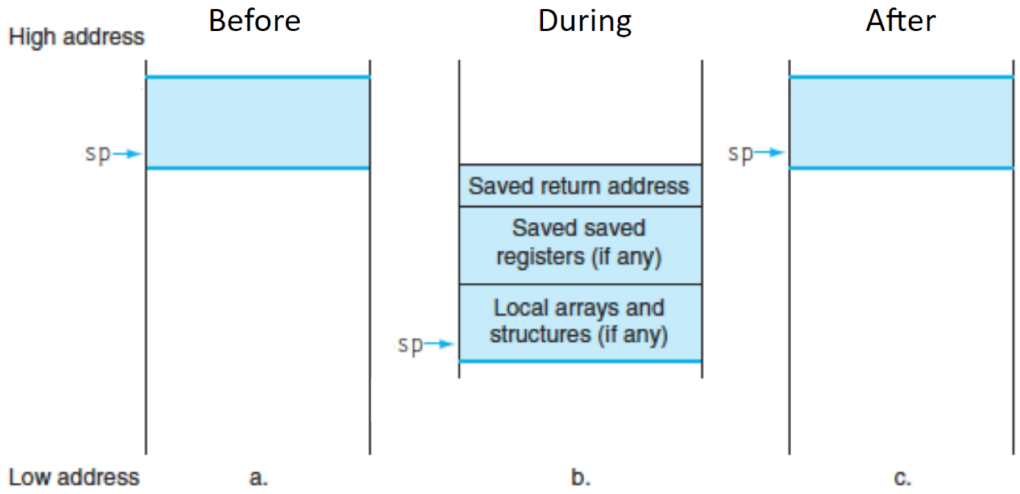
RISC-V Instruction Formats

```
0000000020400094 reset_handler:
; pub unsafe fn reset_handler() {
20400094:   79 71   addi    sp, sp, -48
; rv32i::init_memory();
20400096:   06 d6   sw  ra, 44(sp)
20400098:   26 d4   sw  s1, 40(sp)
2040009a:   4a d2   sw  s2, 36(sp)
2040009c:   4e d0   sw  s3, 32(sp)
2040009e:   52 ce   sw  s4, 28(sp)
204000a0:   56 cc   sw  s5, 24(sp)
204000a2:   97 00 00 00     auipc   ra, 0
204000a6:   e7 80 40 60     jalr    ra, ra, 1540
; asm!("
204000aa:   37 05 40 20     lui a0, 132096
204000ae:   13 05 05 04     addi    a0, a0, 64
204000b2:   73 10 55 30     csrw    mtvec, a0
; unsafe { &*self.ptr }
204000b6:   37 35 40 20     lui a0, 132099
204000ba:   03 25 c5 3d     lw  a0, 988(a0)
204000be:   b7 f5 51 00     lui a1, 1311
204000c2:   93 85 e5 15     addi    a1, a1, 350
204000c6:   4c cd   sw  a1, 28(a0)
204000c8:   23 20 05 00     sw  zero, 0(a0)
204000cc:   4c cd   sw  a1, 28(a0)
204000ce:   b7 f5 09 0d     lui a1, 53407
204000d2:   b5 05   addi    a1, a1, 13
204000d4:   0c cd   sw  a1, 24(a0)
```

Actual RISC-V code generated for the Tock operating system

Review of Last Lecture

- Pseudo-instructions
- Functions in assembly
  - Six steps to calling a function
- Calling conventions
  - Caller and callee saved registers



Great Idea #1: Levels of Representation/Interpretation

Higher-Level Language Program (e.g. C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g. RISCV)

```
lw    t0, 0(x2)
lw    t1, 4(x2)
sw    t1, 0(x2)
sw    t0, 4(x2)
```

We were here

Assembler

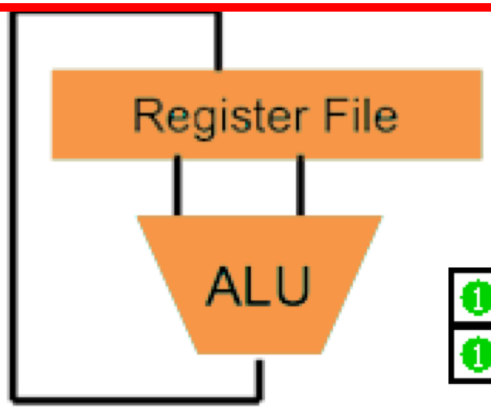
Machine Language Program (RISCV)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

We are here

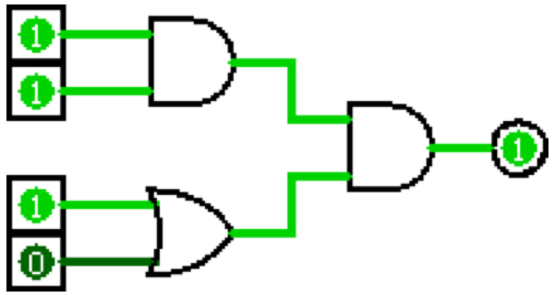
Machine Interpretation

Hardware Architecture Description (e.g. block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



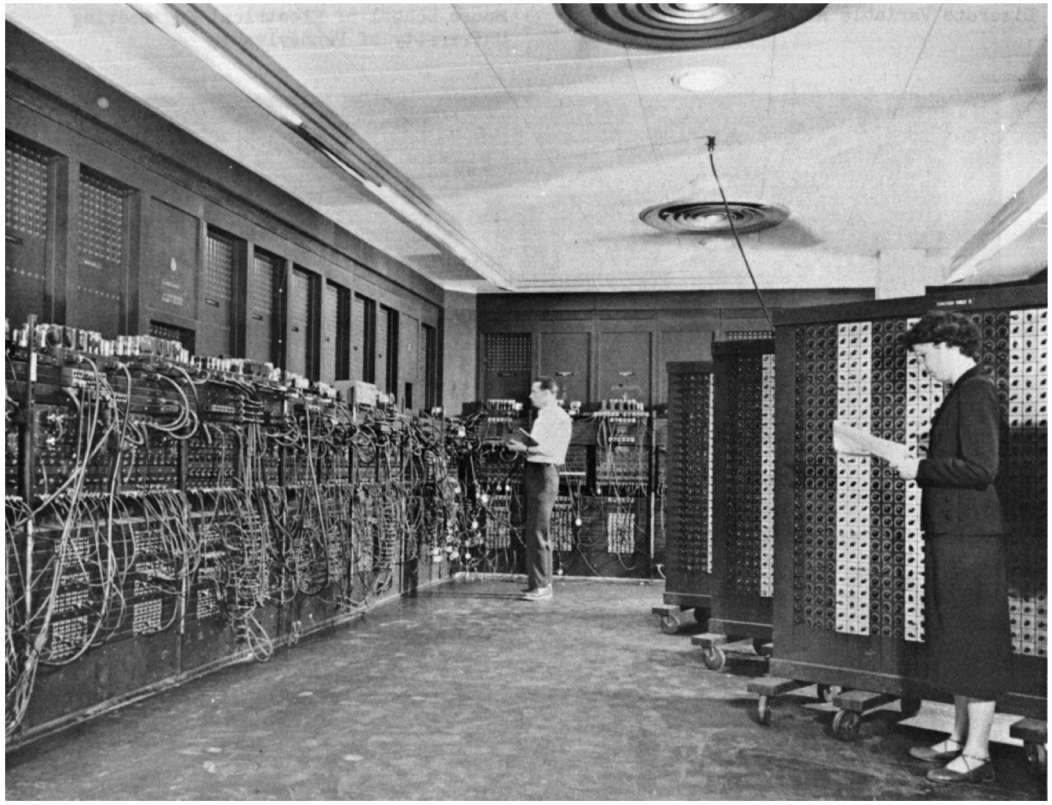
Agenda

- Stored-Program Concept
- R-Format
- I-Format
- S-Format
- SB-Format
- U-Format
- UJ-Format

First computers were hard to reprogram

ENIAC (University of Pennsylvania - 1946)

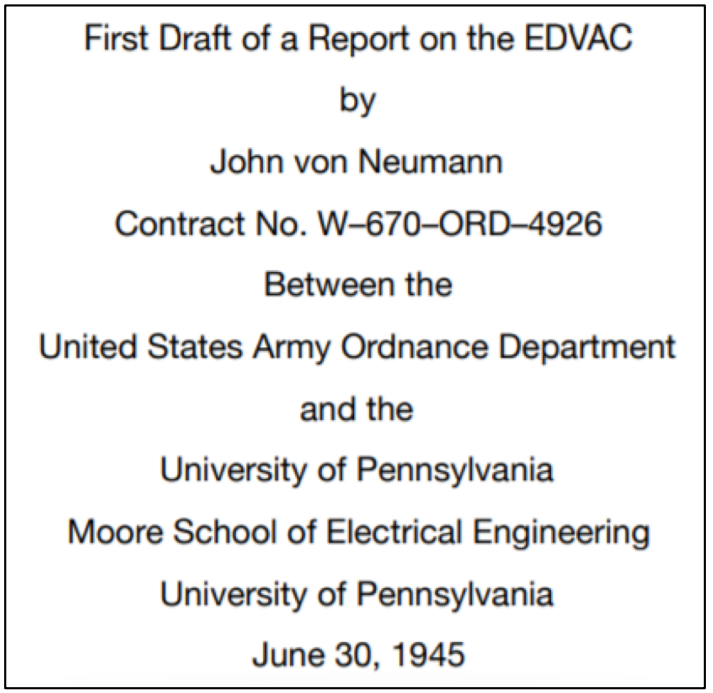
- Blazing fast
  - 10 digit x 10 digit multiply in 2.8 ms
- Programmed with patch cords and switches
  - 2-3 days to set up a new program



