Computer Architecture 1

Computer Organization and Design

THE HARDWARE/SOFTWARE INTERFACE

[Adapted from Computer Organization and Design, RISC-V Edition, Patterson & Hennessy, © 2018, MK]
[Adapted from Great ideas in Computer Architecture (CS 61C) lecture slides, Garcia and Nikolíc, © 2020, UC Berkeley]

RV32 So Far…

- Add/sub

```
add rd, rs1, rs2
sub rd, rs1, rs2
```

- Add immediate

```
addi rd, rs1, imm
```

- Load/store

```
lw  lb
lbu rd,
sw  rs1, rs2, imm
sb  rs1, rs2, imm
```

- Branching

```
beq rs1, rs2, Label
bne rs1, rs2, Label
bge rs1, rs2, Label
blt rs1, rs2, Label
bgeu rs1, rs2, Label
bltu rs1, rs2, Label
j Label
```

RISC-V Logical Instructions

- Useful to operate on fields of bits within a word
  - e.g., characters within a word (8 bits)
- Operations to pack /unpack bits into words
- Called logical operations

Logical operations	C operators	Java operators	RISC-V instructions
Bit-by-bit AND	&	&	**`and`**
Bit-by-bit OR	|	|	**`or`**
Bit-by-bit XOR	^	^	**`xor`**
Shift left logical	<<	<<	**`sll`**
Shift right logical	>>	>>	**`srl`**

RISC-V

RISC-V Logical Instructions

- Always two variants
  - Register: **`and x5, x6, x7 # x5 = x6 & x7`**
  - Immediate: **`andi x5, x6, 3 # x5 = x6 & 3`**
- Used for ‘masks’
  - **`andi`** with **`0000 00FF`**$_{\text{hex}}$ isolates the least significant byte
  - **`andi`** with **`FF00 0000`**$_{\text{hex}}$ isolates the most significant byte
- There is no logical NOT in RISC-V
  - Use **`xor`** with **`11111111`**$_{\text{two}}$
  - Remember - simplicity…

Logical Shifting

- Shift Left Logical (**`sll`**) and immediate (**`slli`**):

  `slli x11,x12,2 #x11=x12<<2`

  - Store in **x11** the value from **x12** shifted by 2 bits to the left (they fall off end), inserting 0's on right; << in C.
  - Before: **`0000 0002`**$_{\texttt{hex}}$

    `0000 0000 0000 0000 0000 0000 0000 0010`$_{\texttt{two}}$
  - After: **`0000 0008`**$_{\texttt{hex}}$

    `0000 0000 0000 0000 0000 0000 0000 1000`$_{\texttt{two}}$
  - What arithmetic effect does shift left have?
- Shift Right: **`srl`** is opposite **`shift; >>`**

