ECE 281

Lesson 26:
RV32I Branching

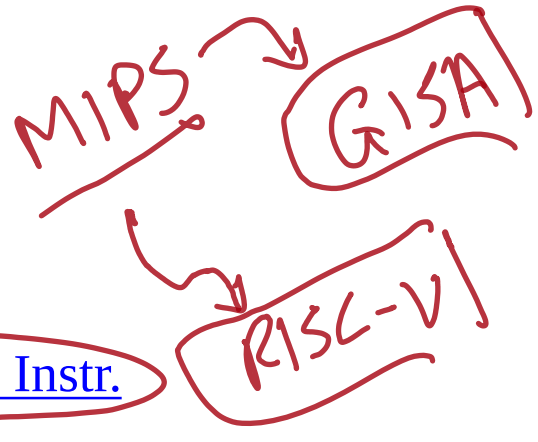
Digital Design and Computer Architecture Lecture Notes

Modified for use by USAF Academy, 2025

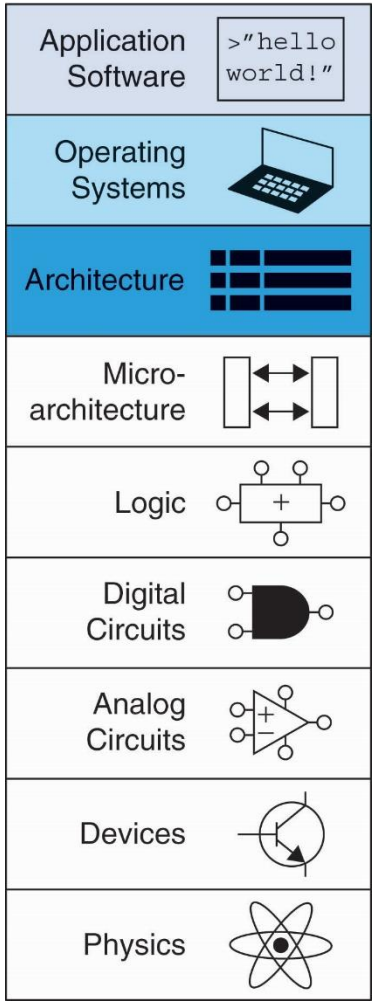
Chapter 6 :: Topics

Videos

- Branches
- Conditional Statements & Loops
- Arrays
- Machine Language: I, S/B, U/J-Type Instr.



Lesson	Topic	Reading	Assigned	Due
26	RV32I U-, S-, B-Type Instructions	6.3.3-6.3.6, 6.4.3-		
27	ICE5 - Basic Elevator Controller		Lab 4 Prelab	Lab 3
28	ICE6 - Time Division Multiplexing			ICE5
29	Lab 4 - Moore Elevator Controller			ICE6, Lab 4 Prelab
30	Lab 4 - Moore Elevator Controller		HW 30	
31	GR #2			Lab 4



