Topic V0A

Using `la` and the `IsBra` Programming Example

Readings: (Section 2.6-2.7)

Load Address

The `la` pseudo instruction

```
.data
flag:
        .word 1
.text
setFlag:
   la   t0 flag        # t0 <- address of flag
   addi t1, zero, 1    # t1 <- 1
   sw   t1, 0(t0)      # Mem[flag] <- 1
   ret
```

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x0fc10297	auipc x5,0x0000fc10	6: la t0 flag # t0 <- address of fl
	0x00400004	0x00028293	addi x5,x5,0x00000000	
	0x00400008	0x00100313	addi x6,x0,0x00000001	7: addi t1, zero, 1 # t1 <- 1
	0x0040000c	0x0062a023	sw x6,0x00000000(x5)	8: sw t1, 0(t0) # Mem[flag] <- 1
	0x00400010	0x00008067	jalr x0,x1,0x00000000	9: ret

Data Segment

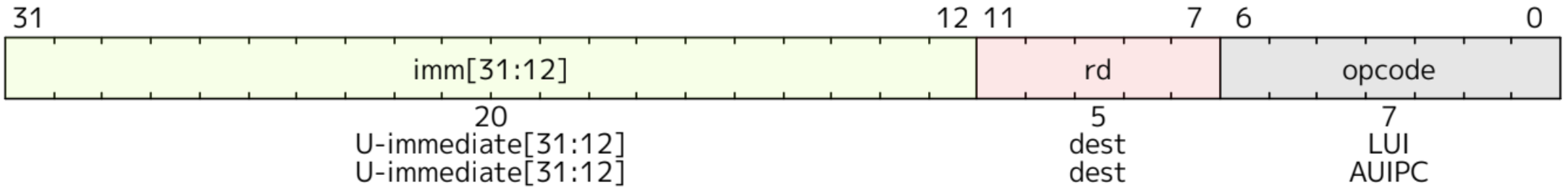
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10010000	0x00000001	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000

The RISC-V Instruction Set Manual Volume I

Unprivileged Architecture

Version 20240411

https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view



LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the 32-bit U-immediate value into the destination register *rd*, filling in the lowest 12 bits with zeros.

AUIPC (add upper immediate to `pc`) is used to build `pc`-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the address of the AUIPC instruction, then places the result in register *rd*.

PC: 0x00400000

\+ 0x0fc10000

0x10010000

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x0fc10297	auipc x5,0x0000fc10	6: la t0 flag # t0 <- address of fl
	0x00400004	0x00028293	addi x5,x5,0x00000000	
	0x00400008	0x00100313	addi x6,x0,0x00000001	7: addi t1, zero, 1 # t1 <- 1
	0x0040000c	0x0062a023	sw x6,0x00000000(x5)	8: sw t1, 0(t0) # Mem[flag] <- 1
	0x00400010	0x00008067	jalr x0,x1,0x00000000	9: ret

Data Segment

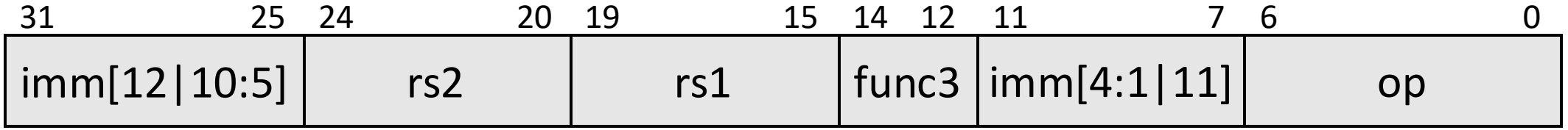
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10010000	0x00000001	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000

Sample Question

One way to implement a “branch always” (`bra`) instruction, is to use the `beq` instruction and to make both registers the same. For example:

```
beq  t0, t0, LABEL
```

Binary format of a branch instruction:

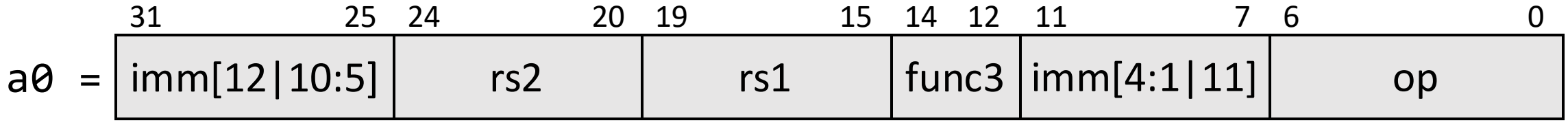


Write the RISC-V code for a function IsBra that receives the binary code of a `beq` instruction in `a0` and returns 1 in `a0` if it is a `bra` and returns `0` in `a0` otherwise.

Sample Question

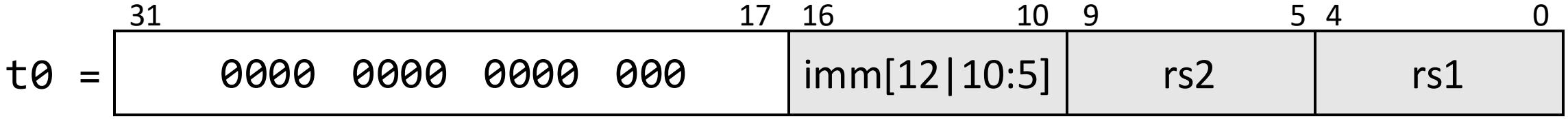
Write the RISC-V code for a function `IsBra:`

- Input parameter:
  - `a0`: binary code of a `beq` instruction;
- Return value:
  - `a0` = 1: the `beq` instruction is a `bra`
  - `a0` = 0: the `beq` instruction is not a `bra`



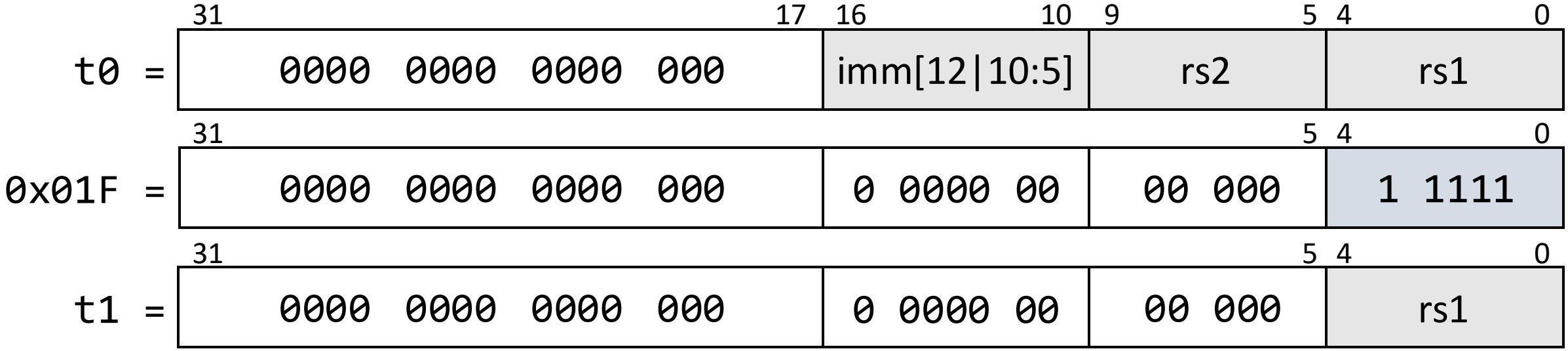
First we shift the content of `a0` to the right by 15 bits:

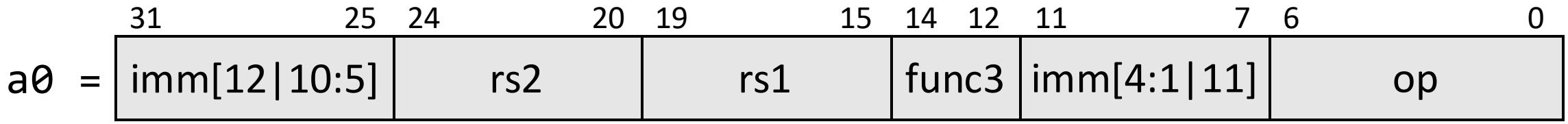
```
srli  t0, a0, 15
```



Now we AND the result with `0x01F`:

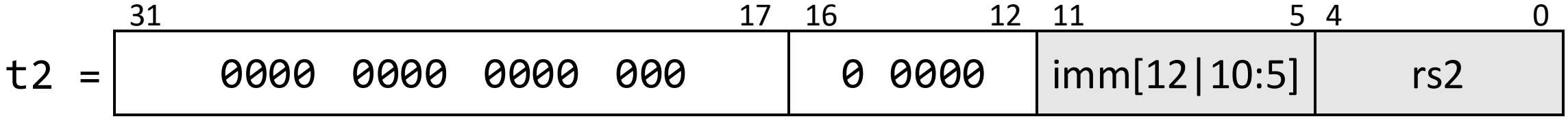
```
andi  t1, t0, 0x01F
```





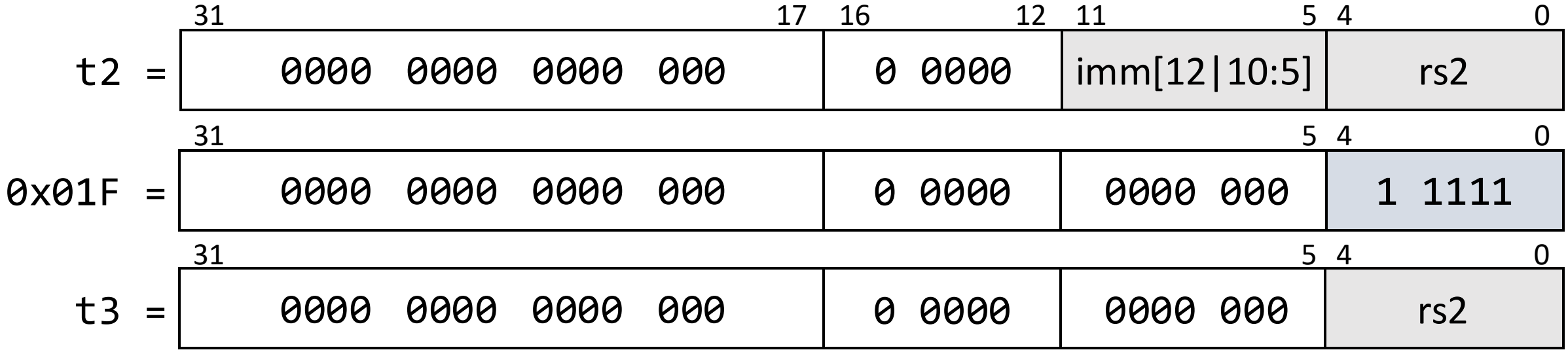
Then we shift the content of `a0` to the right by 20 bits:

```
srli  t0, a0, 20
```



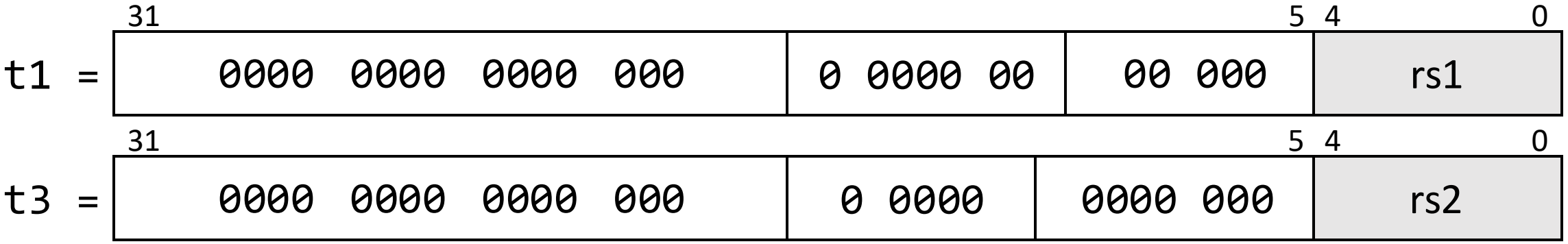
Now we AND the result with `0x01F`:

```
andi  t3, t2, 0x01F
```



