EE-222: Microprocessor Systems

RISC vs. CISC Architectures

Instructor: Dr. Arbab Latif

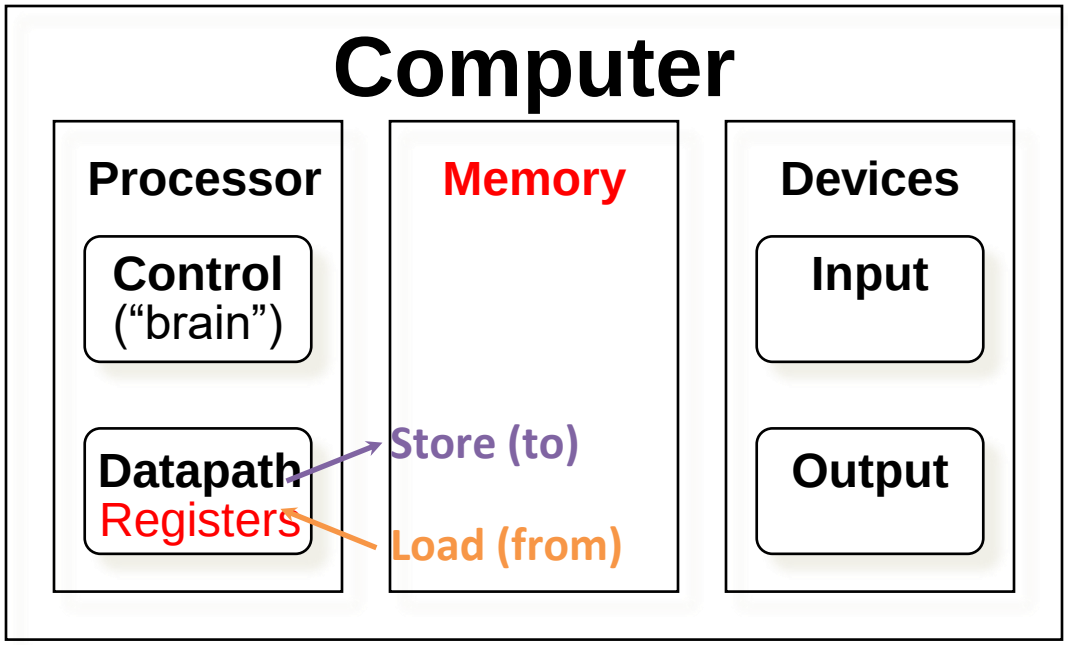
NUST

SCHOOL OF ELECTRICAL ENGINEERING &
COMPUTER SCIENCE (SEECS)

Data Transfer Instructions

Data Transfer Instructions

- Data Transfer instructions are between registers (Datapath) and Memory
  - Allow us to fetch and store operands in memory



Data Transfer

- C variables map onto registers;
  What about large data structures like arrays?
  - Don't forget *memory*, our one-dimensional array indexed by addresses starting at 0

- RISCV instructions only operate on registers!

- Specialized data transfer instructions move data between registers and memory
  - Store (sw): register TO memory
  - Load (lw): register FROM memory

Data Transfer

- Instruction syntax for data transfer:

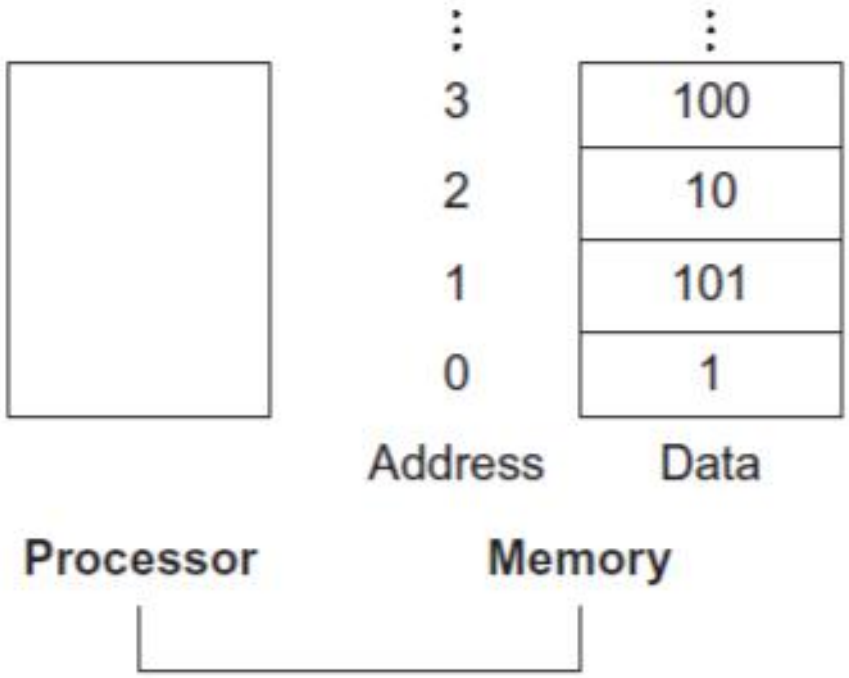
```
memop reg, off(bAddr)
```

  - `memop` = operation name ("operator")
  - `reg` = register for operation source or destination
  - `bAddr` = register with pointer to memory ("base address")
  - `off` = address offset (immediate) in bytes ("offset")

- Accesses memory at address `bAddr+off`
- **Reminder:** A register holds a word of raw data (no type) – make sure to use a register (and offset) that point to a valid memory address

Memory

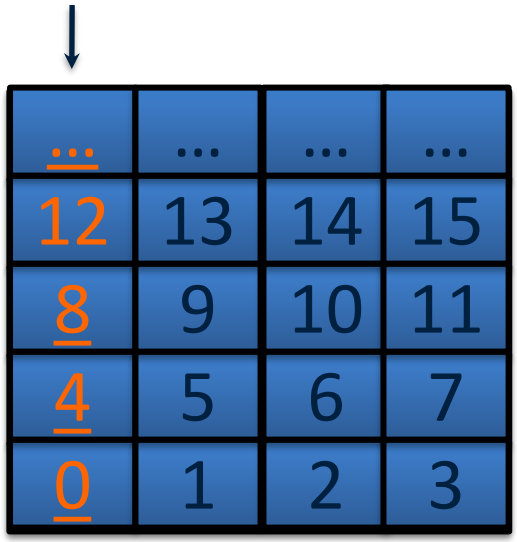
- Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0.