Chapter 6: Architecture

Memory Operands

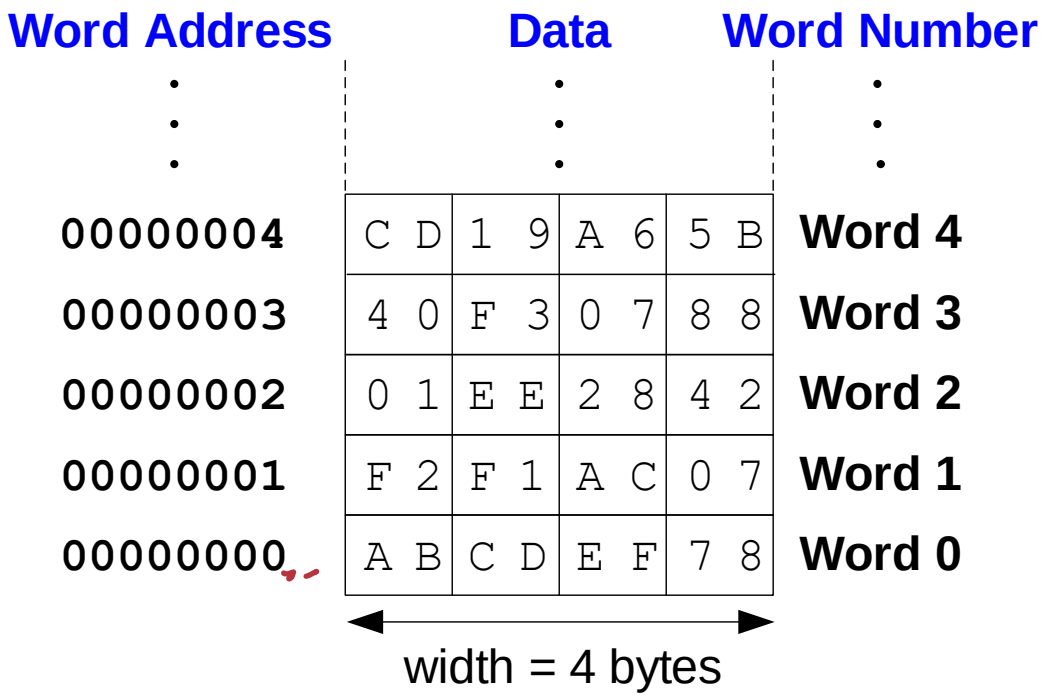
Memory

- First, we'll discuss **word-addressable** memory
- Then we'll discuss **byte-addressable** memory

RISC-V is **byte-addressable**

Word-Addressable Memory

- Each 32-bit data word has a unique address



RISC-V uses **byte-addressable** memory, which we'll talk about next.

Reading Word-Addressable Memory

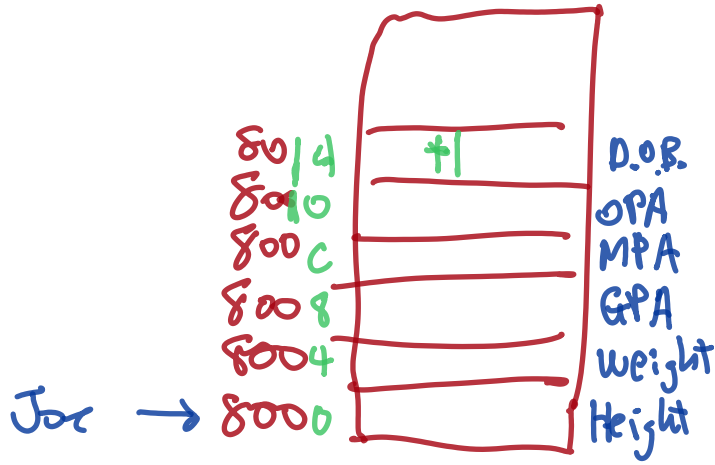
- Memory read called ***load***
- **Mnemonic:** *load word* (`lw`) ← I-type Instruction
- **Format:**

```
lw t1, 5(s0)
lw destination, offset(base)
```

t1 <= (s0+5)

- **Address calculation:**
  - add *base address* (`s0`) to the *offset* (5)
  - address = (`s0` + 5)
- **Result:**
  - `t1` holds the data value at address (`s0` + 5)

Any register may be used as base address



Reading Word-Addressable Memory

- **Example:** read a word of data at memory address 1 into `s3`
  - address = (0 + 1) = 1
  - `s3` = 0xF2F1AC07 after load

Assembly code

```
lw s3, 1(zero) # read memory word 1 into s3
```

Word Address	Data	Word Number
00000004	C D 1 9 A 6 5 B	Word 4
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	**F 2 F 1 A C 0 7**	Word 1
00000000	A B C D E F 7 8	Word 0

Writing Word-Addressable Memory

- Memory write is called a ***store***
- **Mnemonic:** *store word* (`sw`)

Writing Word-Addressable

- **Example:** Write (store) the value in `t4` into memory address 3
  - add the base address (`zero`) to the offset (0x3)
  - address: (0 + 0x3) = 3
  - for example, if `t4` holds the value 0xFEEDCABB, then after this instruction completes, word 3 in memory will contain that value