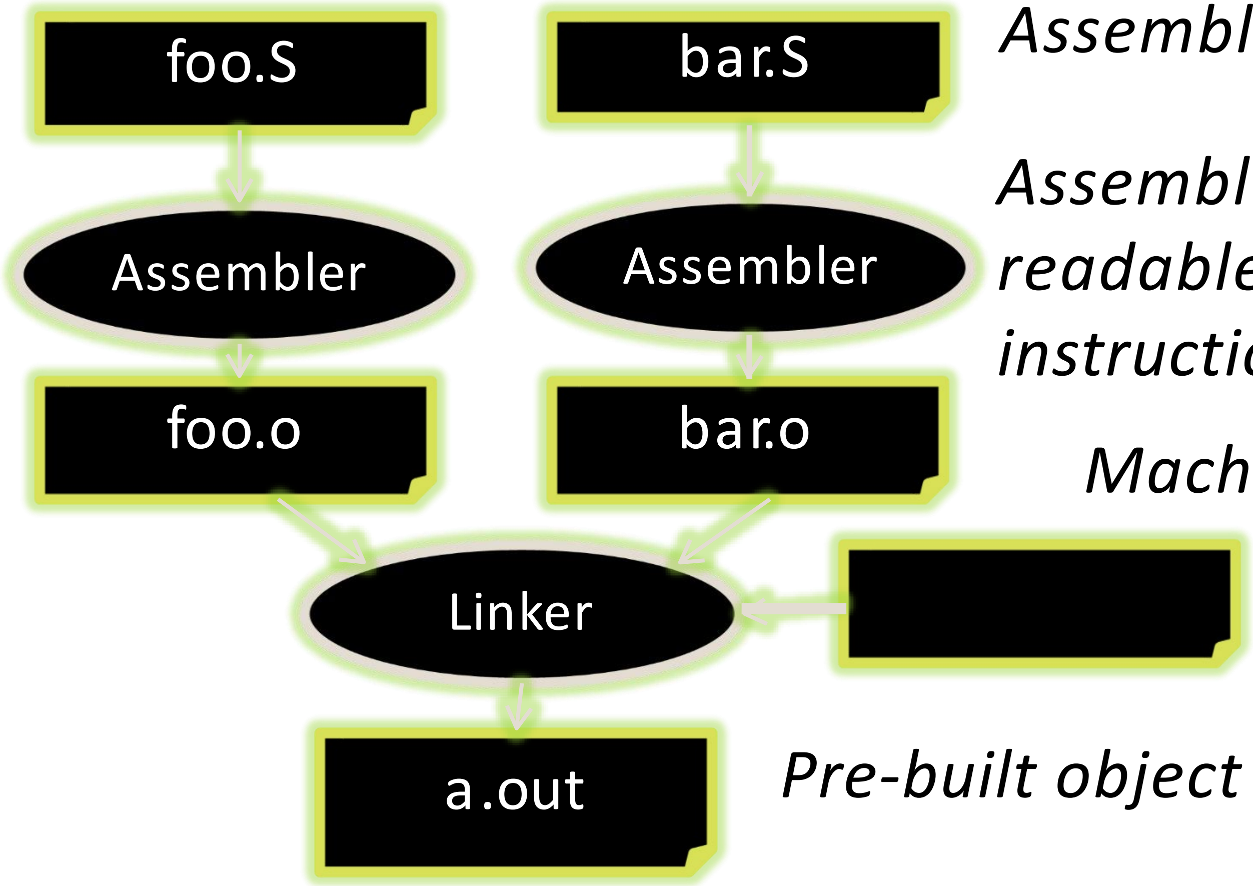
Arithmetic Shifting

- Shift right arithmetic (**`sra`**, **`srai`**) moves *n* bits to the right (insert high-order sign bit into empty bits)
- For example, if register x10 contained

  `1111 1111 1111 1111 1111 1111 1110 0111`$_{two}$= **-25**$_{ten}$
- If execute **`srai x10, x10, 4`**, result is:

  `1111 1111 1111 1111 1111 1111 1111 1110`$_{two}$= **-2**$_{ten}$
- Unfortunately, this is NOT same as dividing by 2^n
  - Fails for odd negative numbers
  - C arithmetic semantics is that division should round towards 0

Assembler to Machine Code (More Later in Course)



Assembler source files (text)

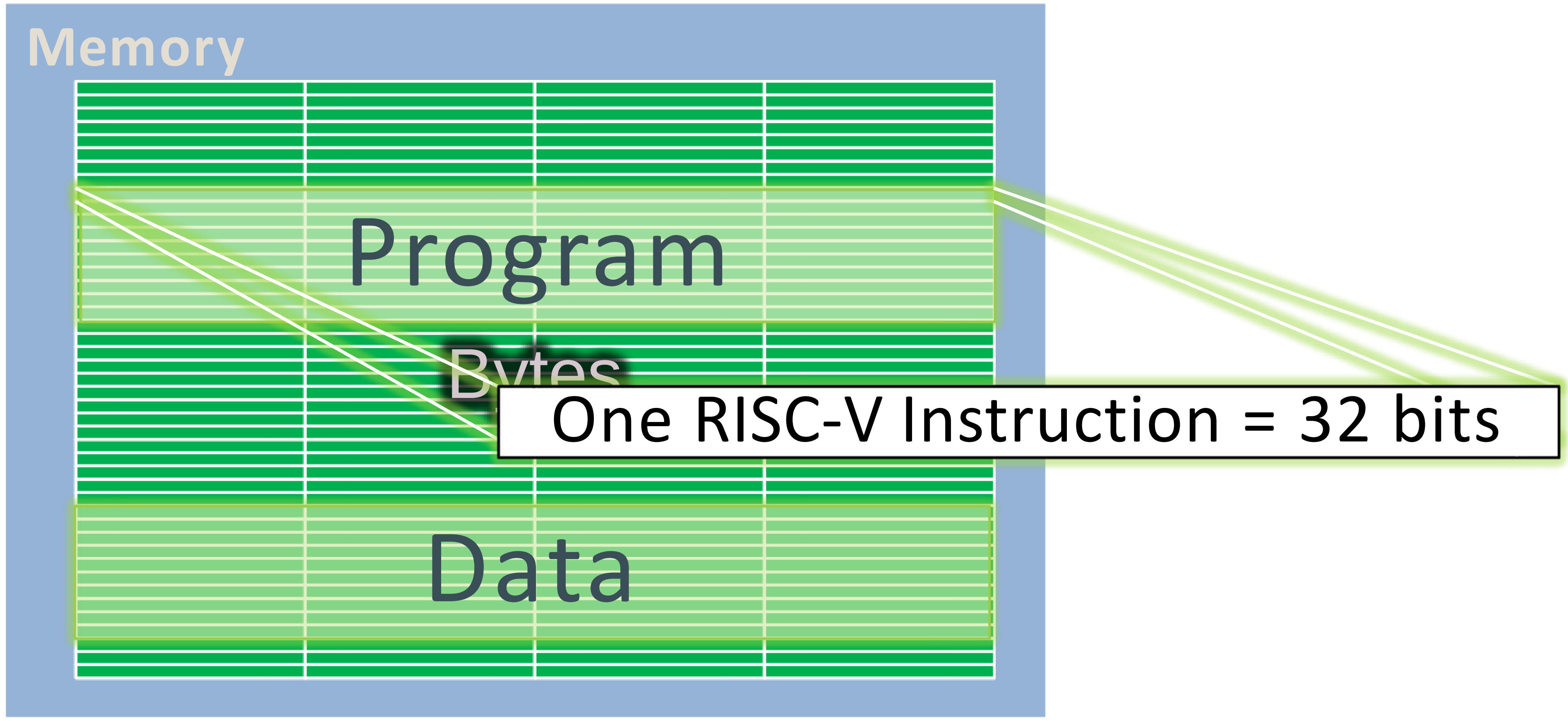
Assembler converts human-readable assembly code to instruction bit patterns