Memory is Byte-Addressed

- What was the smallest data type we saw in C?
  - A char, which was a *byte* (8 bits)
  - Everything in multiples of 8 bits (e.g. 1 word = 4 bytes)

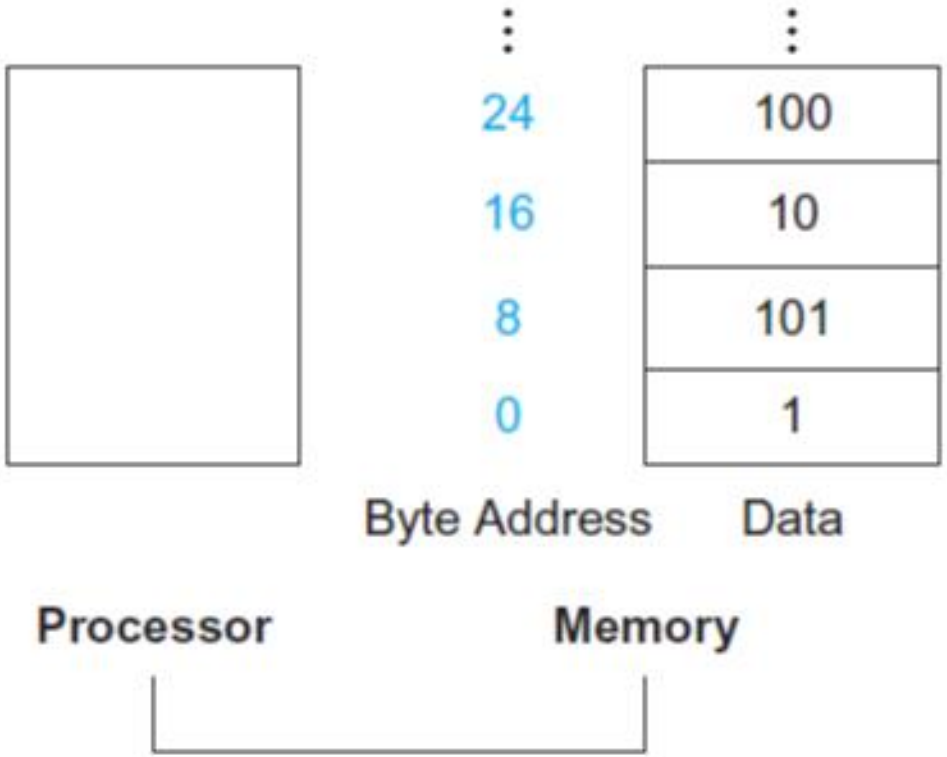
Assume here addr of lowest byte in word is addr of word



- Memory addresses are indexed by *bytes*, not words
- Word addresses are 4 bytes apart
  - Word addr is same as left-most byte
  - Addrs must be multiples of 4 to be "word-aligned"
- Pointer arithmetic not done for you in assembly
  - Must take data size into account yourself

Double-word Memory Address

- Double word addresses are 8 bytes



Data Transfer Instructions

- Load Word (`lw`)
  - Takes data at address `bAddr+off` FROM memory and places it into `reg`
- Store Word (`sw`)
  - Takes data in `reg` and stores it TO memory at address `bAddr+off`
- Example Usage:

```
# addr of int A[] -> s3, a -> s2
lw   t0,12(s3) # $t0=A[3]
add  t0,s2,t0 # $t0=A[3]+a
sw   t0,40(s3) # A[10]=A[3]+a
```

Remember! Pointer arithmetic not done for you in assembly
—Must take data size into account yourself

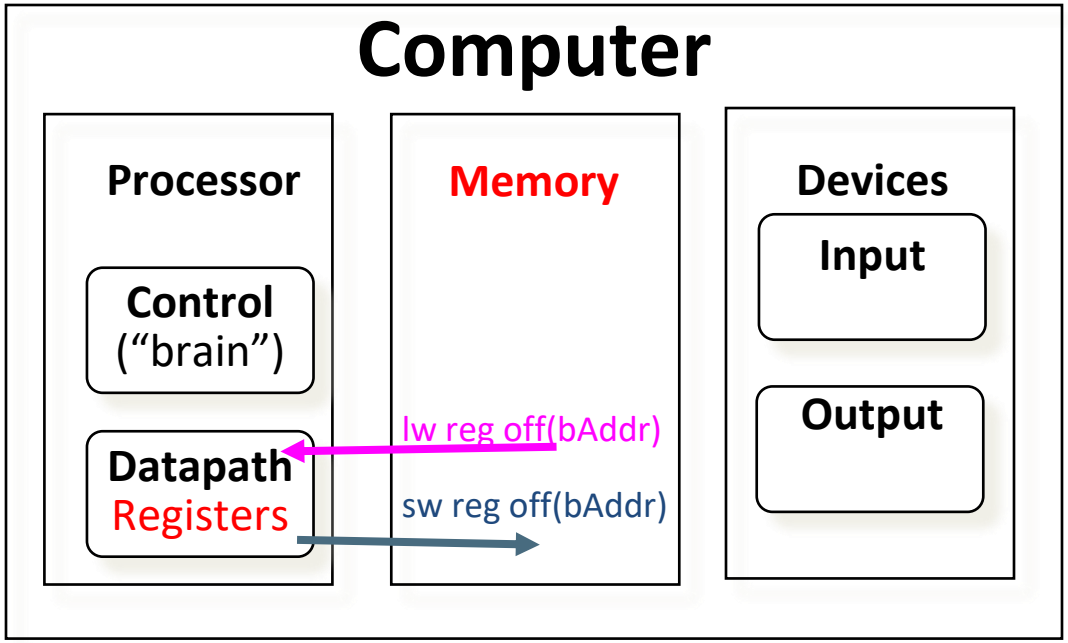
Recommended Reading

- RISC-V Edition - Computer Organization and Design_ The Hardware Software Interface - David A. Patterson, John L. Hennessy:
  - Chapter-2

Data Transfer Instructions Cont.…

Memory and Variable Size

- So Far:
  - lw reg, off(bAddr)
  - sw reg, off(bAddr)



- What about characters (1B) and shorts (sometimes 2B), etc?
  - Want to be able to use interact with memory values smaller than a word.