Offset can be written in **decimal** (default) or **hexadecimal**

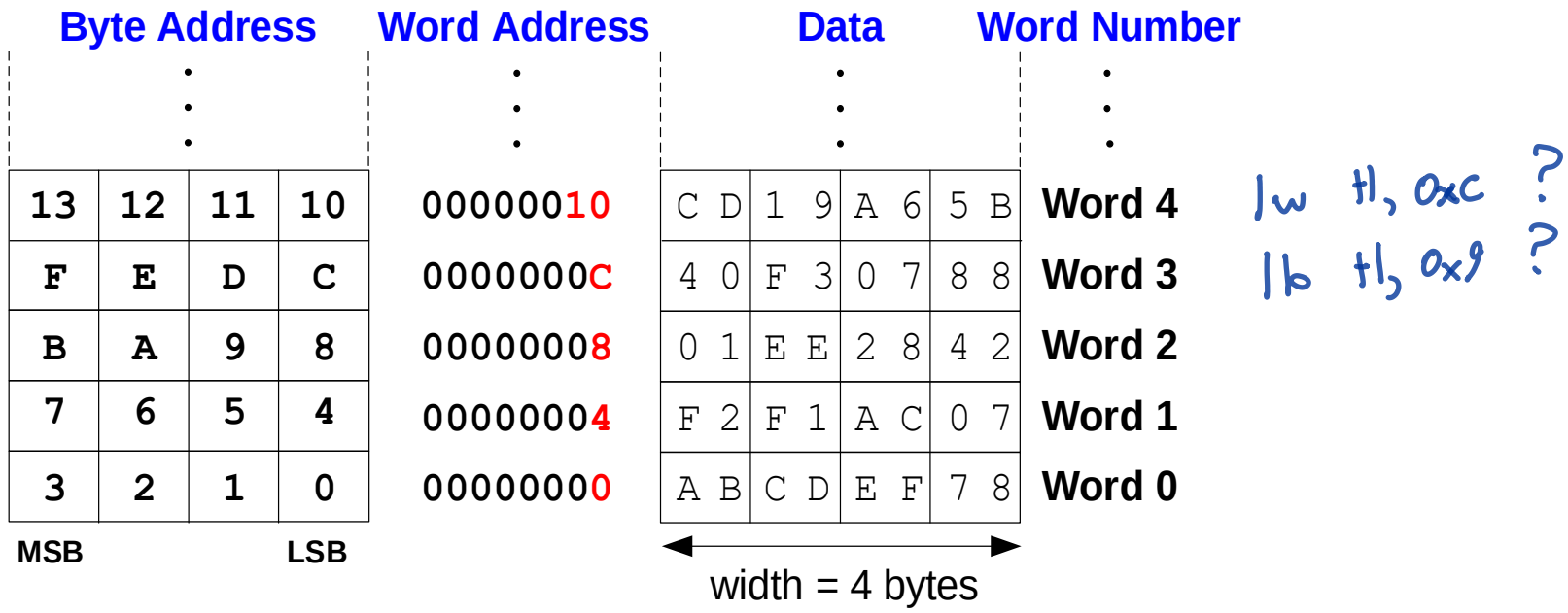
Assembly code

```
sw t4, 0x3(zero)   # write the value in t4
                   # to memory word 3
```

Word Address	Data	Word Number
00000004	C D 1 9 A 6 5 B	Word 4
00000003	**F E E D C A B B**	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Byte-Addressable Memory

- Each data byte has a unique address
- Load/store words or single bytes: load byte (`lb`) and store byte (`sb`)
- 32-bit word = 4 bytes, so word address **increments by 4**



Reading Byte-Addressable Memory

- The address of a memory word must now be multiplied by 4. For example,
  - the address of memory word 2 is $2 \times 4 = 8$
  - the address of memory word 10 is $10 \times 4 = 40$ (0x28)