Big Idea: Stored-Program Concept

- Instructions can be represented as bit patterns
  - Entire programs stored in memory just like data
  - Reprogramming just takes rewriting memory
    - Rather than rewiring the computer

- Known as “von Neumann” computers after widely distributed tech report on EDVAC project


First Draft of a Report on the EDVAC

by

John von Neumann

Contract No. W–670–ORD–4926

Between the

United States Army Ordnance Department

and the

University of Pennsylvania

Moore School of Electrical Engineering

University of Pennsylvania

June 30, 1945

Everything has a memory address

- Since instructions and data are both in memory, addresses can point to either

- C pointers are just memory addresses that point to data
- The Program Counter (PC) just holds a memory address that points to code

How do you distinguish code and data?

Depends on Interpretation

Number **Instruction**

```
7,537,331 <-- 0x007302B3 --> add x5,x6,x7
```

- programs can be stored in memory as numbers
- whether a number is code or a value is all in how you *interpret* it

- Need a method for interpreting numbers as instructions

Binary compatibility

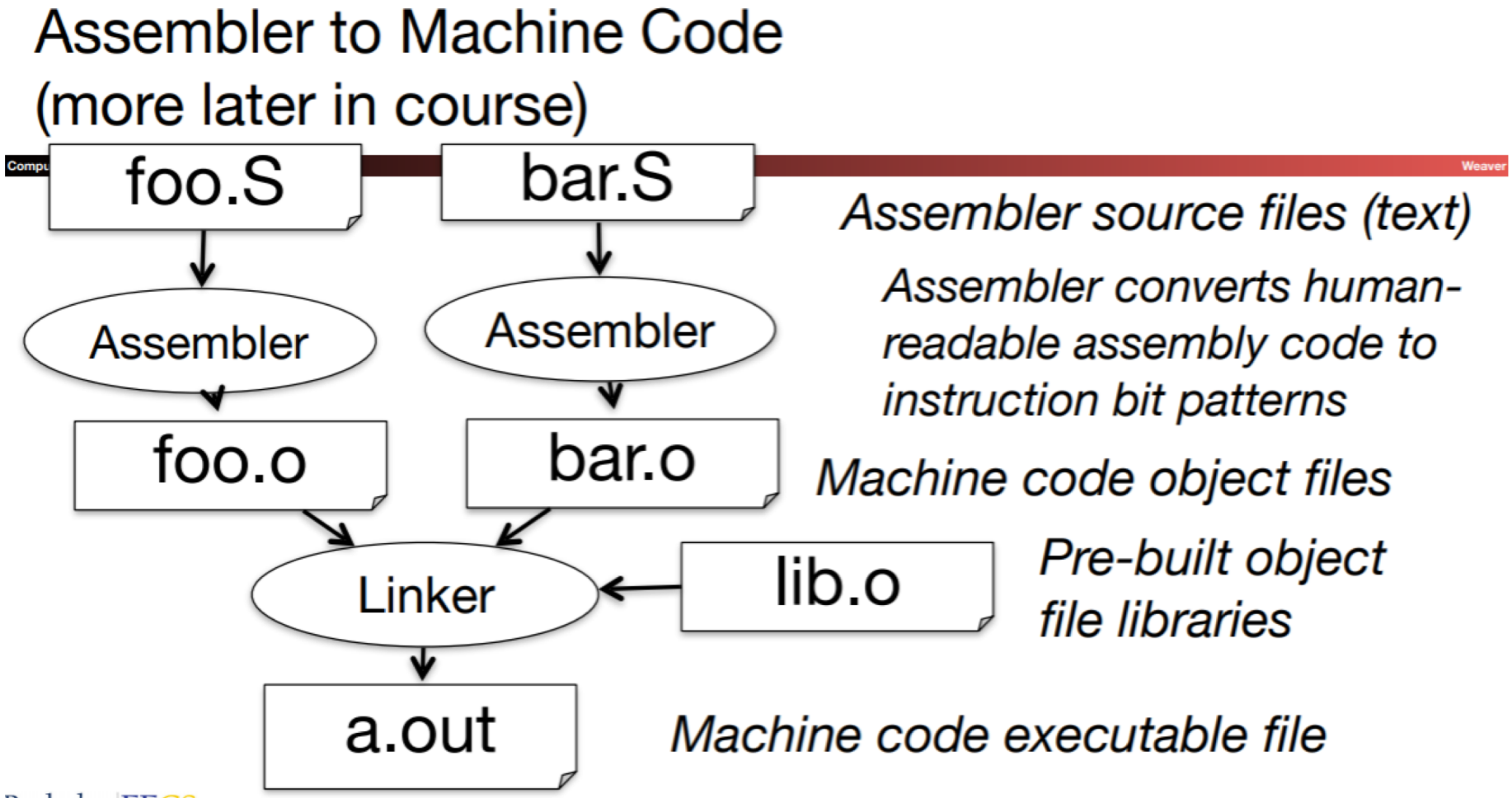
Programs are distributed in binary form

- Bound to a specific instruction set

New machines in the same family want to run old programs (“binaries”) as well as programs using new instructions

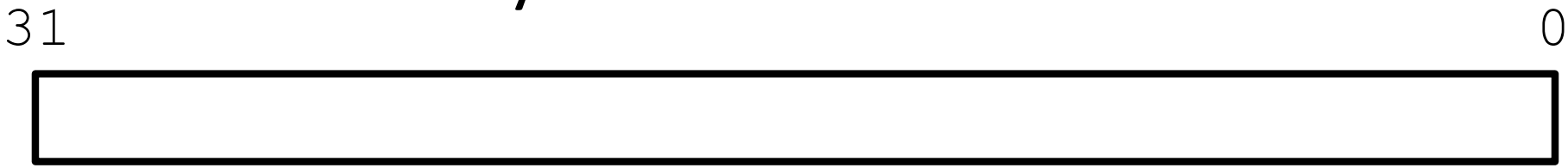
- Leads to “backwards-compatible” instruction set growing over time

Usually, the assembler does this translation



Instructions as Numbers

- By convention, RISC-V instructions are each 1 word = 4 bytes = 32 bits



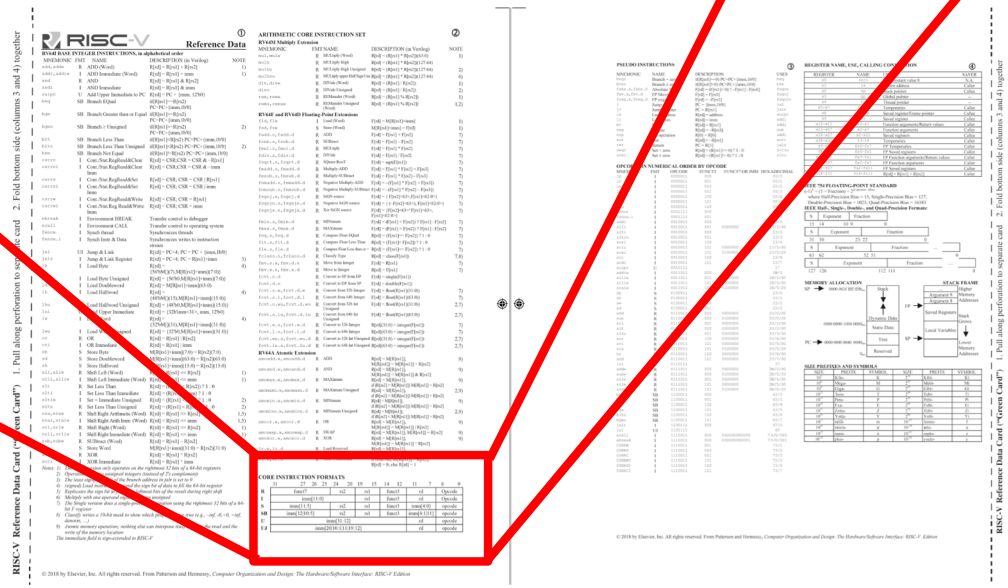
- Divide the 32 bits of an instruction into "fields"
  - regular field sizes → simpler hardware
  - will need some variation….
- Define 6 types of *instruction formats:*
  - R-Format I-Format S-Format
    U-Format SB-Format UJ-Format