Trading Bytes with Memory (2 approaches)

- Method 1: Move words in and out of memory using bit-masking and shifting

```
lw   s0,0(s1)
andi s0,s0,0xFF # lowest byte
```

- Method 2: Load/store byte instructions

```
lb   s1,1(s0)
sb   s1,0(s0)
```

*(s0) = 0x00000180

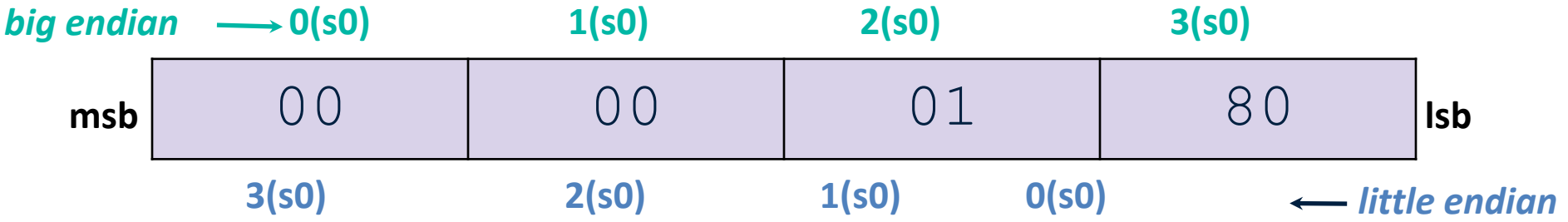
00	00	01	80

Endianness

- Big Endian: Most-significant byte at least address of word
  - word address = address of most significant byte

- Little Endian: Least-significant byte at least address of word
  - word address = address of least significant byte

```
*(s0) = 0x00000180
```



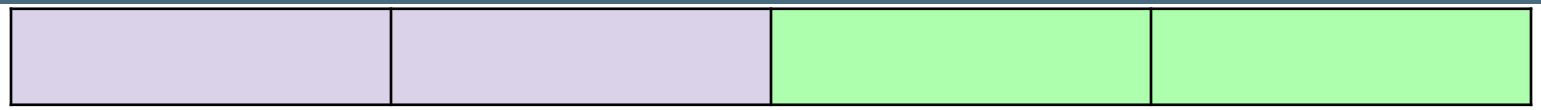
- RISC-V is Little Endian

Byte Instructions

- `lb`/`sb` utilize the least significant byte of the register
  - On `sb`, upper 24 bits are ignored
  - On `lb`, upper 24 bits are filled by sign-extension
- For example, let `*(s0) = 0x00000180`:

```
lb s1,1(s0)   # s1=0x00000001
lb s2,0(s0)   # s2=0xFFFFFF80
sb s2,2(s0)   # *(s0)=0x00800180
```

Half-Word Instructions



- `lh reg, off(bAddr)` "load half"
- `sh reg, off(bAddr)` "store half"
  - On `sh`, upper 16 bits are ignored
  - On `lh`, upper 16 bits are filled by sign-extension

Unsigned Instructions

- `lhu reg, off(bAddr)` "load half unsigned"
- `lbu reg, off(bAddr)` "load byte unsigned"
  - On `l(b/h)u`, upper bits are filled by zero-extension
- Why no s(h/b)u? Why no lwu?

Decision Making Instructions

Computer Decision Making

- In C, we had *control flow*
  - Outcomes of comparative/logical statements determined which blocks of code to execute

- In RISCV, we can't define blocks of code; all we have are labels
  - Defined by text followed by a colon (e.g. `main:`) and refers to the instruction that follows
  - Generate control flow by jumping to labels
  - C has these too, but they are considered bad style

Decision Making Instructions

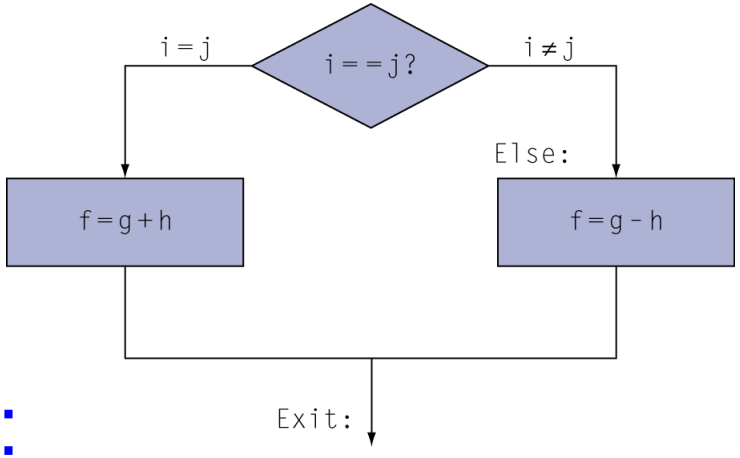
- Branch If Equal (`beq`)
  - `beq reg1,reg2,label`
  - If value in `reg1` = value in `reg2`, go to `label`
- Branch If Not Equal (`bne`)
  - `bne reg1,reg2,label`
  - If value in `reg1` ≠ value in `reg2`, go to `label`
- Jump (`j`)
  - `j label`
  - Unconditional jump to `label`

Breaking Down the If Else

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, … in x19, x20, …



- Compiled RISC-V code:

```
        bne x22, x23, Else
        add x19, x20, x21
        beq x0,x0,Exit // unconditional
Else:   sub x19, x20, x21
Exit:   …
```

Assembler calculates addresses

Breaking Down the If Else

C Code:

```
if(i==j) {
   a = b  /* then */
} else {
   a = -b /* else */
}
```

In English:

- If TRUE, execute the THEN block
- If FALSE, execute the ELSE block

RISCV (beq):

```
# i→s0, j→s1
# a→s2, b→s3

beq s0,s1,???
???    ——— This label unnecessary
sub s2, x0, s3
j   end
then:
add s2, s3, x0
end:
```

Breaking Down the If Else

C Code:

```
if(i==j) {
   a = b  /* then */
} else {
   a = -b /* else */
}
```

In English:

- If TRUE, execute the THEN block
- If FALSE, execute the ELSE block

RISCV (bne):

```
# i→s0, j→s1
# a→s2, b→s3

bne s0,s1,???
???
add s2, s3, x0
j   end
else:
sub s2, x0, s3
end:
```

Branching on Conditions other than (Not) Equal

- Branch Less Than (blt)
  - `blt reg1,reg2, label`
  - If value in `reg1` < value in `reg2`, go to `label`
- Branch Greater Than or Equal (bge)
  - `bge reg1,reg2, label`
  - If value in `reg1` >= value in `reg2`, go to `label`

- Example
  - if (a > b) a += 1;
  - a in x22, b in x23

```
    bge  x23, x22, Exit       // branch if b >= a
    addi x22, x22, 1
Exit:
```

Signed vs. Unsigned

- Signed comparison: blt, bge
- Unsigned comparison: bltu, bgeu

- Example
  - x22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - x23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - x22 < x23 // signed
    - –1 < +1
  - x22 > x23 // unsigned
    - +4,294,967,295 > +1

Branching on Conditions other than (Not) Equal

- Set Less Than (slt)
  - `slt dst, reg1,reg2`
  - If value in `reg1` < value in `reg2`, `dst` = 1, else 0

- Set Less Than Immediate (slti)
  - `slti dst, reg1,imm`
  - If value in `reg1` < `imm`, `dst` = 1, else 0

Recommended Reading

- RISC-V Edition - Computer Organization and Design_ The Hardware Software Interface - David A. Patterson, John L. Hennessy:
  - Chapter-2

RISC-V Bitwise Instructions

RISCV Bitwise Instructions

Note: a→s1, b→s2, c→s3

Instruction	C	RISCV
And	`a = b & c;`	`and  s1,s2,s3`
And Immediate	`a = b & 0x1;`	`andi s1,s2,0x1`
Or	`a = b | c;`	`or  s1,s2,s3`
Or Immediate	`a = b | 0x5;`	`ori  s1,s2,0x5`
Exclusive Or	`a = b ^ c;`	`xor  s1,s2,s3`
Exclusive Or Immediate	`a = b ^ 0xF;`	`xori s1,s2,0xF`

RISC-V Shifting Instructions

Shifting Instructions

- Three kinds:
  - Logical shift: Add zeros as you shift
  - Arithmetic shift: Sign-extend as you shift
    - Only applies when you shift right (preserves sign)
  - Shift by immediate or value in a register

- In binary, shifting an unsigned number left is the same as multiplying by the corresponding power of 2
  - Shifting operations are faster

Shifting Instructions

Instruction Name	RISCV
Shift Left Logical	`sll  s1,s2,s3`
Shift Left Logical Imm	`slli s1,s2,imm`
Shift Right Logical	`srl  s1,s2,s3`
Shift Right Logical Imm	`srli s1,s2,imm`
Shift Right Arithmetic	`sra  s1,s2,s3`
Shift Right Arithmetic Imm	`srai s1,s2,imm`

- When using immediate, only values 0-31 are practical
- When using variable, only lowest 5 bits are used (read as unsigned)

Shifting Instructions: Example

```
# sample calls to shift instructions
addi  t0,x0 ,-256 # t0=0xFFFFFF00
slli  s0,t0,3     # s0=0xFFFFF800
srli  s1,t0,8     # s1=0x00FFFFFF
srai  s2,t0,8     # s2=0xFFFFFFFF

addi t1,x0 ,-22   # t1=0xFFFFFFEA
                  # low 5: 0b01010
sll s3,t0,t1      # s3=0xFFFC0000
# same as slli s3,t0,10
```

Shifting Instructions: Example

```
# lb using lw:  lbu s1,1(s0)
lw    s1,0(s0)  # get word
andi  s1,s1,0xFF00 # get 2nd byte
srli  s1,s1,8   # shift into lowest
```

Shifting Instructions: Example

```
# sb using sw:  sb s1,3(s0)
lw    t0,0(s0)  # get current word
andi  t0,t0,0xFFFFFF # zero top byte
slli  t1,s1,24  # shift into highest
or    t0,t0,t1  # combine
sw    t0,0(s0)  # store back
```

Recommended Reading

- RISC-V Edition - Computer Organization and Design_ The Hardware Software Interface - David A. Patterson, John L. Hennessy:
  - Chapter-2

Loops in RISCV

Loops in RISCV

- There are three types of loops in C:
  - `while`, `do…while`, and `for`
  - Each can be rewritten as either of the other two, so the same concepts of decision-making apply

- **Key Concept:** Though there are multiple ways to write a loop in RISCV, the key to decision-making is the conditional branch

C Loop Mapped to RISC-V Assembly

```
int A[20];
int sum = 0;
for (int i=0; i<20; i++)
    sum +=  A[i];
```

Assume:
Address of A[0] in x9,
sum in x10,
i in x11

```
add x9, x8, x0 # x9=&A[0]
add x10, x0, x0 # sum=0
add x11, x0, x0 # i=0
Loop:
  lw x12, 0(x9) # x12=A[i]
  add x10,x10,x12 # sum+=
  addi x9,x9,4  # &A[i++]
  addi x11,x11,1 # i++
  addi x13,x0,20 # x13=20
  blt x11,x13,Loop
```

C Loop Mapped to RISC-V Assembly

- C code:

```
while (save[i] == k) i += 1;
```

  - i in x22, k in x24, address of save in x25

- Compiled RISC-V code:

```
Loop: slli x10, x22, 3    #Temp reg x10=i*8
      add  x10, x10, x25  # x10 = &save[i]
      ld   x9, 0(x10)     # x9 = save[i]
      bne  x9, x24, Exit  # Exit save[i] ≠ k
      addi x22, x22, 1    # i = i + 1
      beq  x0, x0, Loop   # go to Loop
Exit: …
```

RISC-V Pseudo-Instructions

Assembler Pseudo-Instructions

- More intuitive for programmers, but NOT directly implemented in hardware
- Translated later into instructions which are directly implemented in hardware
- Example:
  `mv dst,reg1` translated into
  `addi dst,reg1,0` or `add dst,reg1,x0`

Full list of RISC-V supported pseudo instructions are on the greensheet.