Machine code object files

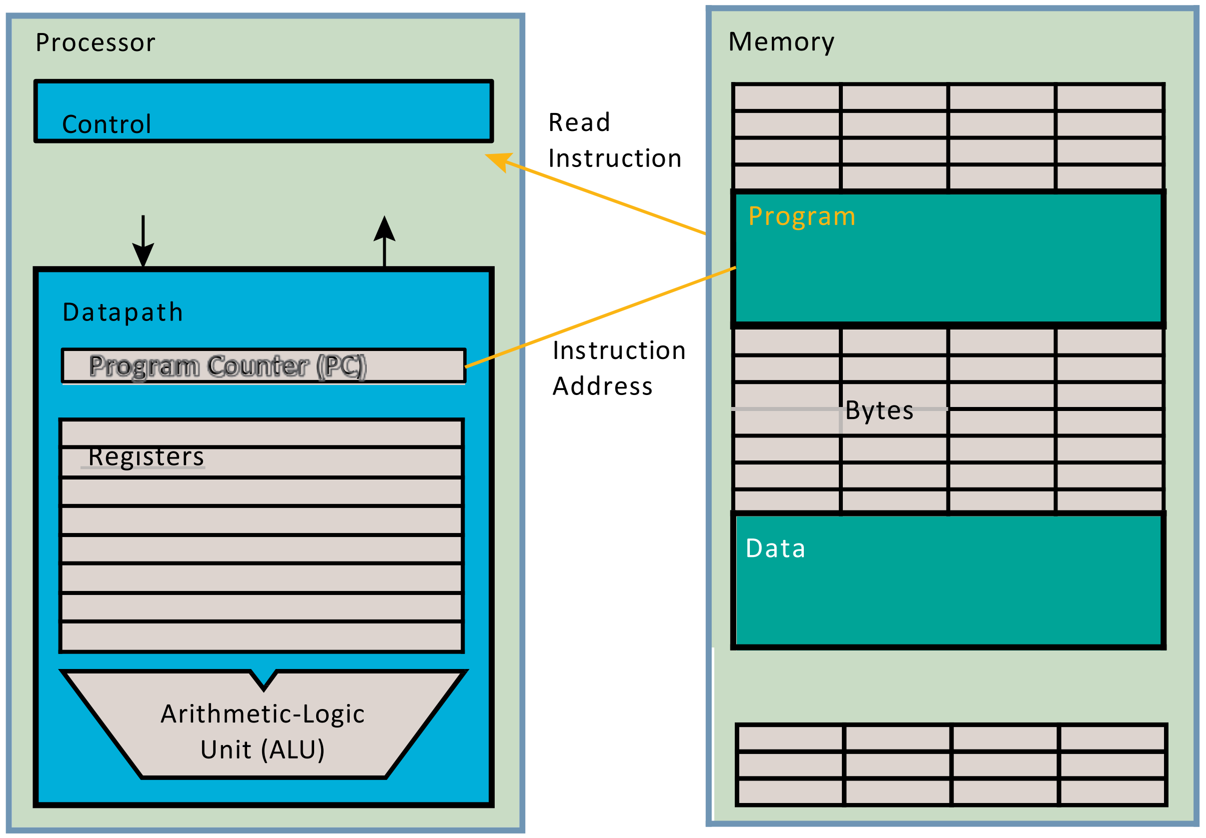
Pre-built object file libraries

Machine code executable file

How Program is Stored



Program Execution



- **PC** (program counter) is a register internal to the processor that holds byte address of next instruction to be executed

Helpful RISC-V Assembler Features

- Symbolic register names
  - E.g., **`a0-a7`** for argument registers (**`x10-x17`**) for function calls
  - E.g., **`zero`** for **`x0`**
- Pseudo-instructions
- Shorthand syntax for common assembly idioms
- E.g., **`mv rd, rs`** = **`addi rd, rs, 0`**
- E.g., **`li rd, 13`** = **`addi rd, x0, 13`**
- E.g., **`nop`** = **`addi x0, x0, 0`**

RISC V Function calls

```
main() {
   int i,j,k,m;
   ...
   i = mult(j,k); ...
   m = mult(i,i); ...

  /* really dumb mult function */

  int mult (int mcand, int mlier){
     int product = 0;
     while (mlier > 0)  {
       product = product + mcand;
       mlier = mlier -1; }
     return product;
     }
```

What information mus compiler/programmer keep track of?

What instructions can accomplish this?

Six Fundamental Steps in Calling a Function

1. Put arguments in a place where function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put return value in a place where calling code can access it and restore any registers you used; release local storage
6. Return control to point of origin, since a function can be called from several points in a program

RISC-V Function Call Conventions

- Registers faster than memory, so use them
- **`a0–a7`** (**`x10-x17`**): eight *argument* registers to pass parameters and two return values (**`a0–a1`**)
- **`ra`**: one *return address* register to return to the point of origin (**`x1`**)
- Also **`s0-s1`** (**`x8-x9`**) and **`s2-s11`** (**`x18-x27`**): saved registers (more about those later)

Instruction Support for Functions (1/4)

C

```
... sum(a,b);... /* a,b:s0,s1 */
     }
          int sum(int x, int y) {
          return x+y;
     }
```

RISC-V

```
address (shown in decimal)
     1000
     1004
     1008
     1012
     1016
     …
     2000
     2004
```

In RISC-V, all instructions are 4 bytes, and stored in memory just like data. So, here we show the addresses of where the programs are stored.