The 6 Instruction Formats

- R-Format: instructions using 3 register inputs
  - `add`, `xor`, `mul` —arithmetic/logical ops
- I-Format: instructions with immediates, loads
  - `addi`, `lw`, `jalr`, `slli`
- S-Format: store instructions: `sw`, `sb`
- SB-Format: branch instructions: `beq`, `bge`
- U-Format: instructions with upper immediates
  - `lui`, `auipc` —upper immediate is 20-bits
- UJ-Format: the jump instruction: `jal`

The 6 Instruction Formats

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

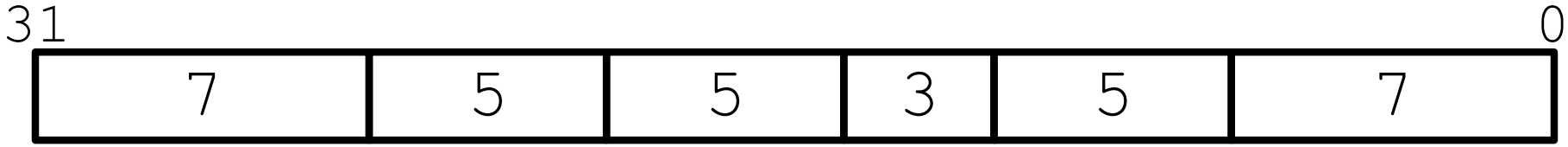


Agenda

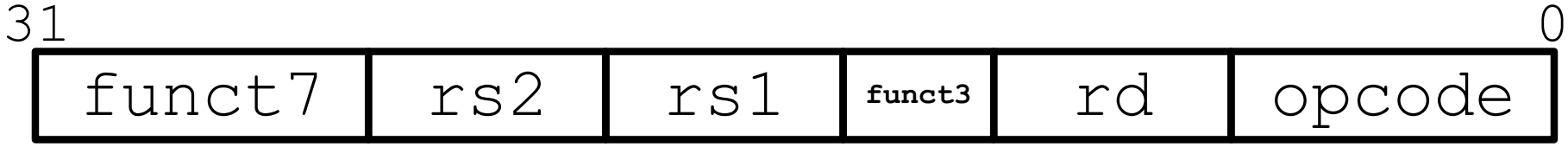
- Stored-Program Concept
- R-Format
- I-Format
- Administrivia
- S-Format
- SB-Format
- U-Format
- UJ-Format

R-Format Instructions (1/3)

- Define "fields" of the following number of bits each: 7 + 5 + 5 + 3 + 5 + 7 = 32

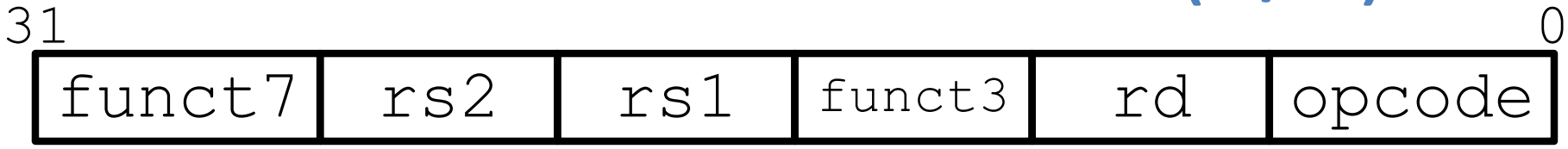


- Each field has a name:



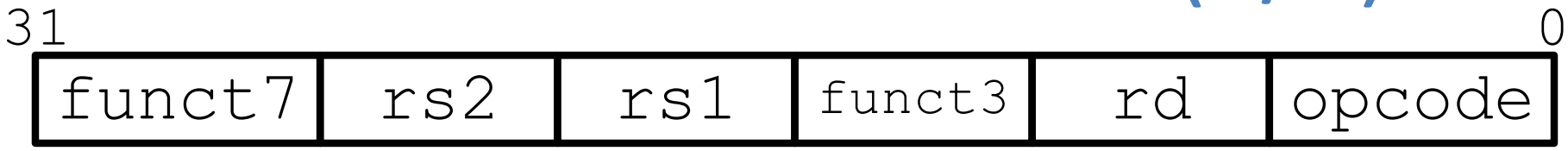
- Each field is viewed as its own unsigned int
  - 5-bit fields can represent any number 0-31, while 7-bit fields can represent any number 0-127, etc.

R-Format Instructions (2/3)



- `opcode` (7): partially specifies operation
  - e.g. **R**-types have opcode = 0b0110011, **SB** (branch) types have opcode = 0b1100011
- `funct7+funct3` (10): combined with `opcode`, these two fields describe what operation to perform
- How many R-format instructions can we encode?
  - with `opcode` fixed at 0b0110011, just funct varies: $(2^7) \times (2^3) = (2^{10}) = 1024$

R-Format Instructions (3/3)



- `rs1` (5): 1st operand ("source register 1")
- `rs2` (5): 2nd operand (second source register)
- `rd` (5): "destination register" — receives the result of computation
- **Recall:** RISCV has 32 registers
  - A 5 bit field can represent exactly $2^5 = 32$ things (interpret as the register numbers **x0-x31**)

Reading from the Green Sheet

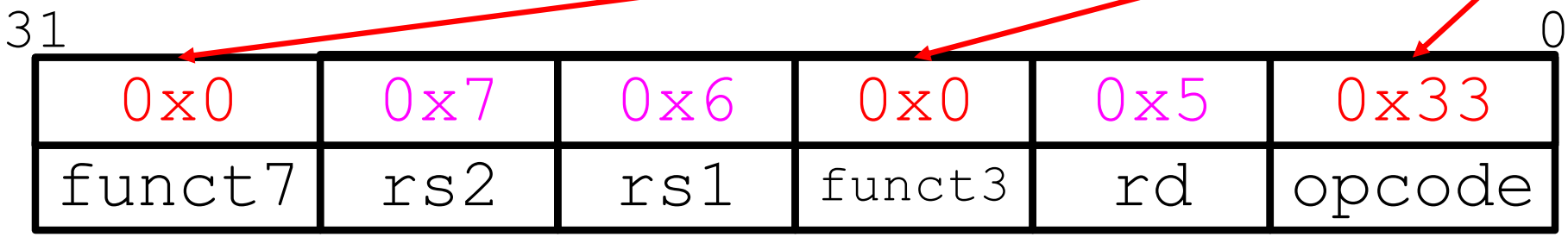
```
add t0 t1 t2
```

RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)
add, addw	R	ADD (Word)	R[rd] = R[rs1] + R[rs2]

OPCODES IN NUMERICAL ORDER BY OPCODE

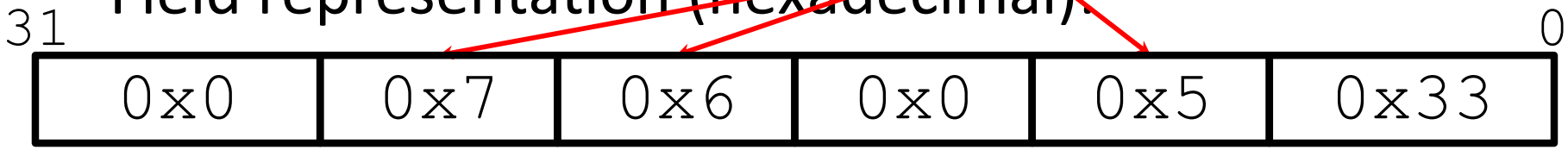
MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
sd	S	0100011	011		23/3
add	R	0110011	000	0000000	33/0/00
sub	R	0110011	000	0100000	33/0/20



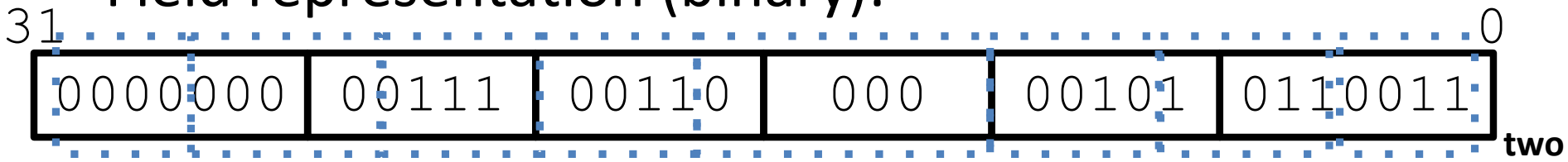
R-Format Example

- RISCV Instruction: `add x5,x6,x7`

Field representation (hexadecimal):



Field representation (binary):



hex representation: `0x 0073 02B3`

decimal representation: 7,537,331

Called a Machine Language Instruction

All RV32 R-format instructions

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Different encoding in funct7 + funct3 selects different operations