More Pseudo-Instructions

- Load Immediate (`li`)
  - `li dst,imm`
  - Loads 32-bit immediate into `dst`
  - translates to: `addi dst x0 imm`

- Load Address (`la`)
  - `la dst,label`
  - Loads address of specified label into `dst`
  - (translation omitted => uses auipc)

J is a Pseudo-Instruction

- Even the `j` instruction is actually a pseudo-Instruction
  - We will see what this converts to later this lecture

- Pseudo-Instructions are core to writing RISC assembly code and you will see it in any RISC assembly code you read

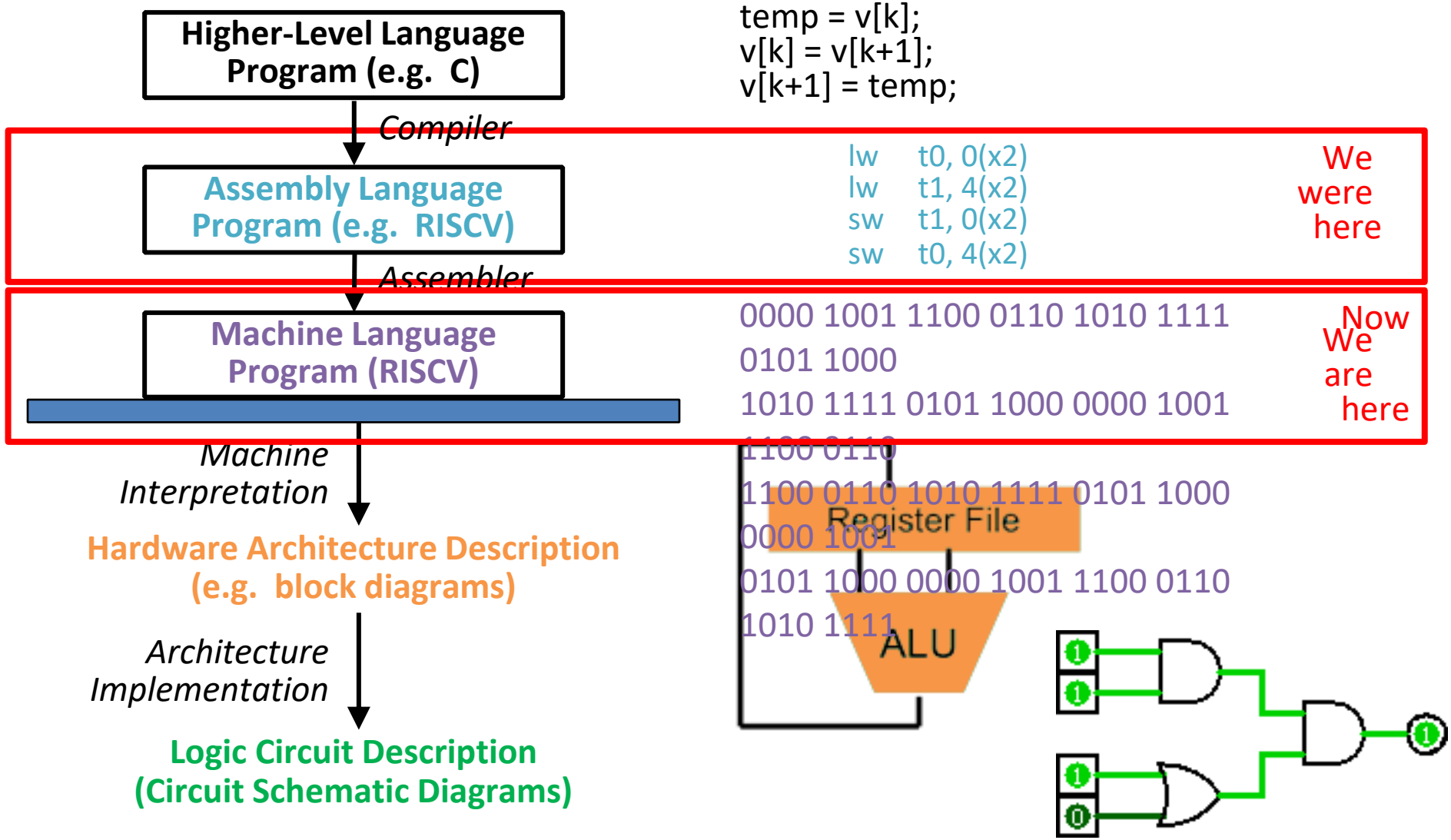
True Assembly vs RISC-V

- True Assembly Language
  - The instructions a computer understands and executes (directly implemented in hardware!)
- RISC-V Assembly Language
  - Instructions the assembly programmer can use (includes pseudo-instructions)
  - Each RISC-V Assembly instruction becomes 1 or more True Assembly instructions
- True Assembly $\subset$ RISC-V Assembly

Recommended Reading

- RISC-V Edition - Computer Organization and Design_ The Hardware Software Interface - David A. Patterson, John L. Hennessy:
  - Chapter-2

Where are we heading?