Finally, we use a `beq` instruction to find if `rs1` and `rs2` are the same register:

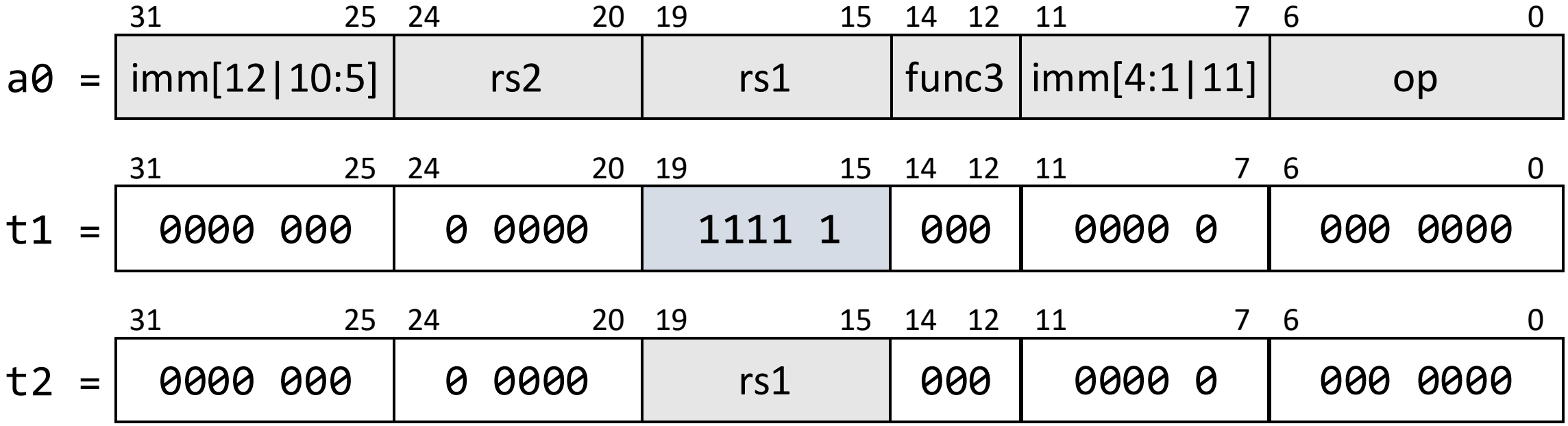
```
beq   t1, t3, braTrue
```




```
IsBra:
        srli   t0, a0, 15          # t0 ← a0 >> 15
        andi   t1, t0, 0x01F       # t1 ← rs1
        srli   t2, a0, 20          # t2 ← a0 >> 20
        andi   t3, t2, 0x01F       # t2 ← rs2
        beq    t1, t3, braTrue     # if rs1 = rs2 goto braTrue
        addi   a0, zero, 0         # a0 ← 0
        jalr   zero, ra, 0         # return
braTrue:
        addi   a0, zero, 1         # a0 ← 1
        jalr   zero, ra, 0         #return
```

Textbook uses the syntax:

`jalr zero, 0(ra)`



Alternatively, we can AND `a0` with `0x000F8000` but you cannot use a constant larger than 12 bits in an `andi` instruction:

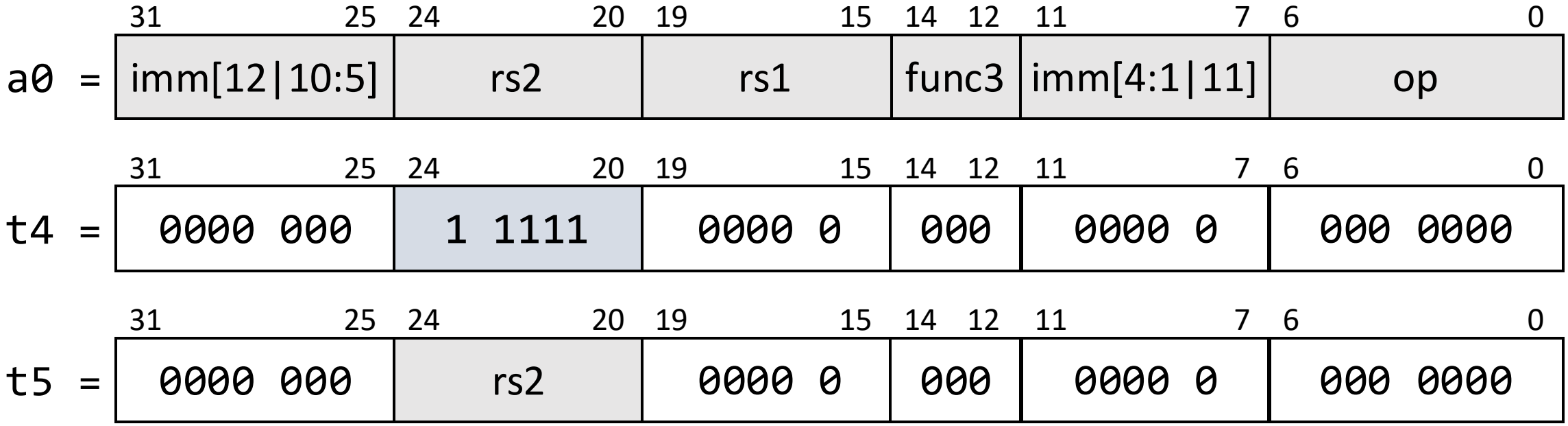
```
li   t0, 0x0F8        # t0 ← 0x0000 00F8
slli t1, t0, 12       # t1 ← 0x000F 8000
```

OR

```
lui  t1, 0x000F8      # t0 ← 0x000F 8000
```

Now we can AND `a0` with `a1` to obtain:

```
and  t2, t1, a0
```



Now we can do something similar to extract the rs2 field:

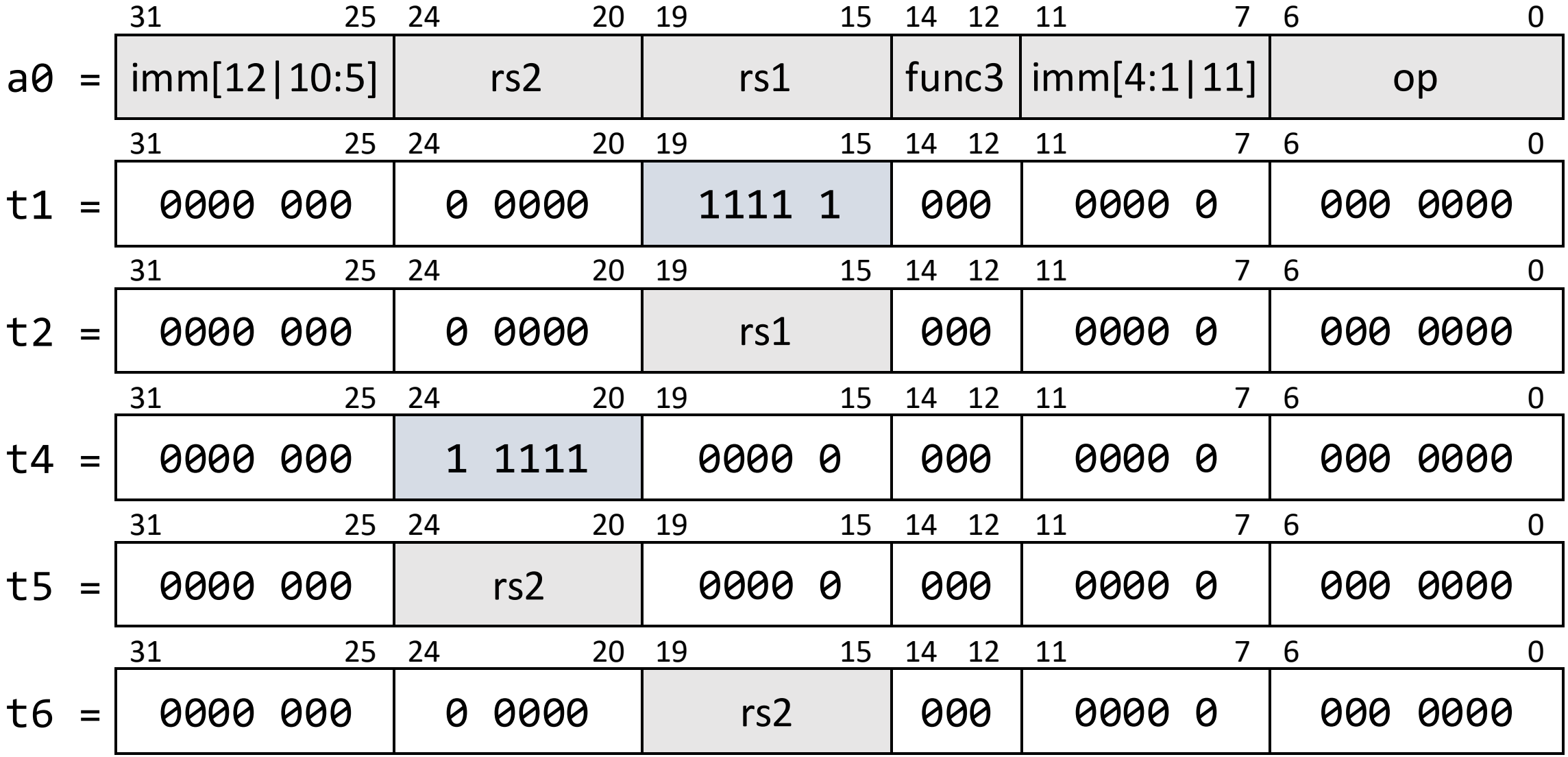
```
li   t3, 0x01F          # t0 ← 0x0000 001F
slli t4, t0, 20         # t1 ← 0x01F0 0000
```