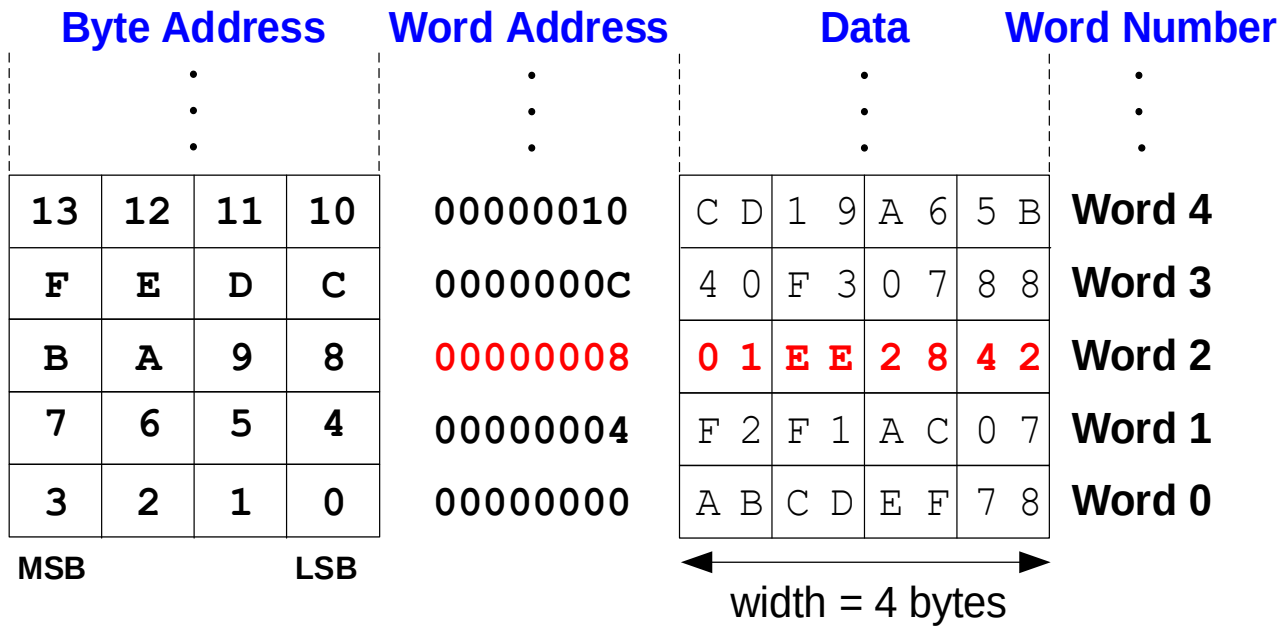
- RISC-V is **byte-addressed**, not word-addressed

Reading Byte-Addressable Memory

- **Example:** Load a word of data at memory address 8 into `s3`.
- `s3` holds the value 0x1EE2842 after load