Instruction Support for Functions (2/4)

C

```
... sum(a,b);... /* a,b:s0,s1 */
    }
          int sum(int x, int y) {
          return x+y;
    }
```

RISC-V

```
address (shown in decimal)
    1000 mv a0,s0                 # x = a
    1004 mv a1,s1                 # y = b
    1008 addi ra,zero,1016        #ra=1016
    1012 j    sum                #jump to sum
    1016 …                       # next inst.
    …
    2000 sum: add a0,a0,a1
    2004 jr  ra #new instr."jump reg"
```

Instruction Support for Functions (3/4)

C

```
... sum(a,b);... /* a,b:s0,s1 */
    }
          int sum(int x, int y) {
          return x+y;
    }
```

RISC-V

- Question: Why use **`jr`** here? Why not use **`j`**?
- Answer: **`sum`** might be called by many places, so we can't return to a fixed place. The calling proc to **`sum`** must be able to say "return here" somehow.

```
…
2000 sum: add a0,a0,a1
2004 jr   ra #new instr."jump reg"
```

Instruction Support for Functions (4/4)

- Single instruction to jump and save return address: jump and link (**`jal`**)
- Before:
  - 1008 addi ra,zero,1016   *`# ra=1016`*
  - 1012 j sum   *`# goto sum`*
- After

```
1008 jal sum  # ra=1012,goto sums
```

- Why have a **`jal`**?
  - Make the common case fast: function calls very common
  - Reduce program size
  - Don't have to know where code is in memory with **`jal`**!

RISC-V Function Call Instructions

- Invoke function*: jump and link* instruction (**`jal`**) (really should be **`laj`** *"link and jump"*)
  - "link" means form an *address* or *link* that points to calling site to allow function to return to proper address
  - Jumps to address and simultaneously saves the address of the following instruction in register ra

```
jal FunctionLabel
```

- Return from function: *jump register* instruction (**`jr`**)
  - Unconditional jump to address specified in register: **`jr ra`**
  - Assembler shorthand: **`ret = jr ra`**

Summary of Instruction Support

Actually, only two instructions:

- `jal   rd, Label` – jump-and-link
- `jalr  rd, rs, imm` – jump-and-link register

`j, jr` and `ret` are pseudoinstructions!

- `j:   jal  x0, Label`

RISC-V Instruction Representation

High Level Language
Program (e.g., C)

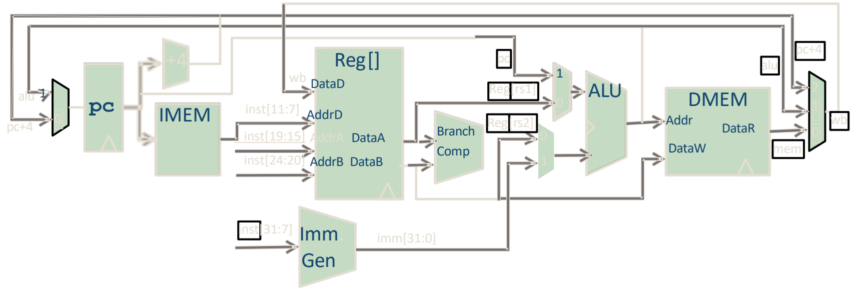
Assembly Language
Program (e.g., RISC-V)

```
lw    x3, 0(x10)
lw    x4, 4(x10)
sw    x4, 0(x10)
sw    x3, 4(x10)
```

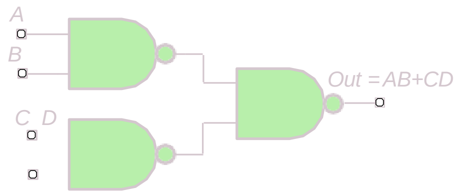
Anything can be represented
as a number,
i.e., data or instructions

Machine Language
Program (RISC-V)

Hardware Architecture Description
(e.g., block diagrams)



Logic Circuit Description
(Circuit Schematic Diagrams)