Agenda

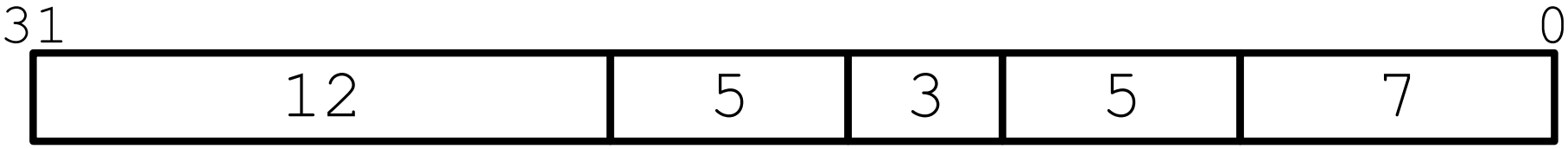
- Stored-Program Concept
- R-Format
- I-Format
- Administrivia
- S-Format
- SB-Format
- U-Format
- UJ-Format

I-Format Instructions (1/4)

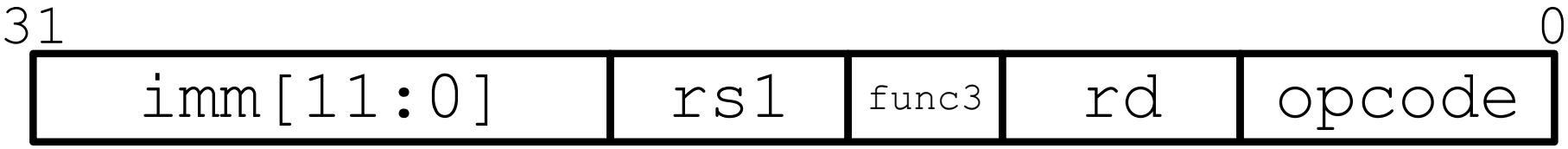
- What about instructions with immediates?
  - 5-bit field too small for most immediates
- Ideally, RISCV would have only one instruction format (for simplicity)
  - Unfortunately here we need to compromise
- Define new instruction format that is *mostly* consistent with R-Format
  - First notice that, if instruction has immediate, then it uses at most 2 registers (1 src, 1 dst)

I-Format Instructions (2/4)

- Define “fields” of the following number of bits each: 12 + 5 + 3 + 5 + 7 = 32 bits

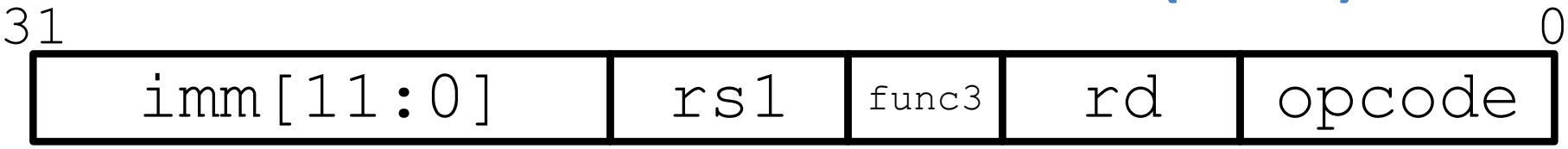


- Field names:



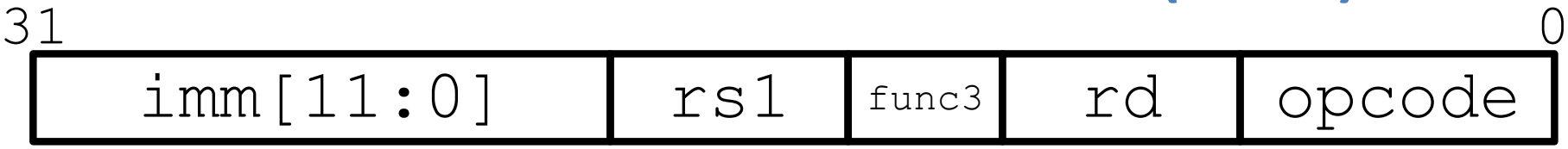
- **Key Concept:** Only **`imm`** field is different from R-format: `rs2` and `funct7` replaced by 12-bit signed immediate, **`imm[11:0]`**

I-Format Instructions (3/4)



- `opcode` (7): uniquely specifies the instruction
- `rs1` (5): specifies a register operand
- `rd` (5): specifies destination register that receives result of computation

I-Format Instructions (4/4)



- `immediate` (12): 12 bit number
  - All computations done in words, so 12-bit immediate must be *extended* to 32 bits
  - always **sign-extended** to 32-bits before use in an arithmetic operation

- Can represent 2^{12} different immediates
  - imm[11:0] can hold values in range $[-2^{11}, +2^{11})$

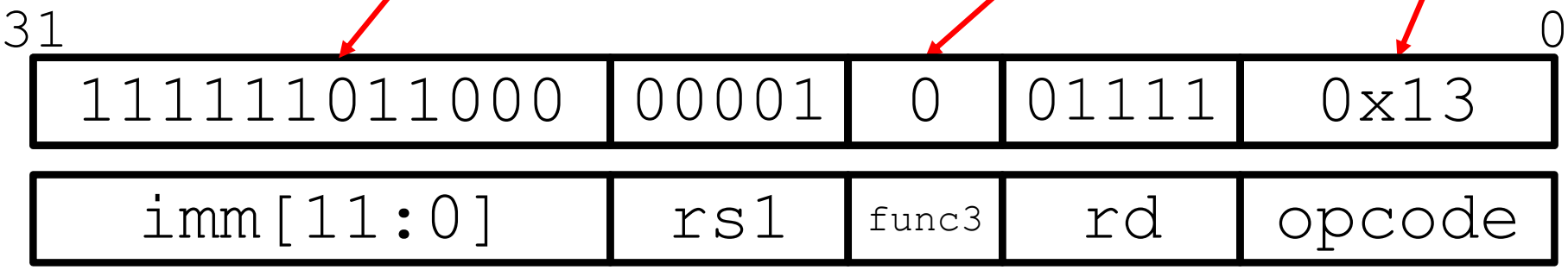
I-Format Example (1/2)

```
addi x15,x1,-40
```

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
fence.i	I	0001111	001		0F/1
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00

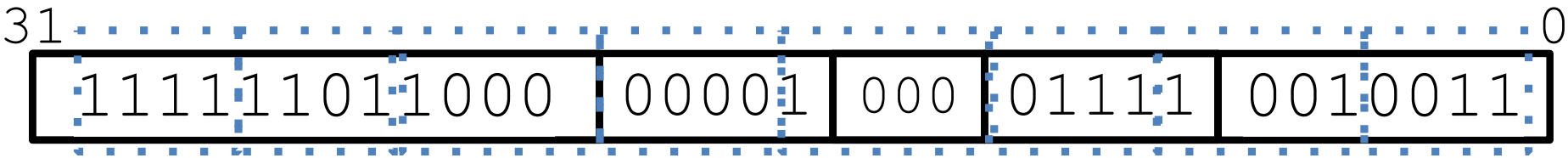
```
rd  = x15
rs1 = x1
```



I-Format Example (2/2)

- RISCV Instruction: `addi x15,x1,-40`

Field representation (binary):



hex representation: 0x FD80 8793

decimal representation: 4,253,058,963

All RISCV I-Type Arithmetic Instructions

imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

One of the higher-order immediate bits is used to distinguish "shift right logical" (SRLI) from "shift right arithmetic" (SRAI)

This is part of the funct7 field

"Shift-by-immediate" instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions)

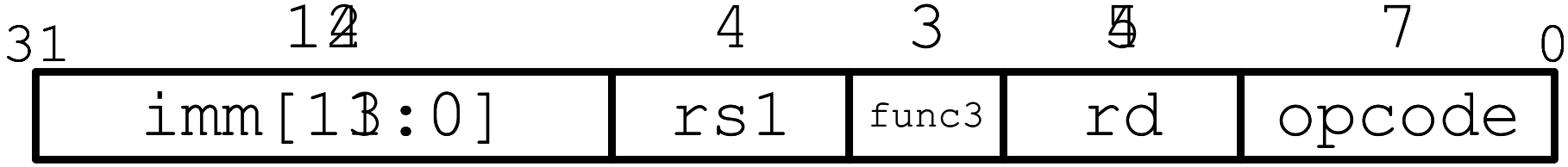
Question: If the number of registers were halved, which statement is true?