RISC-V assembly code

```
lw s3, 8(zero)  # read word at address 8 into s3
```

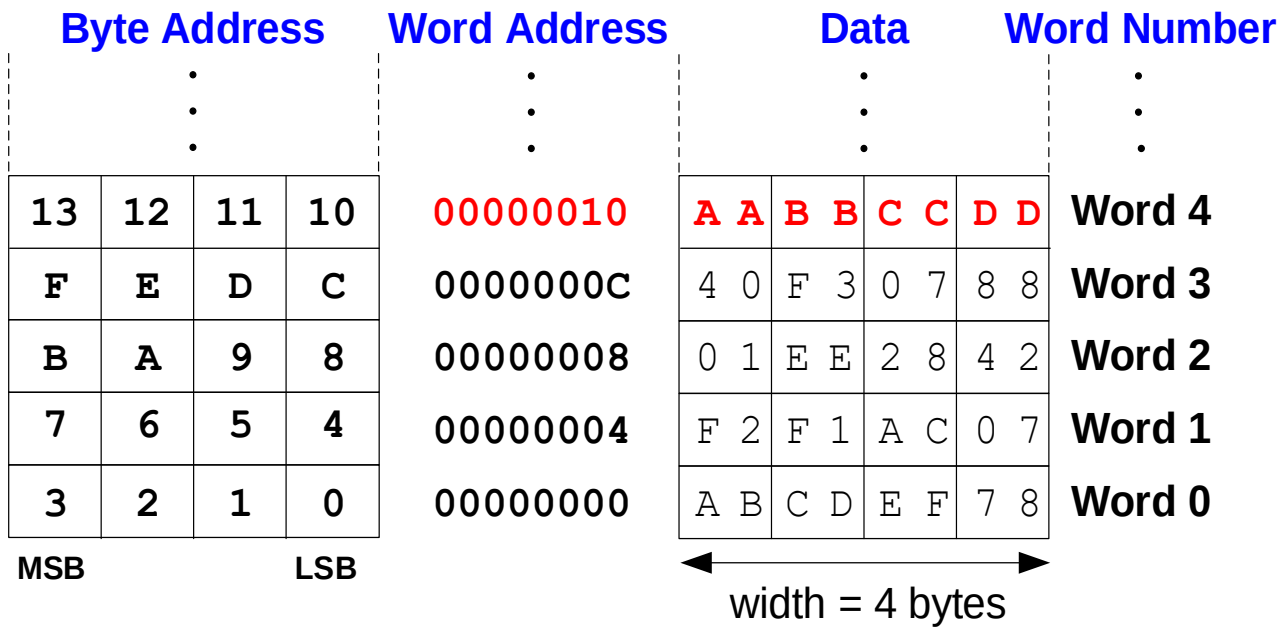


Writing Byte-Addressable Memory

- **Example:** store the value held in `t7` into memory address 0x10 (16)
  - if `t7` holds the value 0xAABBCCDD, then after the `sw` completes, word 4 (at address 0x10) in memory will contain that value

RISC-V assembly code

```
sw t7, 0x10(zero)  # write t7 into address 16
```



Chapter 6: Architecture

Branches & Jumps

Branching

- Execute instructions out of sequence
- **Labels** indicate instruction location. They can't be reserved words and must be followed by a colon (:)
- Types of branches:
  - **Conditional**
    - branch if equal (`beq`)
    - branch if not equal (`bne`)
    - branch if less than (`blt`)
    - branch if greater than or equal (`bge`)
  - **Unconditional**
    - jump (`j`)
    - jump register (`jr`)
    - jump and link (`jal`)
    - jump and link register (`jalr`)

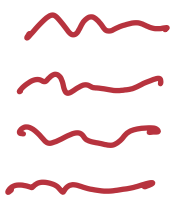
C-code

```
if s0 == s1
    s1 = s0 + s1;
else
    s1 = (s1+1) - s0;
Next:
```

We'll talk about these when discuss function calls

The Branch Not Taken (beq)

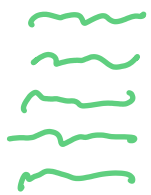
```
# RISC-V assembly
    addi    s0, zero, 4          # s0 = 0 + 4 = 4
    addi    s1, zero, 1          # s1 = 0 + 1 = 1
    slli    s1, s1, 2            # s1 = 1 << 2 = 4
    beq     s0, s1, target       # branch to label
    addi    s1, s1, 1            # not executed
    sub     s1, s1, s0           # not executed
    j       next
target:
    add     s1, s1, s0           # s1 = 4 + 4 = 8
next:
```



The Branch Not Taken (bne)

Instruction Address

```
# RISC-V assembly
8000    addi    s0, zero, 4          # s0 = 0 + 4 = 4
8004    addi    s1, zero, 1          # s1 = 0 + 1 = 1
8008    slli    s1, s1, 2            # s1 = 1 << 2 = 4
800C    bne     s0, s1, target       # branch not taken
8010    addi    s1, s1, 1            # s1 = 4 + 1 = 5
8014    sub     s1, s1, s0           # s1 = 5 - 4 = 1
8018    j       Next
        target:
801C        add     s1, s1, s0       # s1 = 1 + 4 = 5
8020    next:
```