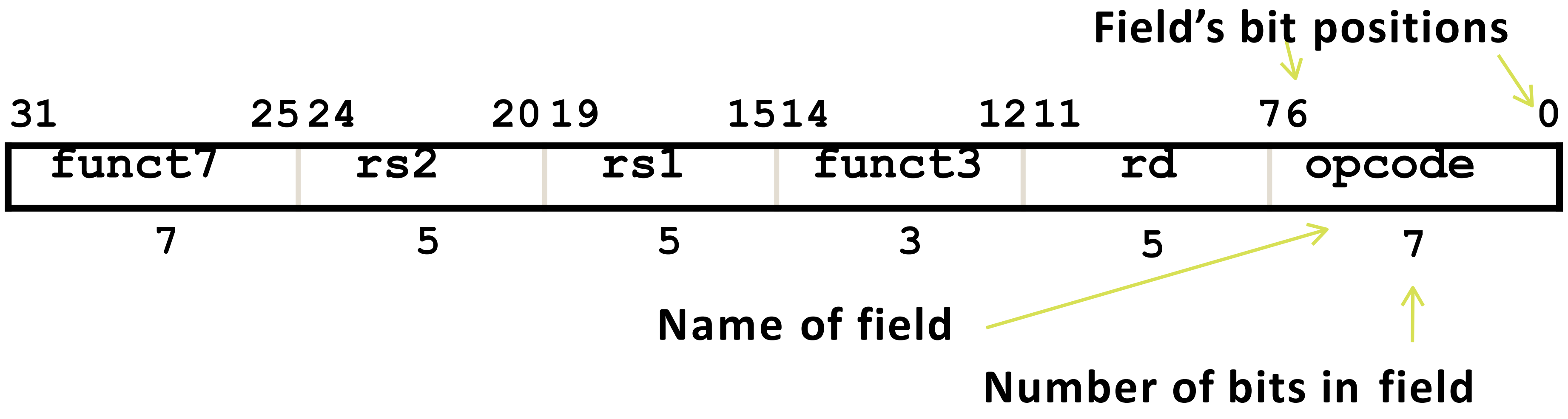
Instructions as Numbers (1/2)

- Most data we work with is in words (32-bit chunks):
  - Each register is a word
  - **`lw`** and **`sw`** both access memory one word at a time
- So how do we represent instructions?
  - Remember: Computer only understands 1s and 0s, so assembler string "**`add x10,x11,x0`**" is meaningless to hardware
  - RISC-V seeks simplicity: since data is in words, make instructions be fixed-size 32-bit words also
    - Same 32-bit instructions used for RV32, RV64, RV128

Instructions as Numbers (2/2)

- One word is 32 bits, so divide instruction word into "fields"
- Each field tells processor something about instruction
- We could define different fields for each instruction, but RISC-V seeks simplicity, so define six basic types of instruction formats:
  - R-format for register-register arithmetic operations
  - I-format for register-immediate arithmetic operations and loads
  - S-format for stores
  - B-format for branches (minor variant of S-format)
  - U-format for 20-bit upper immediate instructions
  - J-format for jumps (minor variant of U-format)

R-Format Instruction Layout



- 32-bit instruction word divided into six fields of varying numbers of bits each: 7+5+5+3+5+7 = 32
- Examples
  - `opcode` is a 7-bit field that lives in bits 6-0 of the instruction
  - `rs2` is a 5-bit field that lives in bits 24-20 of the instruction

R-Format Instructions opcode/funct Fields

31 25	24 20	19 15	14 12	11 7	6 0
`funct7`	`rs2`	`rs1`	`funct3`	`rd`	`opcode`
7	5	5	3	5	7

opcode: partially specifies what instruction it is

- Note: This field is equal to **`0110011`**$_{\texttt{two}}$ for all R-Format register-register arithmetic instructions

`funct7+funct3`: combined with **`opcode`**, these two fields describe what operation to perform

- **Question: You have been professing simplicity, so why aren't `opcode` and `funct7` and `funct3` a single 17- bit field?**
  - **We'll answer this later**

R-Format Instructions Register Specifiers

31 25	24 20	19 15	14 12	11 7	6 0
`funct7`	`rs2`	`rs1`	`funct3`	`rd`	`opcode`
7	5	5	3	5	7

`rs1` (Source Register #1): specifies register containing first operand

`rs2` : specifies second register operand

`rd` (Destination Register): specifies register which will receive result of computation

Each register field holds a 5-bit unsigned integer (0-31) corresponding to a register number (**`x0`**-**`x31`**)

R-Format Example

- RISC-V Assembly Instruction:

```
add   x18,x19,x10
```

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7

31 25	24 20	19 15	14 12	11 7	6 0
0000000	01010	10011	000	10010	0110011
7	5	5	3	5	7
add	rs2=10	rs1=19	add	rd=18	Reg-Reg OP

Your Turn

- What is correct encoding of **`add x4, x3, x2`** ?

1) 4021 8233$_{hex}$
2) 0021 82b3$_{hex}$
3) 4021 82b3$_{hex}$
4) 0021 8233$_{hex}$
5) 0021 8234$_{hex}$

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	110	rd	0110011	or
						and

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
Can you spot two new instructions?

I-Format Instructions

- What about instructions with immediates?
  - Compare:
    - `add  rd, rs1, rs2`
    - `addi rd, rs1, imm`
  - 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
  - Ideally, RISC-V would have only one instruction format (for simplicity): unfortunately, we need to compromise
- Define new instruction format that is mostly consistent with R-format
  - Notice if instruction has immediate, then uses at most 2 registers (one source, one destination)

I-Format Instruction Layout

31 25 24 20	19 15	14 12	11 7	6 0
`imm[11:0]`	`rs1`	`funct3`	`rd`	`opcode`
12	5	3	5	7

- Only one field is different from R-format, **`rs2`** and **`funct7`** replaced by 12-bit signed immediate, **`imm[11:0]`**
- Remaining fields (**`rs1`**, **`funct3`**, **`rd`**, **`opcode`**) same as before
- **`imm[11:0]`** can hold values in range [-2048$_{\text{ten}}$, +2047$_{\text{ten}}$]
- Immediate is always sign-extended to 32-bits before use in an arithmetic operation
- We'll later see how to handle immediates > 12 bits

