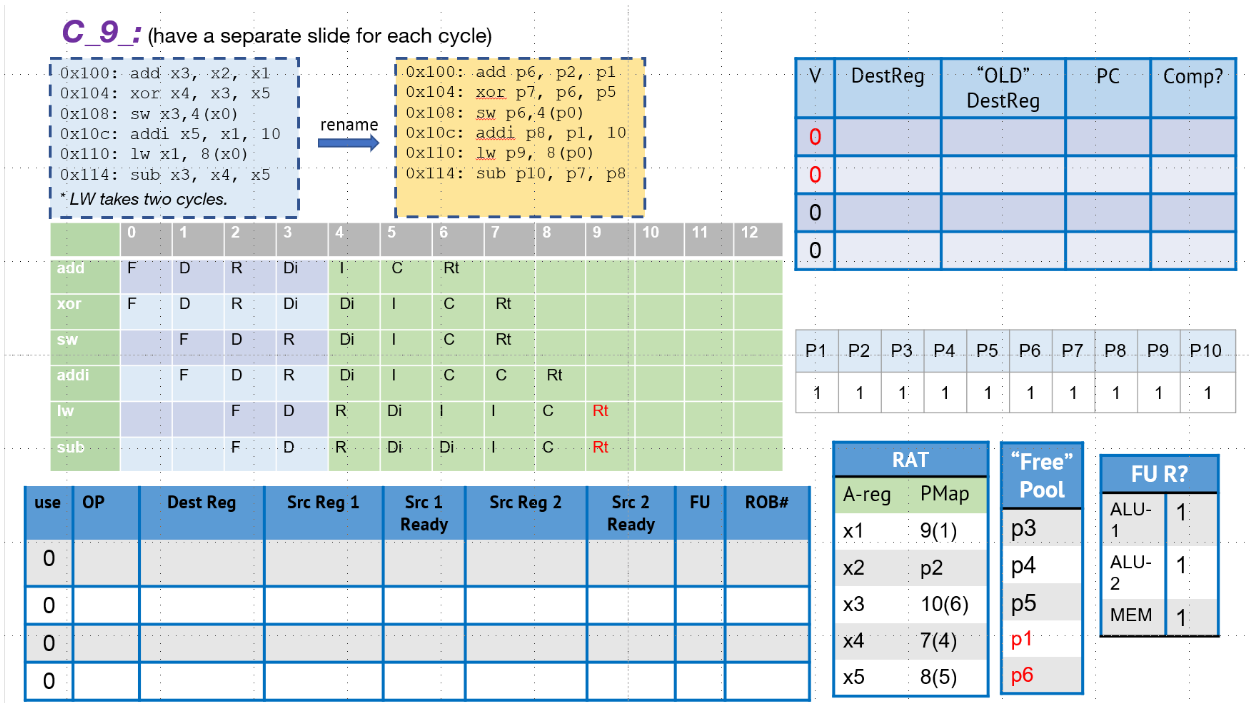
"Free" Pool
p3
p4
p5

FU R?	
ALU-1	1
ALU-2	1
MEM	1

Cycle 9:

“lw”: we retire “lw”. We remove “lw” from the ROB and release p1.

“sub”: we retire “sub”. We remove “sub” from the ROB and release p6.



C_9_: (have a separate slide for each cycle)

```
0x100: add x3, x2, x1
0x104: xor x4, x3, x5
0x108: sw x3,4(x0)
0x10c: addi x5, x1, 10
0x110: lw x1, 8(x0)
0x114: sub x3, x4, x5
```