Unconditional Branching (j)

```
# RISC-V assembly
    j       target          # jump to target
    srai    s1, s1, 2       # not executed
    addi    s1, s1, 1       # not executed
    sub     s1, s1, s0      # not executed

target:
    add     s1, s1, s0      # s1 = 1 + 4 = 5
```

Chapter 6: Architecture

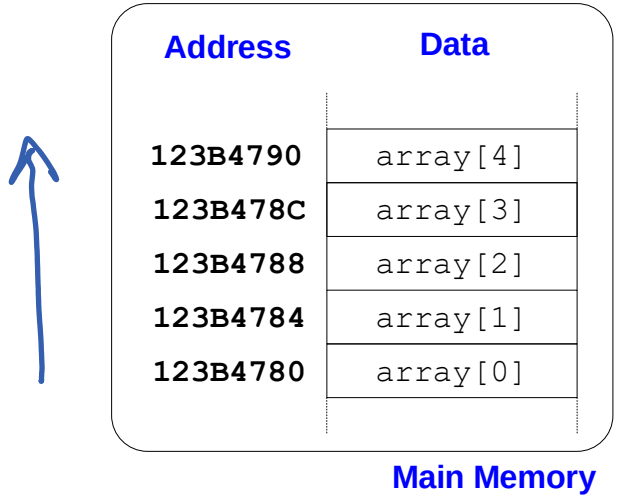
Arrays

Arrays

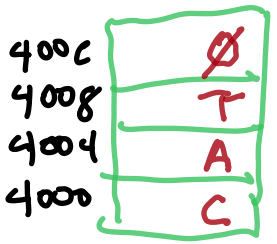
- Access large amounts of similar data
- **Index**: access each element
- **Size**: number of elements

Arrays

- 5-element array
- **Base address** = 0x123B4780 (address of first element, `array[0]`)
- First step in accessing an array: load base address into a register



Accessing Arrays of Characters




```
// C Code
   char str[80] = "CAT";  // all strings terminate with \0
   int len = 0;

   // compute length of string
   // while() will terminate on null character \0 at end of string
   while (str[len]) len++;
```

```
# RISC-V assembly code
# s0 = array base address, s1 = len

      addi s1, zero, 0       # len = 0
while: add t0, s0, s1        # address of str[len]
       lw t1, 0(t0)          # load str[len]
       beq  t1, zero, done   # are we at the end of the string?
       addi s1, s1, 1        # len++
       j while               # repeat while loop
done:
```

Chapter 6: Architecture

Machine Language

S/B-Type

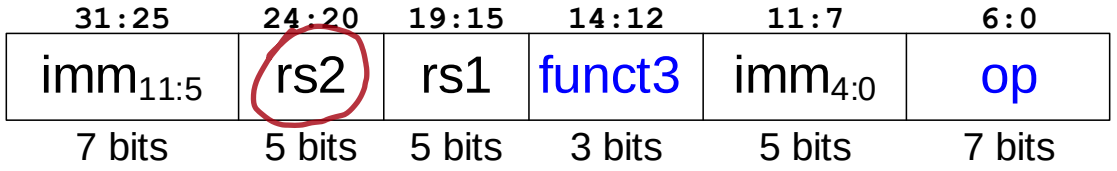
- ***Store-Type***
- ***Branch-Type***
- Differ only in immediate encoding

31:25	24:20	19:15	14:12	11:7	6:0	
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

S-Type

- ***Store-Type***
- 3 operands:
  - `rs1`: base register
  - `rs2`: value to be stored to memory
  - `imm`: 12-bit two's complement immediate
- Other fields:
  - `op`: the opcode
    - Simplicity favors regularity: all instructions have opcode
  - `funct3`: the function (3-bit function code)
    - with opcode, tells computer what operation to perform