I-Format Example

- RISC-V Assembly Instruction:

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

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7

111111001110	00001	000	01111	0010011
imm=-50	rs1=1	add	rd=15	OP-Imm

All RV32 I-format 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)

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

Load Instructions are also I-Type

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD

- 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
- The value loaded from memory is stored in register `rd`

I-Format Load Example

- RISC-V Assembly Instruction:

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

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD

000000001000	00010	010	01110	0000011
imm=+8	rs1=2	lw	rd=14	LOAD
		(load word)		

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

S-Format Used for Stores

31 25	24 20	19 15	14 12	11 7	6 0
`Imm[11:5]`	`rs2`	`rs1`	`funct3`	`imm[4:0]`	`opcode`
7	5	5	3	5	7
`offset[11:5]`	`src`	`base`	`width`	`offset[4:0]`	`STORE`

- Store needs to read two registers, **`rs1`** for base memory address, and **`rs2`** for data to be stored, as well immediate offset!
- Can’t have both **`rs2`** and immediate in same place as other instructions!
- Note that stores don’t write a value to the register file, ***no `rd`***!
- RISC-V design decision is to 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

- RISC-V Assembly Instruction:

```
sw x14, 8(x2)
```

31 25	24 20	19 15	14 12	11 7	6 0
Imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE
0000000	01110	00010	010	01000	0100011
offset[11:5] =0	rs2=14	rs1=2	SW	offset[4:0] =8	STORE

0000000 01000 combined 12-bit offset = 8

All RV32 Store Instructions

- Store byte, halfword, word

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
			width			

RISC-V Conditional Branches

- E.g., **`beq x1, x2, Label`**
- Branches read two registers but don’t write to 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)
  - Function calls and unconditional jumps handled with jump instructions (J-Format)
- **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 offset to PC

- Branches generally change the PC by a small amount
- Can specify ± 2^{11} 'unit' addresses from the PC
- (We will see in a bit that we can encode 12-bit offsets as immediates)

- Why not use byte as a unit of offset from PC?
  - Because instructions are 32-bits (4-bytes)
  - We don't branch into middle of instruction

Scaling Branch Offset

- One idea: To improve the reach of a single branch instruction, multiply the offset by four bytes before adding to PC
- This would allow one branch instruction to reach $\pm\ 2^{11} \times$ 32-bit instructions either side of PC
  - Four times greater reach than using byte offset

Branch Calculation

- If we don’t take the branch:

  `PC = PC + 4` (i.e., next instruction)

- If we do take the branch:

  `PC = PC + immediate*4`

- Observations:
  - **`immediate`** is number of instructions to jump (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
- To enable this, RISC-V scales the branch offset by 2 bytes 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 on either side of PC

RISC-V B-Format for Branches

31	30 25	24 20	19 15	14 12	11 8	7	6 0
`imm[12]`	`imm[10:5]`	`rs2`	`rs1`	`funct3`	`imm[4:1]`	`imm[11]`	`opcode`
1	**6**	**5**	**5**	**3**	**4**	**1**	**7**

- B-format is mostly same as S-Format, with two register sources (**`rs1`**/**`rs2`**) and a 12-bit immediate **`imm[12:1]`**
- But now immediate represents values

  -4096 to +4094 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, Determine Offset

- RISC-V Code:

```
Loop: beq  x19,x10,End
      add  x18,x18,x10
      addi x19,x19,-1
      j    Loop
End:  # target instruction
```

Count instructions from branch

1

2

3

4

- Branch offset =

4 × 32-bit instructions = 16 bytes

- (Branch with offset of 0, branches to itself)

Branch Example, Determine Offset

- RISC-V Code:

Count instructions from branch

```
Loop: beq   x19,x10,End
      add   x18,x18,x10     1
      addi  x19,x19,-1      2
      j     Loop            3
End:  # target instruction  4
```

???????	01010	10011	000	?????	1100011
imm	rs2=10	rs1=19	BEQ	imm	BRANCH

Branch Example, Determine Offset

- RISC-V Code:

Offset = 16 bytes
= 8 x 2

```
Loop: beq  x19,x10,End
      add  x18,x18,x10      1
      addi x19,x19,-1       2
      j    Loop             3
End:  # target instruction  4
```

				01000	
???????	01010	10011	000	?????	1100011
imm	rs2=10	rs1=19	BEQ	imm	BRANCH

RISC-V Immediate Encoding

Instruction encodings, inst[31:0]

31 30 25	24 20	19 15	14 12	11 8 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type

32-bit immediates produced, imm[31:0]

31 25 24 12	11	10 5	4 1	0	
-inst[31]-		inst[30:25]	inst[24:21]	inst[20]	I-imm.
-inst[31]-		inst[30:25]	inst[11:8]	inst[7]	S-imm.
-inst[31]-	inst[7]	inst[30:25]	inst[11:8]	0	B-imm.