* *LW takes two cycles.*

rename

```
0x100: add p6, p2, p1
0x104: xor p7, p6, p5
0x108: sw p6,4(p0)
0x10c: addi p8, p1, 10
0x110: lw p9, 8(p0)
0x114: sub p10, p7, p8
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
add	F	D	R	Di	I	C	Rt						
xor	F	D	R	Di	Di	I	C	Rt					
sw		F	D	R	Di	I	C	Rt					
addi		F	D	R	Di	I	C	C	Rt				
lw			F	D	R	Di	I	I	C	Rt			
sub			F	D	R	Di	Di	I	C	Rt			

V	DestReg	“OLD” DestReg	PC	Comp?
0				
0				
0				
0				

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
1	1	1	1	1	1	1	1	1	1

use	OP	Dest Reg	Src Reg 1	Src 1 Ready	Src Reg 2	Src 2 Ready	FU	ROB#
0								
0								
0								
0								

RAT	
A-reg	PMap
x1	9(1)
x2	p2
x3	10(6)
x4	7(4)
x5	8(5)

“Free” Pool
p3
p4
p5
p1
p6

FU R?	
ALU-1	1
ALU-2	1
MEM	1

Q2. Assume that we have two caches, a fully associative (FA), and a 4-way set-associative. Assume that the addresses are 12 bits. Assume that the FA cache has 8 lines/sets, thus the 4-way set-associative cache should have 2 sets. Answer the following questions:

NOTE: The addresses are CPU addresses (so they need to be shifted).

a. Assuming that each cache line is 16 bytes, what is the block offset, index, and tag size for each cache?

   SOLUTION:

   FA cache:

   Block Offset: 4 bits (lowest 4 bits represent entry into one of the 16 bytes within a block)
   Index Size: 0 (FA means no index, as we are fully associative)
   Tag Size: 8 bits (12 total bits, minus 4 bits for block offset)

   4-way set-associative cache:

   Block Offset: 4 bits (same as before)
   Index Size: 1 bit (need 1 bit for 2 sets)
   Tag Size: 7 bits (12 total bits, minus 4 bits for block offset, minus 1 bit for index size)

b. Complete the following table for the FA cache. Assume that the cache is filled already except for two lines. ('inv'=invalid and each cell shows the tag). Use (perfect) LRU replacement policy where (7) means the most recently used.

SOLUTION:

LRU means "Least Recently Used". We keep track of which entries have been used least recently, with "0" meaning the least recently used and "7" being the most recently used. An item is "used" on a cache miss when a new block is brought up to the cache, or when there's a cache hit on a given block. We then update the LRU order to reflect the usage.

In Fully Associative, there is no index, so entries are placed into the cache until all blocks are full. Then, we use LRU to choose which block to evict.

To convert hex address 121 into a cache address:

121 in binary: 000100100001

Offset: 0001 = 1 in decimal. This is the 1st position byte within this block. We shift these offset bits out as the granularity of our CPU address is higher than our cache address (multiple bytes belong to the same block)

Tag: 00010010 = 12 in hex. This is our tag. We use the tag to identify this portion of memory in the cache.

Tag 12 is in the cache, so it is a hit. C4 is now the most recently used block, so we set its LRU value to 7, then shift the previous LRU positions 5-7 down by 1, as C4 was previously at LRU position 4).

Hex address 141 has Tag = 14. 14 is not in the cache, so we need to evict a block. C6 is the least recently used, so we evict C6 and place the new block that address 141 brings up into the cache into C6. We update the LRU (C6 is now at LRU position 7 as the most recently used, and the other blocks are shifted down by 1).

Another example of a hit is hex address 111, which has Tag = 11. 11 is in the cache (C0). C0 is now the most recently used block, so we set its LRU value to 7, then shift the previous LRU positions 4-7 down by 1, as C0 was previously at LRU position 3).

The remainder of the addresses not discussed are processed in the same way. Convert the hex address into a cache address by shifting out the lowest 4 bits of the CPU address. The rest of the address is the Tag. If the Tag is in the table, it is a cache hit and we update the LRU to reflect this change. If the Tag is not in the table, it is a cache miss, so we evict the cache block that is the least recently used (labeled as "0"), and place the new Tag in the table, and finally updating LRU to reflect this change