Agenda

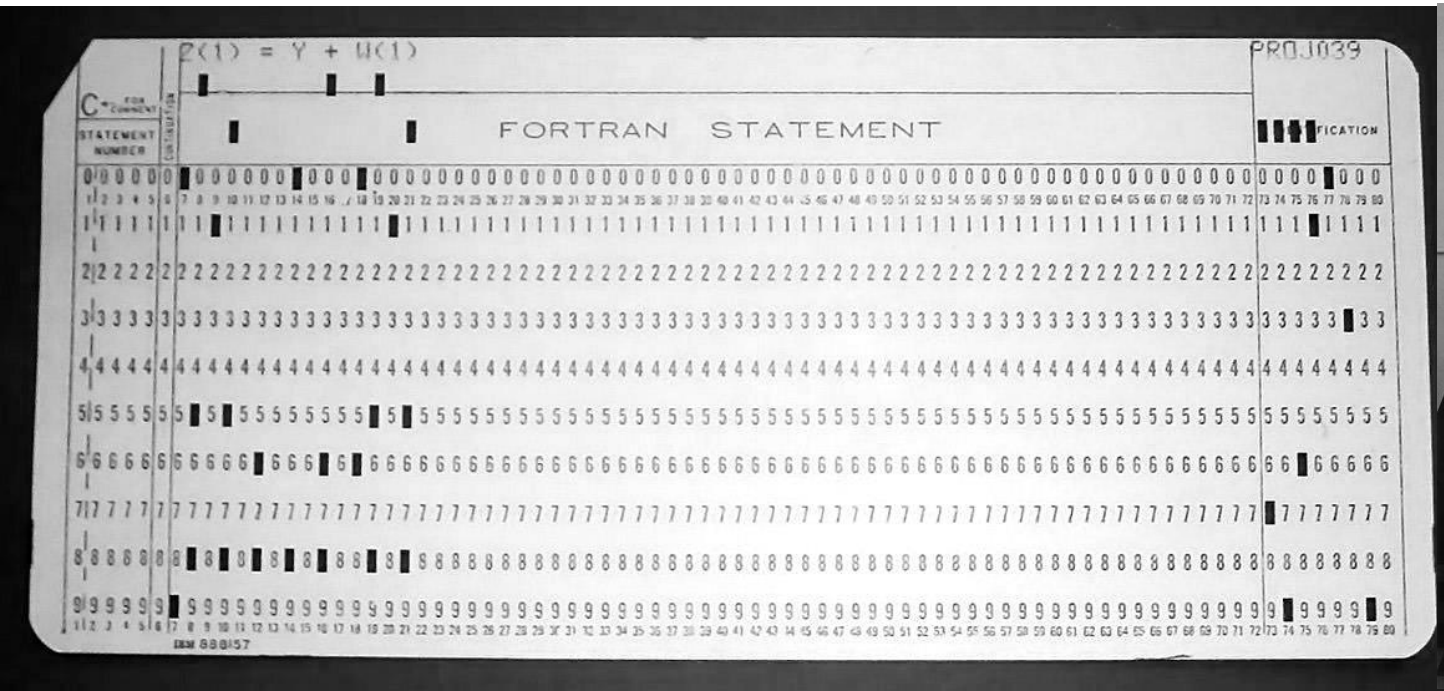
- Stored-Program Concept
- R-Format
- I-Format
- S-Format
- SB-Format
- U-Format
- UJ-Format

Big Idea: Stored-Program Concept

INSTRUCTIONS ARE DATA

- programs can be stored in memory as numbers
- Before: a number can mean anything
- **Now: make convention for interpreting numbers as instructions**

Introduction to Machine Language



Instructions as Numbers

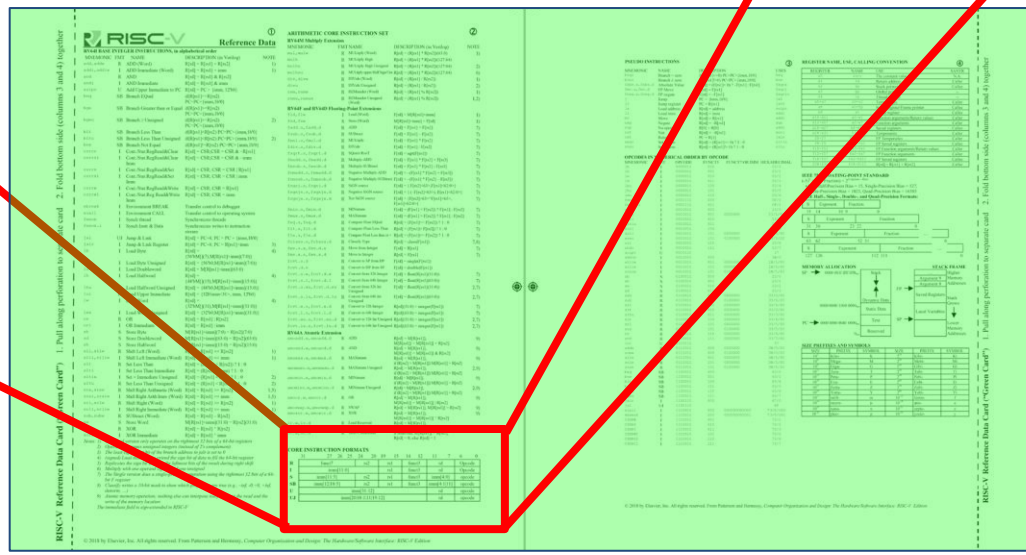
- By convention, RISCV instructions are each 1 word = 4 bytes = 32 bits

- Divide the 32 bits of an instruction into "fields"
  - regular field sizes → simpler hardware
  - will need some variation….
- Define 6 types of *instruction formats*
  - *See next slide*

51

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



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

Agenda

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

54

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.

55

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?

56

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**)

57

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

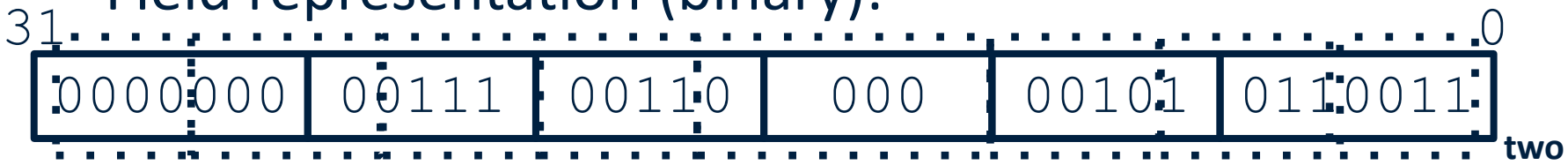
Different encoding in funct7 + funct3 selects different operations

R-Format Example

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

Field representation (decimal):

Field representation (binary):



hex representation: `0x 0073 02B3`

decimal representation: 7,537,331

Called a Machine Language Instruction

60

Agenda

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

61

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]`**

63

I-Format Instructions (2/4)

- `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} , $+2^{11}$)