(A) There must be more R-type instructions

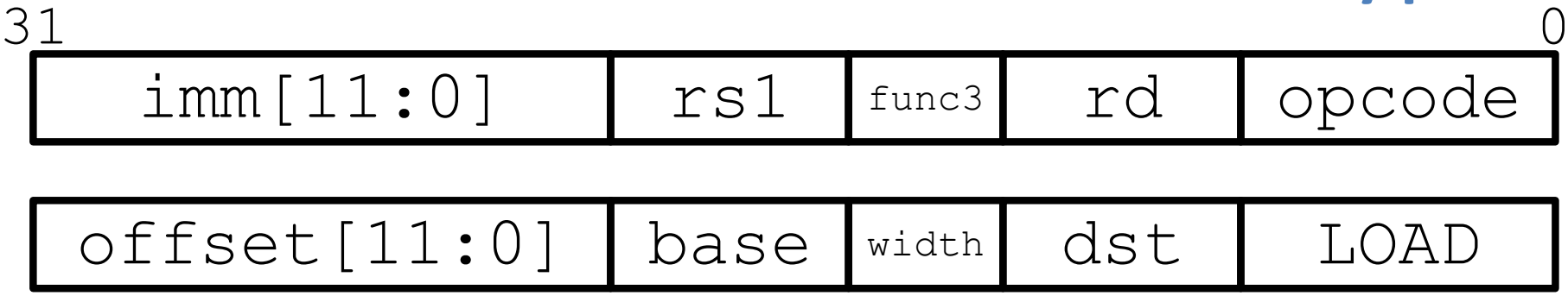
(B) There must be less I-type instructions

(C) Shift amounts would change to 0-63

(D) I-type instructions could have 2 more immediate bits



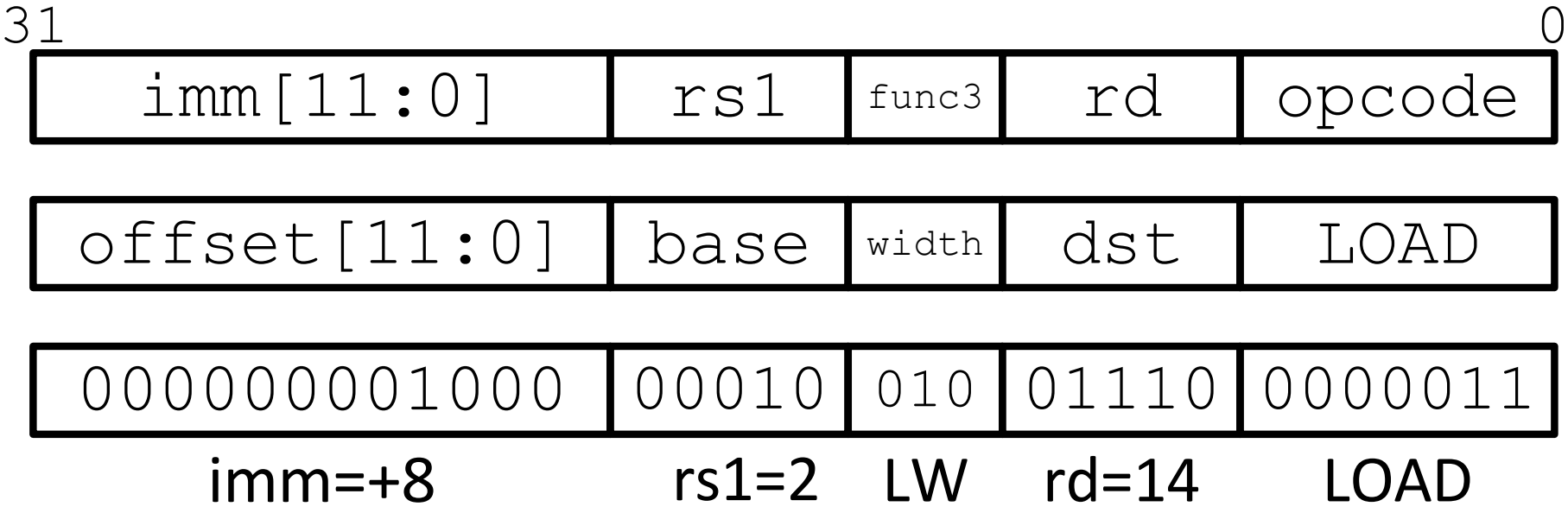
Load Instructions are also I-Type



- The 12-bit signed immediate is added to the base address in register `rs1` to form the memory address
  - This is very similar to the add-immediate operation but used to create addresses, not to create final result
- Value loaded from memory is stored in `rd`

I-Format Load Example

- **`lw x14, 8(x2)`**



All RV32 Load Instructions

imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

funct3 field encodes size and signedness of load data

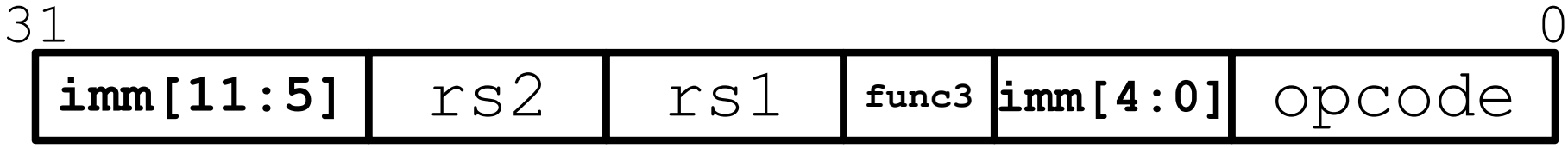
- LBU is “load unsigned byte”
- LH is “load halfword”, which loads 16 bits (2 bytes) and sign-extends to fill destination 32-bit register
- LHU is “load unsigned halfword”, which zero-extends 16 bits to fill destination 32-bit register
- There is no LWU in RV32, because there is no sign/zero extension needed when copying 32 bits from a memory location into a 32-bit register

Agenda

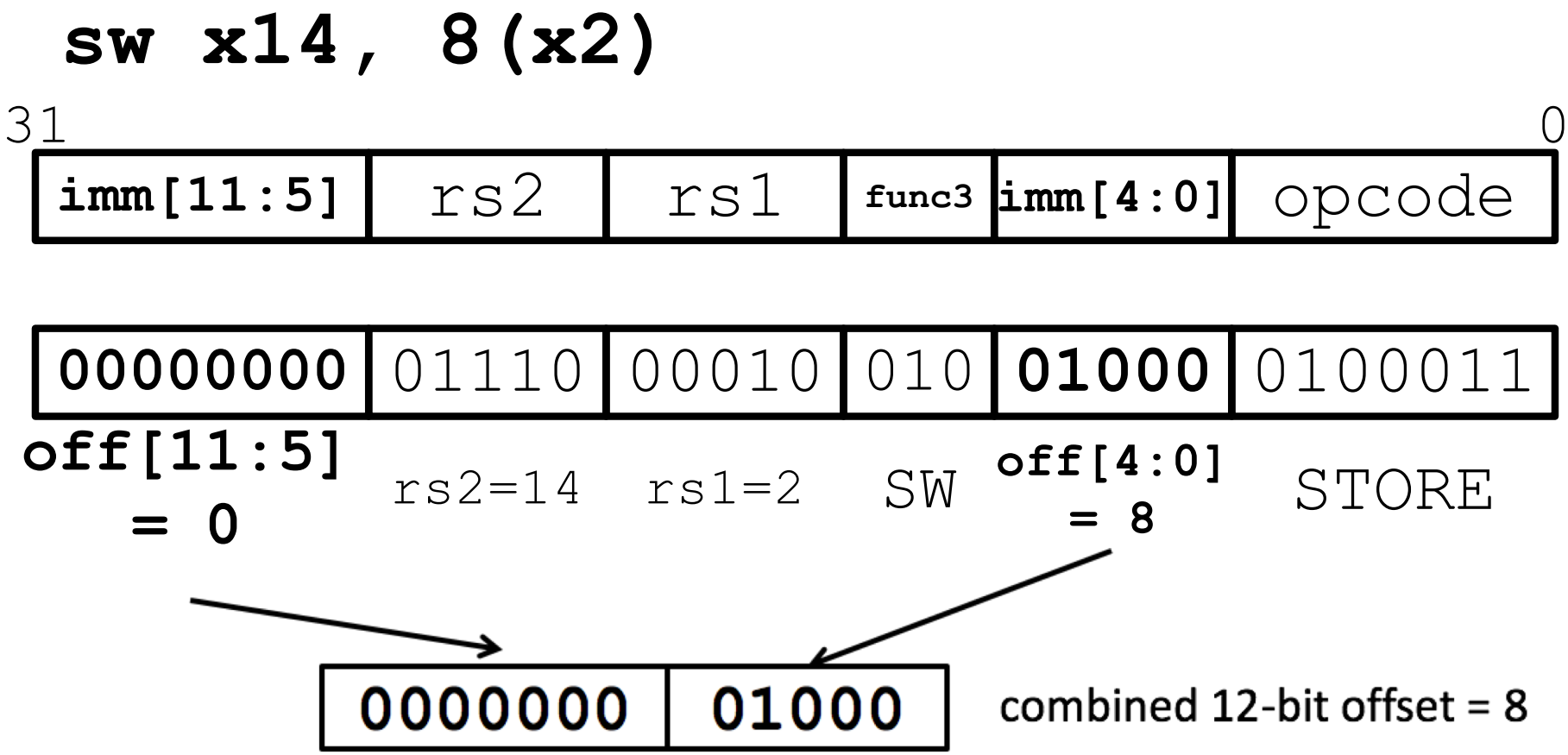
- Stored-Program Concept
- R-Format
- I-Format
- S-Format
- SB-Format
- U-Format
- UJ-Format

S-Format Used for Stores

- Store needs to read two registers, `rs1` for base memory address, and `rs2` for data to be stored, as well as need immediate offset!
- Can't have both `rs2` and immediate in same place as other instructions!
- Note: stores don't write a value to the register file, no `rd`!
- RISC-V design decision is **move low 5 bits of immediate** to where **`rd`** field was in other instructions – keep `rs1`/`rs2` fields in same place
- *register names more critical than immediate bits in hardware design*



S-Format Example



All RV32 Store Instructions

imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

Agenda

- Stored-Program Concept
- R-Format
- I-Format
- Administrivia
- S-Format
- SB-Format
- U-Format
- UJ-Format

Branching Instructions

- `beq`, `bne`, `bge`, `blt`
  - Need to specify an **address** to go to
  - Also take **two** registers to compare
  - Doesn't write into a register (similar to stores)
- How to encode label, i.e., where to branch to?