In the end, we have 5 hits.

FA LRU	Addresses (coming from the CPU) and tags are shown in HEX. Compute the tag for each Cell/Line (C0, C1, ...)								
Address	C0	C1	C2	C3	C4	C5	C6	C7	Hit/Miss
110	11 (6)	inv	15 (2)	28 (3)	12 (4)	18 (5)	22 (0)	79 (1)	M
136	11 (6)	13 (7)	15 (2)	28 (3)	12 (4)	18 (5)	22 (0)	79 (1)	M
121	11 (5)	13 (6)	15 (2)	28 (3)	12 (7)	18 (4)	22 (0)	79 (1)	H
141	11 (4)	13 (5)	15 (1)	28 (2)	12 (6)	18 (3)	14 (7)	79 (0)	M
220	11 (3)	13 (4)	15 (0)	28 (1)	12 (5)	18 (2)	14 (6)	22 (7)	M
111	11 (7)	13 (3)	15 (0)	28 (1)	12 (4)	18 (2)	14 (5)	22 (6)	H
C20	11 (6)	13 (2)	C2 (7)	28 (0)	12 (3)	18 (1)	14 (4)	22 (5)	M
142	11 (5)	13 (2)	C2 (6)	28 (0)	12 (3)	18 (1)	14 (7)	22 (4)	H
158	11 (4)	13 (1)	C2 (5)	15 (7)	12 (2)	18 (0)	14 (6)	22 (3)	M
15C	11 (4)	13 (1)	C2 (5)	15 (7)	12 (2)	18 (0)	14 (6)	22 (3)	H
1A0	11 (3)	13 (0)	C2 (4)	15 (6)	12 (1)	1A (7)	14 (5)	22 (2)	M
189	11 (2)	18 (7)	C2 (3)	15 (5)	12 (0)	1A (6)	14 (4)	22 (1)	M
301	11 (1)	18 (6)	C2 (2)	15 (4)	30 (7)	1A (5)	14 (3)	22 (0)	M
110	11 (7)	18 (5)	C2 (1)	15 (3)	30 (6)	1A (4)	14 (2)	22 (0)	H
135	11 (6)	18 (4)	C2 (0)	15 (2)	30 (5)	1A (3)	14 (1)	13 (7)	M

c. Assume that the SA cache uses a **pseudo-LRU** replacement policy. Complete the table for this structure. (if all flag bits are 1, you should reset them). For eviction in each set, start from the left (i.e., W0) and find the first way with flag=0.

SOLUTION:

In pseudo LRU, we are approximating LRU. We store 1 bit (flag) per row. At the beginning, every row's flag = 0. On a hit, we set the flag = 1. On a miss, we place the new data into the first set with flag = 0 or evict the first set with flag = 0. In either case, we place the new tag into the cache location and set flag = 1. When all the flags = 1, we reset the flags to all equal 0, since there is no candidate to evict.

In a set associative cache, we need to apply pseudo LRU independently to each set.

To convert hex address 121 into a cache address:

121 in binary: 000100100001

Offset: 0001 = 1 in decimal. This is the 1st position byte within this block. We shift these offset bits out as the granularity of our CPU address is higher than our cache address (multiple bytes belong to the same block)

Index: 0 = 0 in decimal. This means we index into Set 0.

Tag: 0001001 = 9 in hex. This is our tag. We use the tag to identify this portion of memory in the cache.

Tag = 9 is in Set 0. This is a hit. We set the flag in W1 = 1.

Hex address 141: offset = 0001. Index = 0. Tag = A. Tag = A is not present in Set 0, so we need to evict. Since W0 has the first flag = 0, we evict W0.

Hex address 220: offset = 0000. Index = 0. Tag = 11. Tag = 11 is present in Set 0, so this is a hit. We set the flag in W3 = 1.

The first instance of a need to clear the flags is for Hex address C20: offset = 0000. Index = 0. Tag = 61. When we place Tag = 61 in the cache, evicting the block at W2 in Set 0, all flags in Set 0 = 1, so we reset all the flags in Set 0 to 0.

The second instance of a need to clear the flags is for Hex address = 301: offset = 0001. Index = 0. Tag = 18. When we place Tag = 61 in the cache, evicting the block at W3 in Set 0, all flags in Set 0 = 1, so we reset all the flags in Set 0 to 0.

The remainder of the addresses not discussed are processed in the same way. Convert the hex address into a cache address by shifting out the lowest 4 bits of the CPU address. The next bit is the index (referring to Set 0 or Set 1). The rest of the address is the Tag. If the given Tag is within the assigned set, it is a cache hit and we set the flag = 1. If the Tag is not in the table, it is a cache miss, so we evict the cache block that has the first instance of flag = 0, and place the new Tag in the table (updating flag = 1 in the process). If at any moment we place new data into a way within a set, and all the flags = 1, we immediately set all the flags within that set to zero (this is done via wired logic).