OR

```
lui  t4, 0x01F00        # t0 ← 0x01F0 0000
```

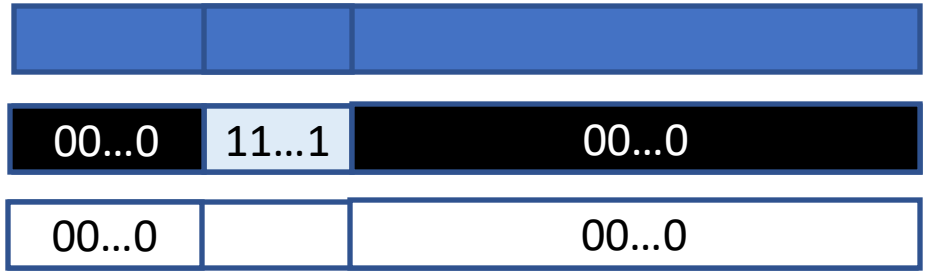
Now we can AND a0 with a1 to obtain:

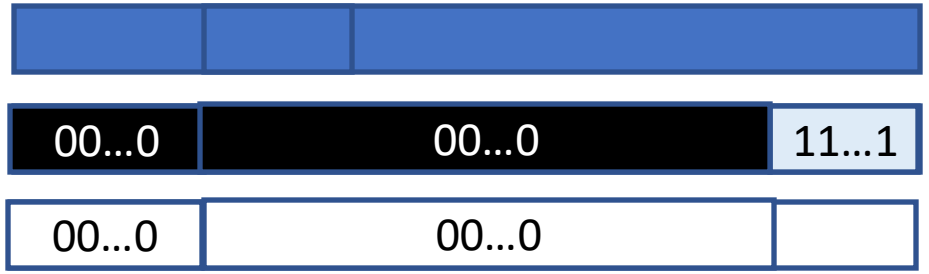
```
and  t5, t4, a0
```



Finally we can shift `t5` to the right by 5 before comparing with `t2`:

```
srli  t6, t5, 5        # t6 ← t5 >> 5
beq   t6, t2, braTrue
…
```





Recap