Branching Instruction Usage

- Branches typically used for loops (`if-else`, `while`, `for`)
  - Loops are generally small (< 50 instructions)
- **Recall:** Instructions stored in a localized area of memory (Code/Text)
  - Largest branch distance limited by size of code
  - Address of current instruction stored in the program counter (PC)

PC-Relative Addressing

- PC-Relative Addressing: Use the `immediate` field as a two's complement *word* offset to PC
  - Branches generally change the PC by a small amount
  - Can specify ± 2^{11} ***addresses*** from the PC

- Why not use *byte* address offset from PC as the `immediate`?

Branching Reach

- **Recall:** RISCV uses 32-bit addresses, and memory is **byte-addressed**
- Instructions are "***word-aligned***": Address is always a multiple of 4 (in bytes)
  - PC ALWAYS points to an instruction

- Let immediate specify #words instead of #bytes
  - Instead of specifying $\pm 2^{11}$ ***bytes*** from the PC, we will now specify $\pm 2^{11}$ ***words*** = $\pm 2^{13}$ byte addresses around PC

Branch Calculation

- If we don't take the branch:

  `PC = PC+4` = next instruction

- If we do take the branch:

  `PC = PC + (immediate*4)`

- **Observations:**
  - `immediate` is number of instructions to move (remember, specifies words) either forward (+) or backwards (–)

RISC-V Feature, n×16-bit instructions

- Extensions to RISC-V base ISA support 16-bit compressed instructions and also variable-length instructions that are multiples of 16-bits in length
- 16-bit = half-word
- To enable this, RISC-V scales the branch offset to be **half-words** even when there are no 16-bit instructions
- Reduces branch reach by half and means that ½ of possible targets will be errors on RISC-V processors that only support 32-bit instructions (as used in this class)
- RISC-V conditional branches can only reach $\pm 2^{10} \times 32$-bit instructions either side of PC

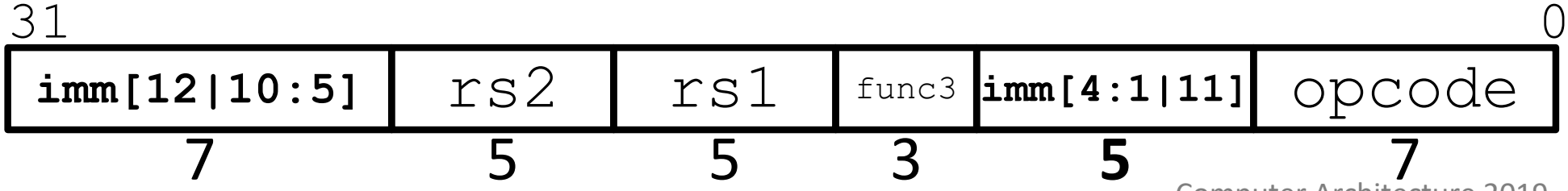
Example Instruction Compression

Code taken from Tock embedded operating system

```
00000000020400094 reset_handler:
; pub unsafe fn reset_handler() {
20400094:   79 71    addi    sp, sp, -48
; rv32i::init_memory();
20400096:   06 d6    sw  ra, 44(sp)
20400098:   26 d4    sw  s1, 40(sp)
2040009a:   4a d2    sw  s2, 36(sp)
2040009c:   4e d0    sw  s3, 32(sp)
2040009e:   52 ce    sw  s4, 28(sp)
204000a0:   56 cc    sw  s5, 24(sp)
204000a2:   97 00 00 00      auipc   ra, 0
204000a6:   e7 80 40 60      jalr    ra, ra, 1540
; asm!("
204000aa:   37 05 40 20      lui a0, 132096
204000ae:   13 05 05 04      addi    a0, a0, 64
204000b2:   73 10 55 30      csrw    mtvec, a0
; unsafe { &*self.ptr }
204000b6:   37 35 40 20      lui a0, 132099
204000ba:   03 25 c5 3d      lw  a0, 988(a0)
204000be:   b7 f5 51 00      lui a1, 1311
204000c2:   93 85 e5 15      addi    a1, a1, 350
204000c6:   4c cd    sw  a1, 28(a0)
204000c8:   23 20 05 00      sw  zero, 0(a0)
204000cc:   4c cd    sw  a1, 28(a0)
204000ce:   b7 f5 09 0d      lui a1, 53407
204000d2:   b5 05    addi    a1, a1, 13
204000d4:   0c cd    sw  a1, 24(a0)
```

RISC-V B-Format for Branches

- B-format is mostly same as S-Format, with two register sources (`rs1`/`rs2`) and a 12-bit immediate
- But now immediate represents values -2^{12} to $+2^{12}-2$ in 2-byte increments
- The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)



Branch Example (1/3)

- RISCV Code:

```
Loop: beq  x19,x10,End
      add  x18,x18,x10
      addi x19,x19,-1
      j    Loop
End:  <target instr>
```

Start counting from instruction AFTER the branch

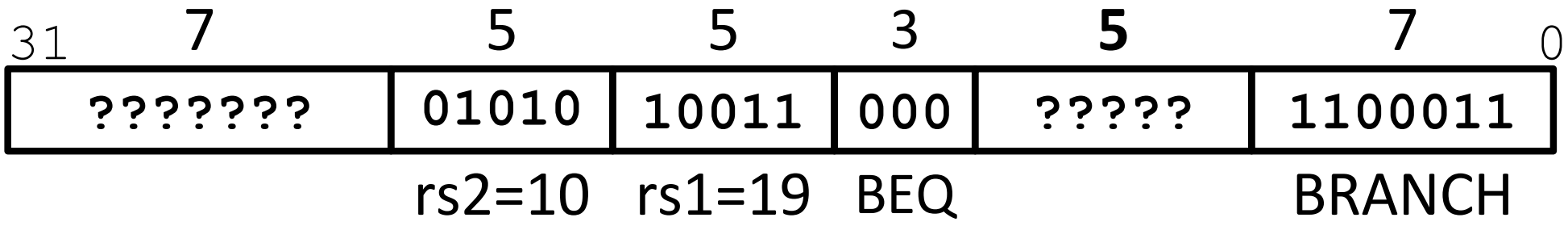
- Branch offset = 4×32-bit instructions = 16 bytes
- (Branch with offset of 0, branches to itself)

Branch Example (2/3)

- RISCV Code:

```
Loop: beq  x19,x10,End
      add  x18,x18,x10
      addi x19,x19,-1
      j    Loop
End:  <target instr>
```

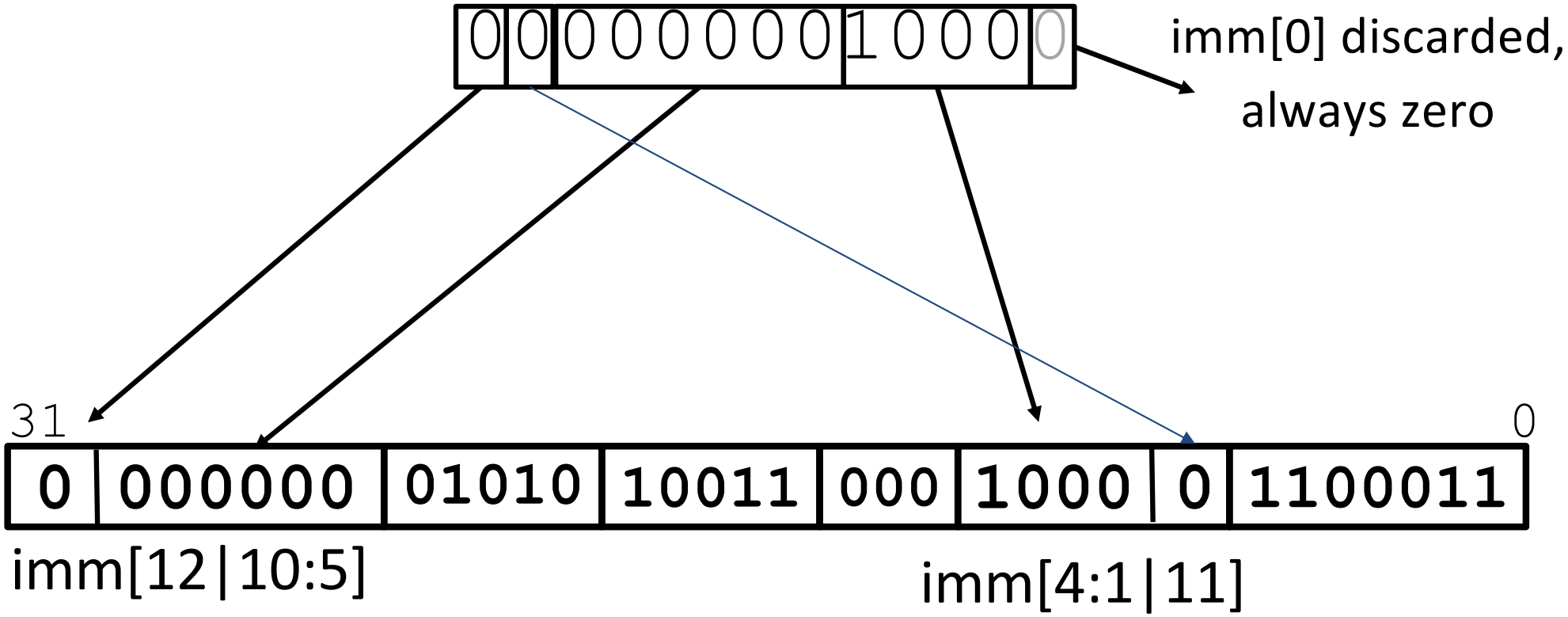
Start counting from instruction AFTER the branch