In the end, we have 8 hits.

SA **PLRU**	Addresses and tags are shown in HEX. Compute the tag for each Way (W0, W1, ...).								
	Set 0				Set 1				
Address	W0	W1	W2	W3	W0	W1	W2	W3	Hit/Miss
110	14 (0)	9 (0)	C (0)	11 (0)	3C (0)	8 (1)	A (0)	inv	M
136	14 (0)	9 (0)	C (0)	11 (0)	3C (0)	8 (1)	A (0)	9 (1)	M
121	14 (0)	9 (1)	C (0)	11 (0)	3C (0)	8 (1)	A (0)	9 (1)	H

141	A (1)	9 (1)	C (0)	11 (0)	3C (0)	8 (1)	A (0)	9 (1)	M
220	A (1)	9 (1)	C (0)	11 (1)	3C (0)	8 (1)	A (0)	9 (1)	H
111	A (1)	9 (1)	C (0)	11 (1)	3C (0)	8 (1)	A (0)	9 (1)	H
C20	A (0)	9 (0)	61 (0)	11 (0)	3C (0)	8 (1)	A (0)	9 (1)	M
142	A (1)	9 (0)	61 (0)	11 (0)	3C (0)	8 (1)	A (0)	9 (1)	H
158	A (1)	9 (0)	61 (0)	11 (0)	3C (0)	8 (1)	A (1)	9 (1)	H
15C	A (1)	9 (0)	61 (0)	11 (0)	3C (0)	8 (1)	A (1)	9 (1)	H
1A0	A (1)	D (1)	61 (0)	11 (0)	3C (0)	8 (1)	A (1)	9 (1)	M
189	A (1)	D (1)	C (1)	11 (0)	3C (0)	8 (1)	A (1)	9 (1)	M
301	A (0)	D (0)	C (0)	18 (0)	3C (0)	8 (1)	A (1)	9 (1)	M
110	A (0)	D (0)	C (0)	18 (0)	3C (0)	8 (1)	A (1)	9 (1)	H
135	A (0)	D (0)	C (0)	18 (0)	3C (0)	8 (1)	A (1)	9 (1)	H

d. What is the miss rate for each case (FA with LRU and SA with PLRU)?

SOLUTION:

Miss Rate Formula: $\frac{\textit{Number of Cache Misses}}{\textit{Number of Accesses}}$ **x 100**

FA with LRU miss rate: $\frac{10}{15}$ **x 100 = 66.66%**

4-Way SA with Pseudo LRU miss rate: $\frac{7}{15}$ **x 100 = 46.66%**

Q3. Provide a short answer for each of the following questions:

A. Do hierarchical page tables improve performance, area overhead, or both? Explain why?
B. How does TLB improve performance?
C. Why "address translation" is needed in a modern processor?

SOLUTION:

a)
Hierarchical page tables improve area overhead, but not performance. Hierarchical page tables leverage the fact that virtual addresses are sparse. Thus, we are able to compress the page table storage.

Performance decreases (without the addition of TLB). Instead of a singular page table with 1 access to the page table and 1 access to the physical memory, we need multiple accesses within the hierarchical page table and 1 access to the physical memory.

b)
The Translation Lookaside Buffer (TLB) improves performance because we are able to cache recent virtual address to physical address translations, increasing performance because we don't need to search through the hierarchical page table.

If the address exists in the buffer (meaning it has been recently used), we can quickly get the physical address. If the address is not in the buffer, we need to go through the hierarchical page table.

Hit Time of Cache accesses = (Hit time of TLB) + (Time of L1)

Miss Time of Cache accesses = (Miss time of TLB) + (Time of L1)
 -Note: where Miss time of TLB equals the time to access the hierarchical page table.

c)
In modern processors, we have many programs that benefit from shared code, shared libraries, and dynamic memory partitioning. When our memory layout is static, it is difficult to add new programs and adjust the memory assigned to

existing programs. A solution to this problem is to make memory partitioning dynamic. We divide the memory into smaller, fixed-size blocks (pages), where we assign pages to each program when it is needed. Each program has a page table that references the location of the given pages in memory. The introduction of page tables means the physical address does not need to used for accesses anymore, thus this results in the use of virtual addresses, where address translation occurs from a virtual address to the physical address. The virtual address space is unique to the individual program and looks large and unified to the individual program.