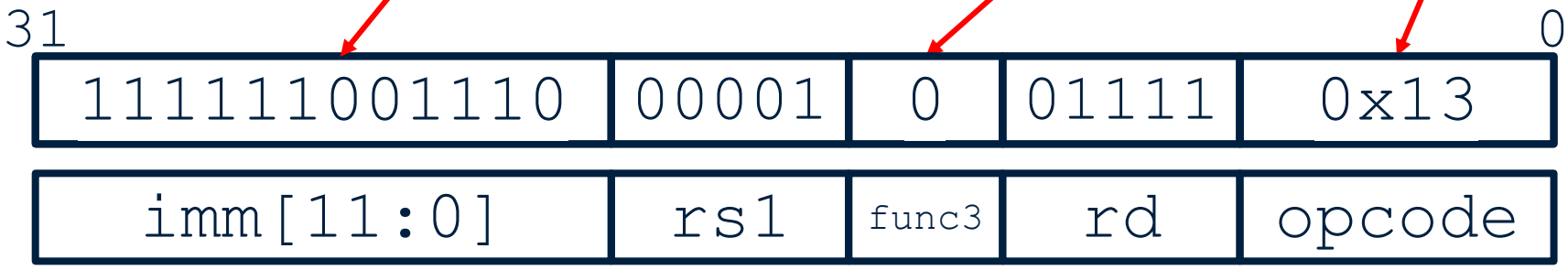
I-Format Example (1/2)

```
addi x15,x1,-50
```

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
```



66

I-Format Example (2/2)

- RISCV Instruction: `addi x15,x1,-50`

Field representation (binary):

hex representation: 0xFCE0 8793

decimal representation: 4,242,573,203

two

67

All RISCV I-Type Arithmatic

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)

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

Recommended Reading

- RISC-V Edition - Computer Organization and Design_ The Hardware Software Interface - David A. Patterson, John L. Hennessy:
  - Chapter-2:
    - 2.5

Clarification(s)

- Procedure call: jump and link

  `jal x1, ProcedureLabel`

  - Address of following instruction put in x1
  - Jumps to target address

- Procedure return: jump and link register

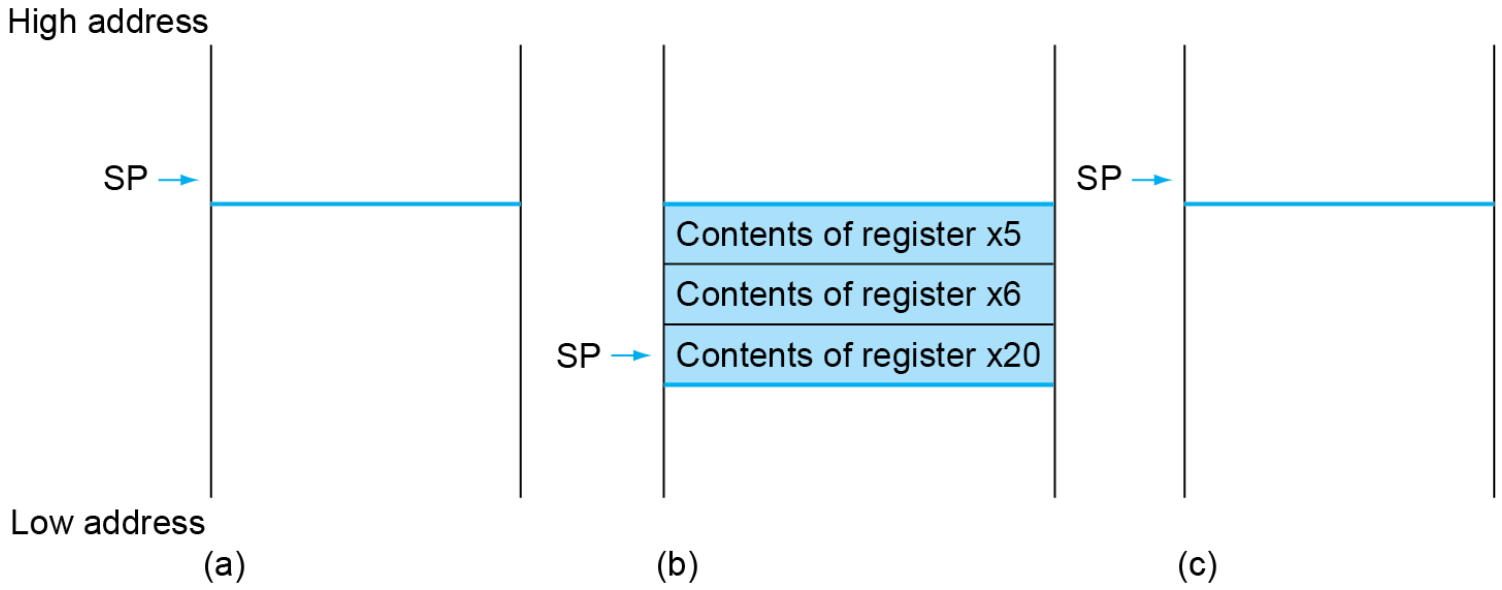
  `jalr x0, 0(x1)`

  - Like jal, but jumps to 0 + address in x1
  - Use x0 as rd (x0 cannot be changed)
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Clarification(s)

- PUSH = you push values onto the stack by subtracting from the stack pointer.
- POP = Adding to the stack pointer shrinks the stack, thereby popping values off the stack.