Branch Example (3/3)

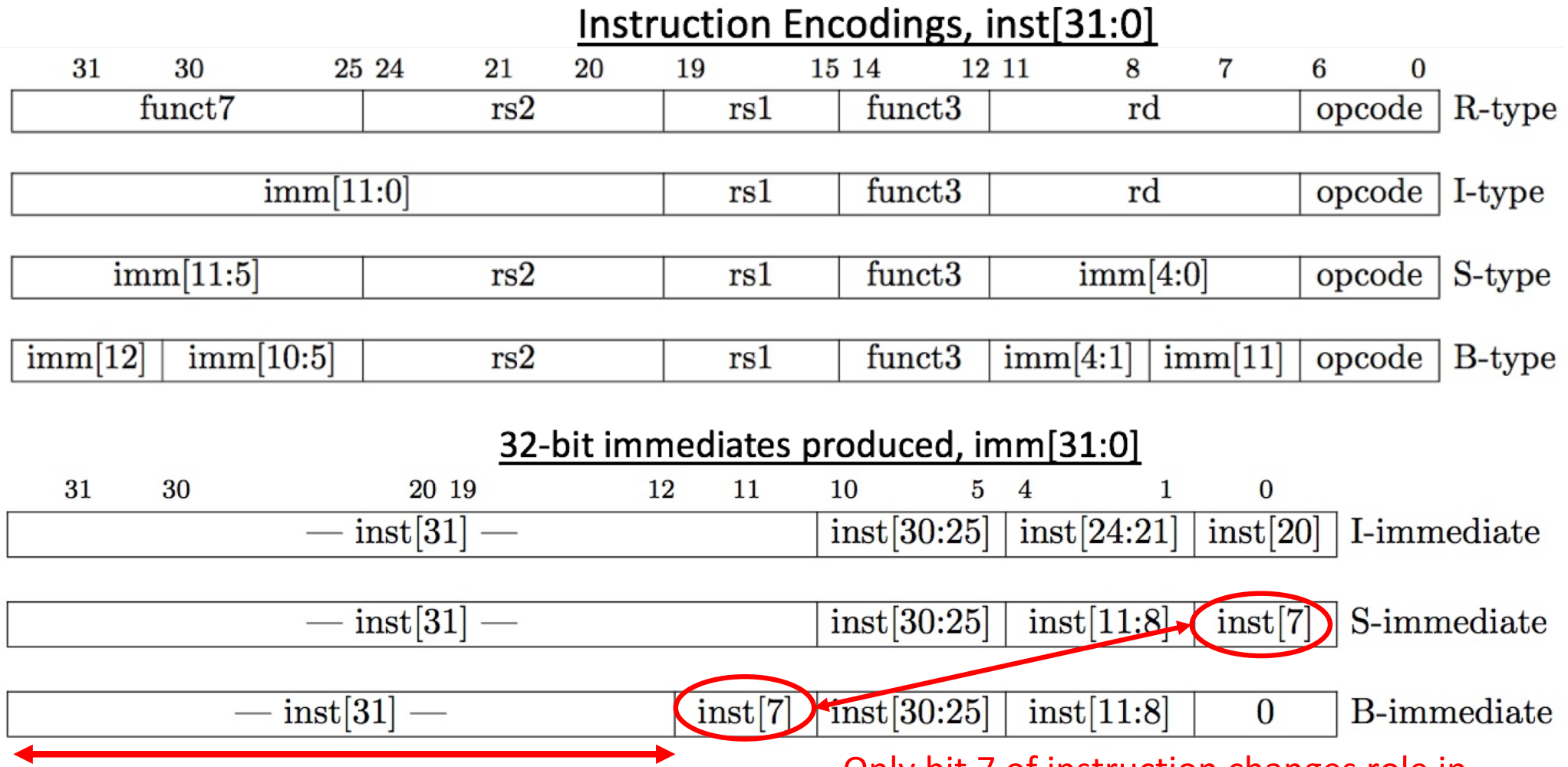
beq x19,x10,offset = 16 bytes

13-bit immediate, imm[12:0], with value 16



RISC-V Immediate Encoding

- Why is it so confusing?!?!



All RISC-V Branch Instructions

imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

Questions on PC-addressing

- Does the value in branch immediate field change if we move the code?
  - If moving individual lines of code, then yes
  - If moving all of code, then no (why?)
- What do we do if destination is > 2^{10} instructions away from branch?
  - Other instructions save us:

```
beq x10,x0,far                bne x10,x0,next
# next instr          →       j   far
                              next: # next instr
```

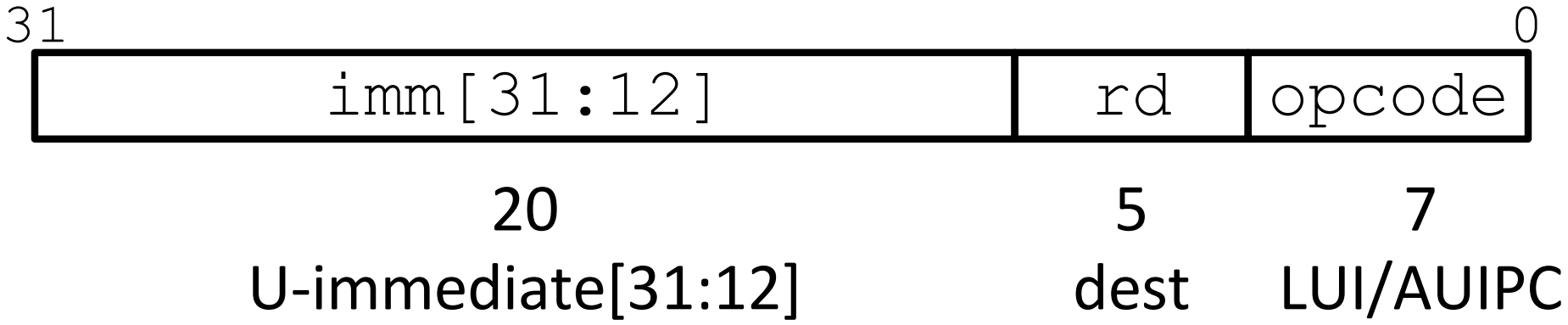
Agenda

- Stored-Program Concept
- R-Format
- I-Format
- Administrivia
- S-Format
- SB-Format
- U-Format
- UJ-Format

Dealing With Large Immediates

- How do we deal with 32-bit immediates?
  - Our I-type instructions only give us 12 bits
- **Solution:** Need a new instruction format for dealing with the rest of the 20 bits.
- This instruction should deal with:
  - a destination register to put the 20 bits into
  - the immediate of 20 bits
  - the instruction opcode

U-Format for “Upper Immediate” instructions



- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, `rd`
- Used for two instructions
  - LUI – Load Upper Immediate
  - AUIPC – Add Upper Immediate to PC

LUI to create long immediates

- `lui` writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits
- Together with an `addi` to set low 12 bits, can create any 32-bit value in a register using two instructions (`lui`/`addi`).

```
lui x10, 0x87654        # x10 = 0x87654000
addi x10, x10, 0x321    # x10 = 0x87654321
```

Corner Case

- How to set 0xDEADBEEF?

```
lui x10, 0xDEADB       # x10 = 0xDEADB000
addi x10, x10,0xEEF    # x10 = 0xDEADAEEF
```

`addi` 12-bit immediate is always sign-extended!

- if top bit of the 12-bit immediate is a 1, it will subtract -1 from upper 20 bits

Solution

- How to set 0xDEADBEEF?

```
lui x10, 0xDEADC      # x10 = 0xDEADC000
addi x10, x10,0xEEF   # x10 = 0xDEADBEEF
```

Pre-increment value placed in upper 20 bits, if sign bit will be set on immediate in lower 12 bits.