S-Type



S-Type Examples

store word
store half word
store byte

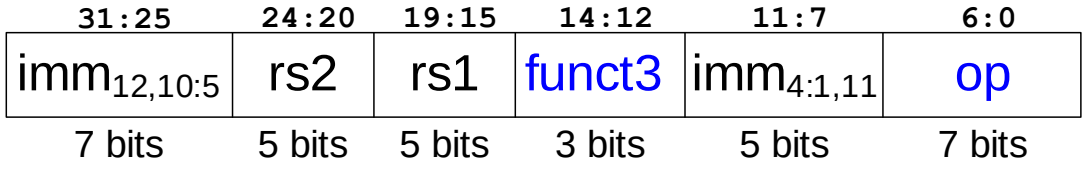
Assembly	imm$_{11:5}$	rs2	rs1	funct3	imm$_{4:0}$	op	imm$_{11:5}$	rs2	rs1	funct3	imm$_{4:0}$	op	
`sw t2, -6(s3)`	1111 111	7	19	2	11010	35	1111 111	00111	10011	010	11010	010 0011	(0xFE79AD23)
`sw x7, -6(x19)`													
`sh s4, 23(t0)`	0000 000	20	5	1	10111	35	0000 000	10100	00101	001	10111	010 0011	(0x01429BA3)
`sh x20,23(x5)`													
`sb t5, 0x2D(zero)`	0000 001	30	0	0	01101	35	0000 001	11110	00000	000	01101	010 0011	(0x03E006A3)
`sb x30,0x2D(x0)`	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

B-Type

- ***Branch-Type*** (similar format to S-Type)
- 3 operands:
  - `rs1`: register source 1
  - `rs2`: register source 2
  - $\texttt{imm}_{12:1}$: 12-bit two's complement immediate – address offset
- Other fields:
  - `op`: the opcode
    - Simplicity favors regularity: all instructions have opcode
  - `funct3`: the function (3-bit function code)
    - with opcode, tells computer what operation to perform

b a ==b

B-Type



B-Type Example

- The 13-bit immediate encodes where to branch (relative to the branch instruction)
- Immediate encoding is strange
- **Example:**

changes for 485 H&P

PC = PC + 4

↳ then relative addr

```
# RISC-V Assembly
0x70      beq  s0, t5, L1   1
0x74      add  s1, s2, s3   2
0x78      sub  s5, s6, s7   3
0x7C      lw   t0, 0(s1)    4
0x80 L1:  addi s1, s1, -15
```

L1 is 4 instructions (i.e., **16 bytes**) past `beq`

shift 1 bit or 2 bits ?
(2) (4)

$imm_{12:0}$ = 16	0	0	0	0	0	0	0	1	0	0	0	0
bit number	12	11	10	9	8	7	6	5	4	3	2	1	0

Assembly

```
beq s0, t5,  L1
beq x8, x30, 16
```

Field Values

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
0000 000	30	8	0	1000 0	99
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Machine Code

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
0000 000	11110	01000	000	1000 0	110 0011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

(0x01E40863)

Review: Instruction Formats

7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
funct7	rs2	rs1	funct3	rd	op	**R-Type**
$imm_{11:0}$		rs1	funct3	rd	op	**I-Type**
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**
$imm_{31:12}$				rd	op	**U-Type**
$imm_{20,10:1,11,19:12}$				rd	op	**J-Type**
20 bits				5 bits	7 bits	

Example Program: RISC-V Assembly

Address	Machine Code		RISC-V Assembly Code
10144:	ff010113	func:	addi sp,sp,-16
10148:	00112623		sw ra,12(sp)
1014c:	00812423		sw s0,8(sp)
10150:	00050413		mv s0,a0
10154:	00a58533		add a0,a1,a0
10158:	0005da63		bgez a1,1016c <func+0x28>
1015c:	00c12083		lw ra,12(sp)
10160:	00812403		lw s0,8(sp)
10164:	01010113		addi sp,sp,16
10168:	00008067		ret
1016c:	fff58593		addi a1,a1,-1
10170:	00040513		mv a0,s0
10174:	fd1ff0ef		jal ra,10144 <func>
10178:	00850533		add a0,a0,s0
1017c:	fe1ff06f		j 1015c <func+0x18>