```
addi sp,sp,-24
sd   x5,16(sp)
sd   x6,8(sp)
sd   x20,0(sp
```



Acknowledgements

- The slides in this lecture contain materials and illustrations developed and copyright by:
  - Prof. David A. Patterson and Prof. John L. Hennessy [UC Berkely]
  - Prof. Onur Mutlu [ETH Zurich]
  - Steven Ho and Nick Riasanovsky[UC Berkely]
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - Lavanya Ramapantulu (NTU)

Agenda

- Stored-Program Concept
- R-Format
- I-Format (cont.)
- S-Format
- SB-Format
- U-Format
- UJ-Format

73

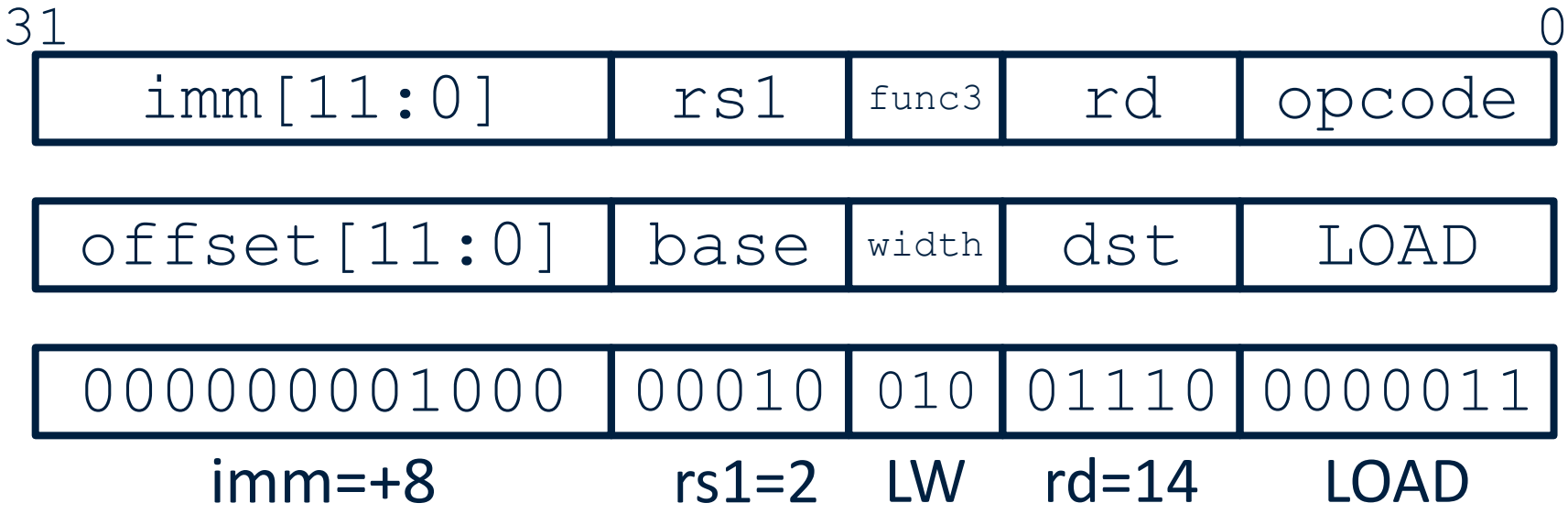
Load Instructions are also I-Type

- 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 address, not to create final result*
- Value loaded from memory is stored in `rd`

74

I-Format Load Example

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



75

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

76

Agenda

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

77

S-Format Used for Stores

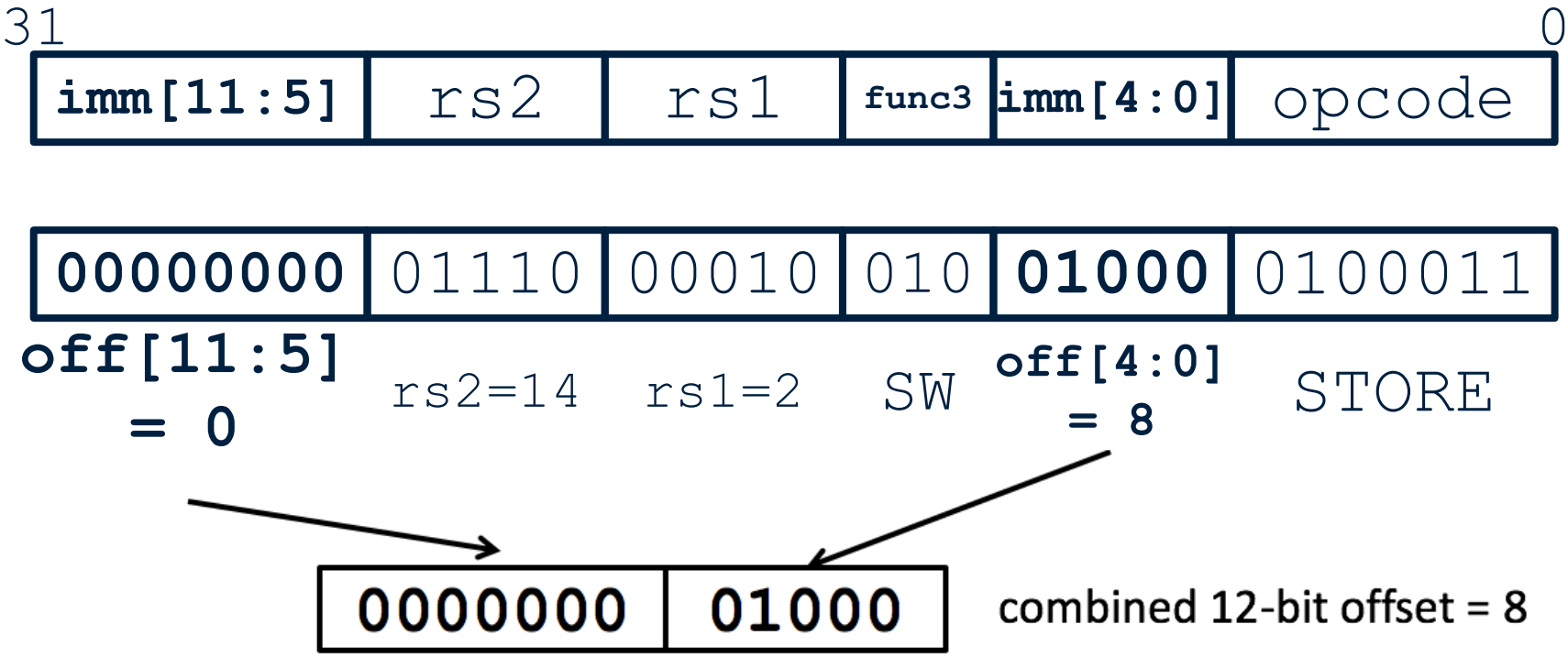
- Store needs to read two registers, `rs1` for base memory address, and `rs2` for data to be stored, as well as need immediate offset!
  - Note: stores don't write a value to the register file, no `rd`!

- 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
    - Keep formats as similar as possible

78

S-Format Example

sw x14, 8(x2)



79

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

Recommended Reading

- RISC-V Edition - Computer Organization and Design_ The Hardware Software Interface - David A. Patterson, John L. Hennessy:
  - Chapter-2:
    - 2.5