Assembler pseudo-op handles all of this:

```
li x10, 0xDEADBEEF  # Creates two instructions
```

AUIPC

- Adds upper immediate value to PC and places result in destination register
- Used for PC-relative addressing

- `Label: auipc x10, 0`
  - Puts address of label into `x10`

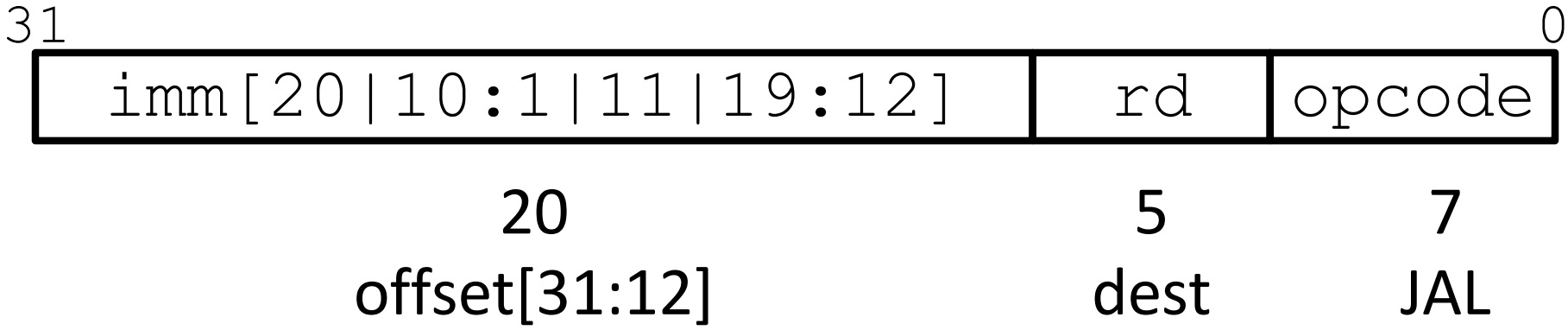
Agenda

- Stored-Program Concept
- R-Format
- I-Format
- Administrivia
- S-Format
- SB-Format
- U-Format
- UJ-Format

UJ-Format Instructions (1/3)

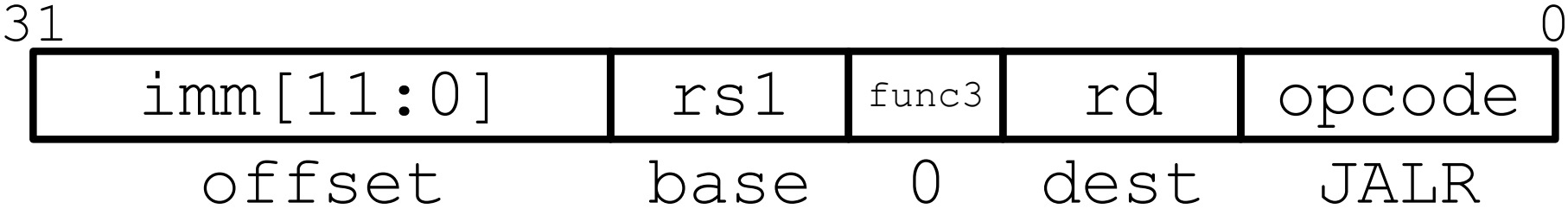
- For branches, we assumed that we won't want to branch too far, so we can specify a *change* in the PC
- For general jumps (`jal`), we may jump to *anywhere* in code memory
  - Ideally, we would specify a 32-bit memory address to jump to
  - Unfortunately, we can't fit both a 7-bit `opcode` and a 32-bit address into a single 32-bit word
  - Also, when linking we must write to an `rd` register

UJ-Format Instructions (2/3)



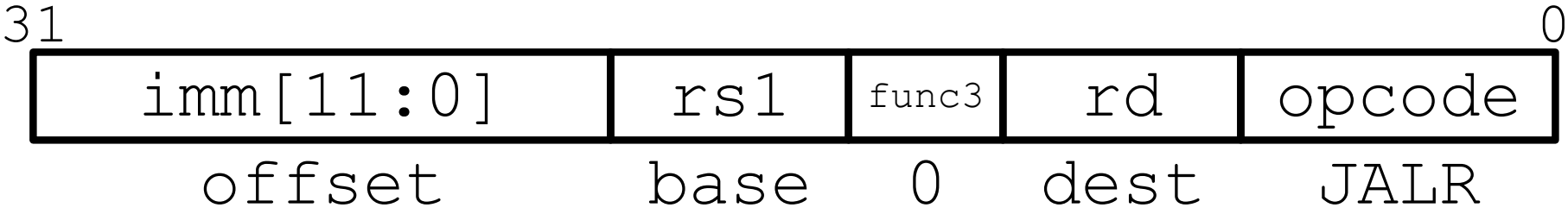
- `jal` saves PC+4 in register `rd` (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
- $\pm 2^{18}$ 32-bit instructions
- Reminder: "j" jump is a pseudo-instruction—the assembler will instead use `jal` but sets rd=x0 to discard return address
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

`jalr` Instruction (I-Format)



- `jalr rd, rs1, offset`
- Writes PC+4 to `rd` (return address)
- Sets PC = `rs1` + `offset`
- Uses same immediates as arithmetic & loads
  - no multiplication by 2 bytes

Uses of `jalr`




```
# ret and jr psuedo-instructions
ret = jr ra = jalr x0, ra, 0
# Call function at any 32-bit absolute address
lui x1, <hi 20 bits>
jalr ra, x1, <lo 12 bits>
# Jump PC-relative with 32-bit offset
auipc x1, <hi 20 bits>
jalr x0, x1, <lo 12 bits>
```

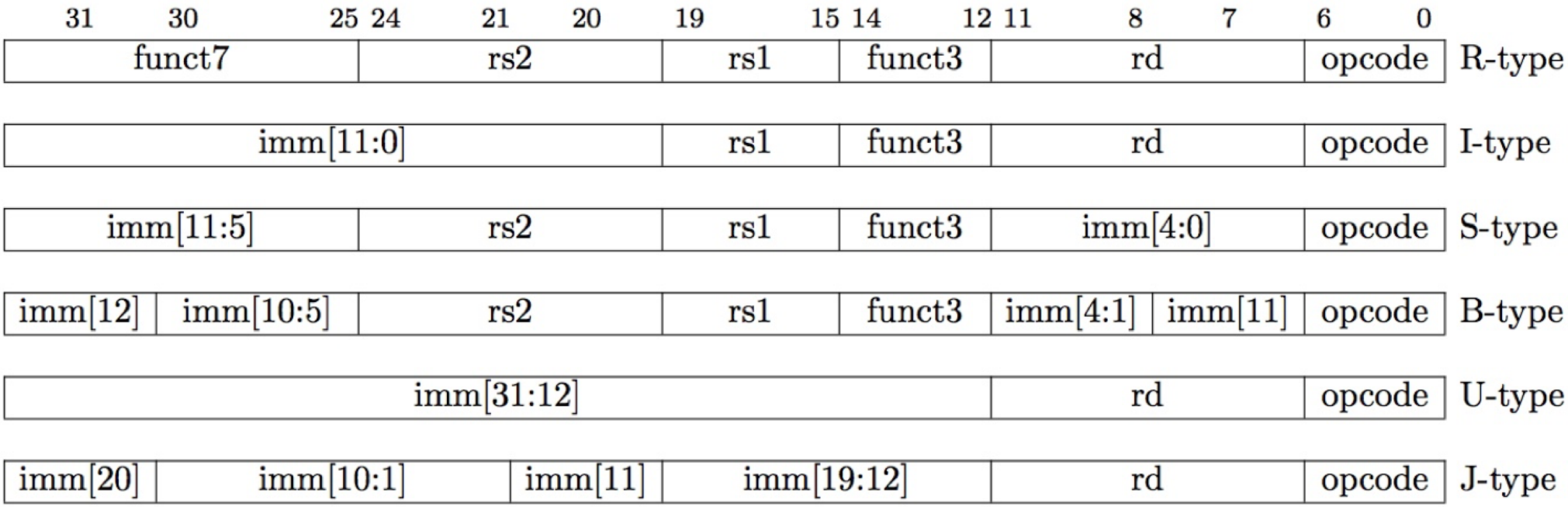
Question: When combining two C files into one executable, we can compile them independently and then merge them together.

When merging two or more binaries:

1) Jump instructions don't require any changes
2) Branch instructions don't require any changes

	1	2
(A)	F	F
(B)	F	T
(C)	T	F
(D)	T	T

Summary of RISC-V Instruction Formats



Summary

- The Stored Program concept is very powerful
  - Instructions can be treated and manipulated the same way as data in both hardware and software
- RISCV Machine Language Instructions:

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI

0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK
csr			rs1	001	rd	1110011	CSRRW
csr			rs1	010	rd	1110011	CSRRS
csr			rs1	011	rd	1110011	CSRRC
csr			zimm	101	rd	1110011	CSRRWI
csr			zimm	110	rd	1110011	CSRRSI
csr			zimm	111	rd	1110011	CSRRCI

Not in CS61C