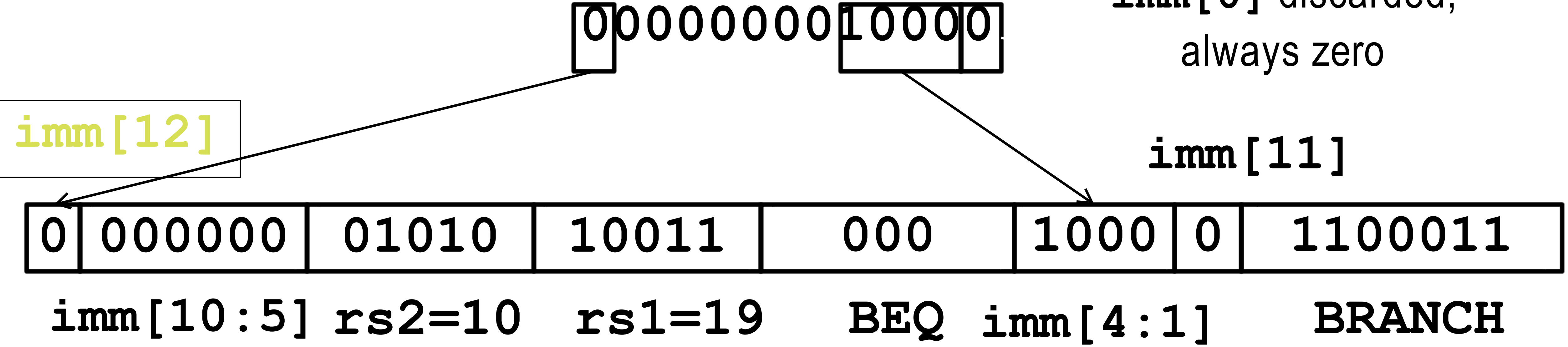
Upper bits sign-extended from **inst[31]** always

Only bit 7 of instruction changes role in immediate between S and B

Branch Example, Complete Encoding

`beq   x19,x10, offset = 16 bytes`

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



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 (‘position-independent code’)
- What do we do if destination is > 2^{10} instructions away from branch?
  - Other instructions save us

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 (‘position-independent code’)
- What do we do if destination is > 2^{10} instructions away from branch?
  - Other instructions save us

```
 beq x10,x0,far

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

U-Format for “Upper Immediate” Instructions

31 12	11 7	6 0
`imm[31:12]`	`rd`	`opcode`
20	5	7
`U-immediate[31:12]`	`dest`	`LUI`
`U-immediate[31:12]`	`dest`	`AUIPC`

- 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
```

One 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 is set, 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 in x10
```

J-Format for Jump Instructions

31	30 21	20	19 12	11 7	6 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7
	offset[20:1]			dest	JAL

- **`jal`** saves PC+4 in register **`rd`** (the return address)
  - Assembler "**`j`**" jump is pseudo-instruction, uses JAL but sets **`rd=x0`** to discard 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
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

Uses of JAL

```
# j pseudo-instruction
j Label = jal x0, Label # Discard return
address
```

```
# Call function within 2^18 instructions
of PC
jal ra, FuncName
```

JALR Instruction (I-Format)

31 20	19 15	14 12	11 7	6 0
`imm[11:0]`	`rs1`	`func3`	`rd`	`opcode`
12	5	3	5	7
`offset[11:0]`	`base`	`0`	`dest`	`JALR`

- **`jalr rd, rs, immediate`**
  - Writes PC+4 to rd (return address)
  - Sets PC = **`rs + immediate`**
  - Uses same immediates as arithmetic and loads
    - ***no*** multiplication by 2 bytes
    - In contrast to branches and **`jal`**

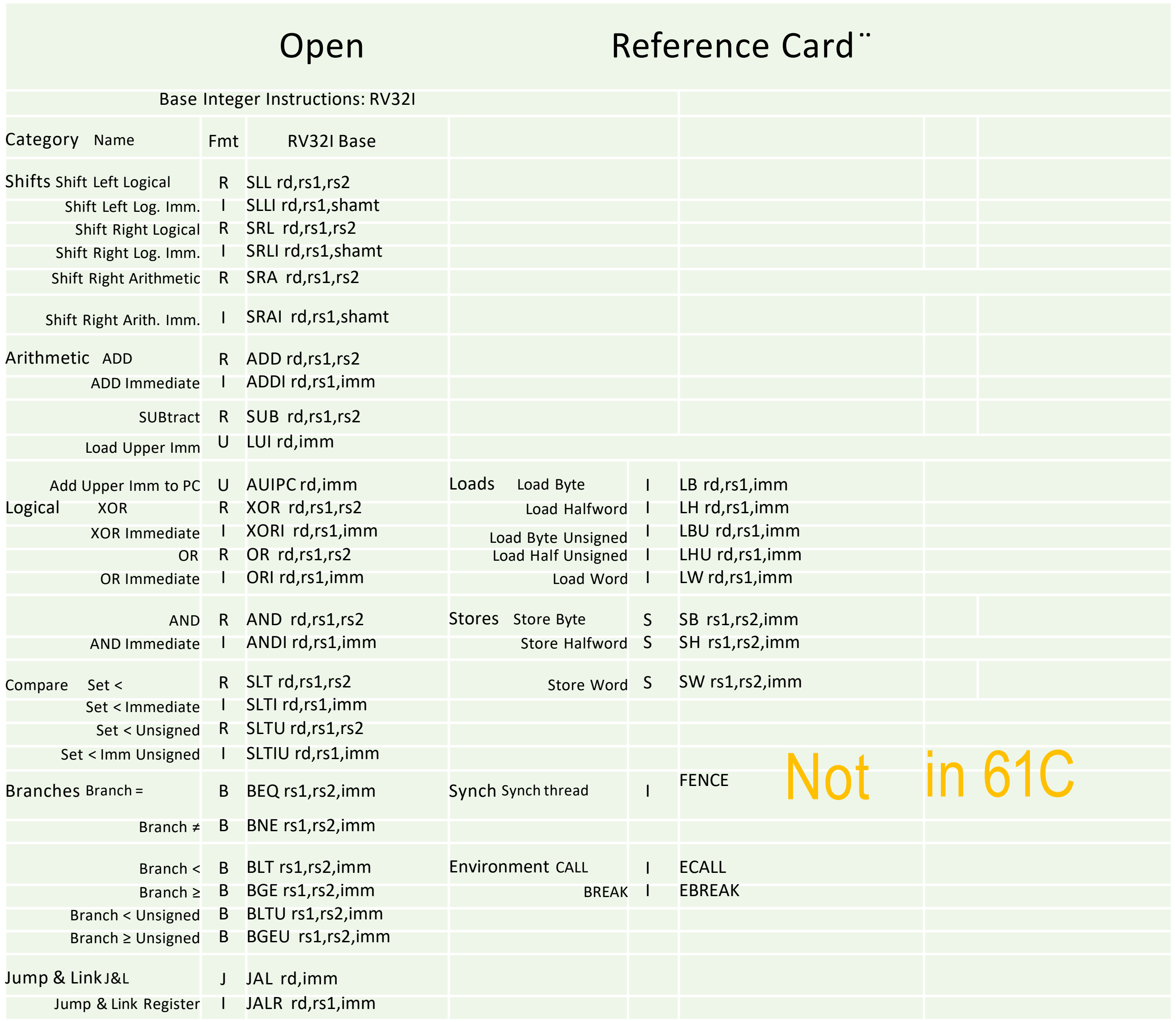
Uses of JALR

```
# ret and jr psuedo-instructions
ret = jr ra = jalr x0, ra, 0
# Call function at any 32-bit absolute
address
lui x1, <hi20bits>
jalr ra, x1, <lo12bits>
# Jump PC-relative with 32-bit offset
auipc x1, <hi20bits>
jalr x0, x1, <lo12bits>
```

Summary of RISC-V Instruction Formats

31 30 25	24 21 20	19 15	14 12	11 8 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3 im	m[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11]]		imm[19:12]		rd	opcode	J-type

Complete RV32I ISA



Open Reference Card

Base Integer Instructions: RV32I

Category	Name	Fmt	RV32I Base	Category	Name	Fmt	RV32I Base
Shifts	Shift Left Logical	R	SLL rd,rs1,rs2				
	Shift Left Log. Imm.	I	SLLI rd,rs1,shamt				
	Shift Right Logical	R	SRL rd,rs1,rs2				
	Shift Right Log. Imm.	I	SRLI rd,rs1,shamt				
	Shift Right Arithmetic	R	SRA rd,rs1,rs2				
	Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt				
Arithmetic	ADD	R	ADD rd,rs1,rs2				
	ADD Immediate	I	ADDI rd,rs1,imm				
	SUBtract	R	SUB rd,rs1,rs2				
	Load Upper Imm	U	LUI rd,imm				
	Add Upper Imm to PC	U	AUIPC rd,imm	Loads	Load Byte	I	LB rd,rs1,imm
Logical	XOR	R	XOR rd,rs1,rs2		Load Halfword	I	LH rd,rs1,imm
	XOR Immediate	I	XORI rd,rs1,imm		Load Byte Unsigned	I	LBU rd,rs1,imm
	OR	R	OR rd,rs1,rs2		Load Half Unsigned	I	LHU rd,rs1,imm
	OR Immediate	I	ORI rd,rs1,imm		Load Word	I	LW rd,rs1,imm
	AND	R	AND rd,rs1,rs2	Stores	Store Byte	S	SB rs1,rs2,imm
	AND Immediate	I	ANDI rd,rs1,imm		Store Halfword	S	SH rs1,rs2,imm
Compare	Set <	R	SLT rd,rs1,rs2		Store Word	S	SW rs1,rs2,imm
	Set < Immediate	I	SLTI rd,rs1,imm				
	Set < Unsigned	R	SLTU rd,rs1,rs2				
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm				
Branches	Branch =	B	BEQ rs1,rs2,imm	Synch	Synch thread	I	FENCE
	Branch ≠	B	BNE rs1,rs2,imm				
	Branch <	B	BLT rs1,rs2,imm	Environment	CALL	I	ECALL
	Branch ≥	B	BGE rs1,rs2,imm		BREAK	I	EBREAK
	Branch < Unsigned	B	BLTU rs1,rs2,imm				
	Branch ≥ Unsigned	B	BGEU rs1,rs2,imm				
Jump & Link	J&L	J	JAL rd,imm				
	Jump & Link Register	I	JALR rd,rs1,imm				

Not in 61C