Course Info

- Lab 3 is released, get yourself prepared before going to lab sessions!
- Project 1.1 available, and will be marked in lab sessions. Deadline March 13th.
- Will have HW3 this week.
- Discussion on RISC-V related materials and assembly coding.

信息科学与技术学院
School of Information Science and Technology

CS 110
Computer Architecture
Intro to RISC-V III

Instructors:

Siting Liu & Chundong Wang

Course website: https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2023/index.html

School of Information Science and Technology (SIST)

ShanghaiTech University

2023/2/6

Computer Decision Making—Branch

- Normal operation: execute instructions in sequence
- In programming languages: `if/while/for`-statement
- RISV-V provides conditional branch & unconditional jump

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type

- RISC-V: `if`-statement instructions are

  `beq/bne/blt/bltu/bge/bgeu rs1, rs2, L(imm/label)`

  Go to statement labeled L1 if (value in `rs1`) =/≠/<</≥ (value in `rs2`) using singed/unsigned comparison; otherwise, go to next statement

Translate Assembly

PC reg.

```
→ addi    x10, x0, 0x7                x10 = 7
  add     x12, x0, x0                 x12 = 0
  label_a: andi   x14, x10, 1         label_a: x14 = x10 & 1
  beq     x14, x0, label_b            if (x14!=0)
  add     x12, x10, x12               {x12 = x10+x12;}
  label_b: addi   x10, x10, -1        label_b: x10 = x10-1;
  bne     x10, x0, label_a            if (x10!=0)
                                      {go to label_a;}
```

Call a Function

```
#include <stdio.h>
int B(int a, int b)
{
}
int A(int argc, const char * argv[]) {
}
```

Func_called:
0x2000 //one instruction
0x2004 //another instruction
0x2008 **ret** //**need jump back to main() return address**

PC

Start:
0x1000 //one instruction
0x1004 //another instruction
0x1008 //a third instruction
0x100c //PC jump to 0x2000 (call function B)
0x1010 //next instruction… …
… …

Call a Function—Jump

- JAL: Jump & Link, jump to function
- Unconditional jump (J-type)

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

```
jal rd label
```

1. Jump to label (`imm+PC`, explained later)
2. Save return address (`PC+4`) to `rd` (`x1`/`ra`) by convention;

When `rd` is `x0`, it is simply unconditional jump (j) without recording PC+4, usually used in loop/if/while

```
jal x0 label == j label (pseudo instruction)
```

Jump Example

- C code

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

Five variables `f` through `j` correspond to the five registers `x19` through `x23`

- Assembly

```
bne x22, x23, Else
//Go to Else if i≠j

add x19, x20, x21
//f = g + h (skipped if i≠j)

j Exit
//Jump to Exit

Else: sub x19, x20, x21
Exit: //Else branch & Exit
```

Call a Function Example

Registers

0
ra
sp
… …
s1
a0
a1
… …

Caller function:

```
# execute some instructions
jal  ra, callee_label
# execute some more instructions
```

Callee function:

```
callee_label: #execute some instructions
# how to return?
```

Return—Jump

- JALR: Jump & Link Register
- Unconditional jump (**I-type**)

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

```
jalr rd imm(rs1)
```

1. Jump to `(imm+rs1)`, `rs1` can be the return address we just saved to `ra`
2. Save return address `(PC+4)` to `rd`

JALR

- When we want to return from a function
  - Our return address is stored in a register (by JAL)
  - We don't need to save another return address

```
jalr x0, 0(ra) == jr ra == ret
```

rd

Call a Function Example

Caller function:

```
# execute some instructions
jal  ra, callee_label
# execute some more instructions
```

Callee function:

```
callee_label: #execute some instructions
# ret (jalr x0, 0(ra))
```

Call a Function

```
#include <stdio.h>
int B(int a, int b)
{
}
int A(int argc, const char * argv[]) {
}
```

Func_called:
0x2000 //one instruction
0x2004 //another instruction
0x2008 **ret** //**need jump back to main()**

Start:
PC 0x1000 //one instruction
0x1004 //another instruction
0x1008 //a third instruction
0x100c //**jal ra 0x2000** (call function B)
0x1010 //next instruction… …
… …

Registers

0
ra
sp
… …
s1
a0
a1
a2
a3
a4
… …

Jump

`—jal rd offset  —jalr rd rs offset`

- Jump and Link
  - Add the immediate value to the current address in the program (the "Program Counter"), go to that location
    - The offset is 20 bits, sign extended and left-shifted one (not two)
  - At the same time, store into rd the value of PC+4
    - So we know where it came from/need to return to
  - `jal offset == jal x1/ra offset` (pseudo-instruction; x1 = ra = return address)
  - `j offset == jal x0 offset` (jump is a pseudo-instruction in RISC-V)
- Two uses:
  - Unconditional jumps in loops and the like
  - Calling other functions

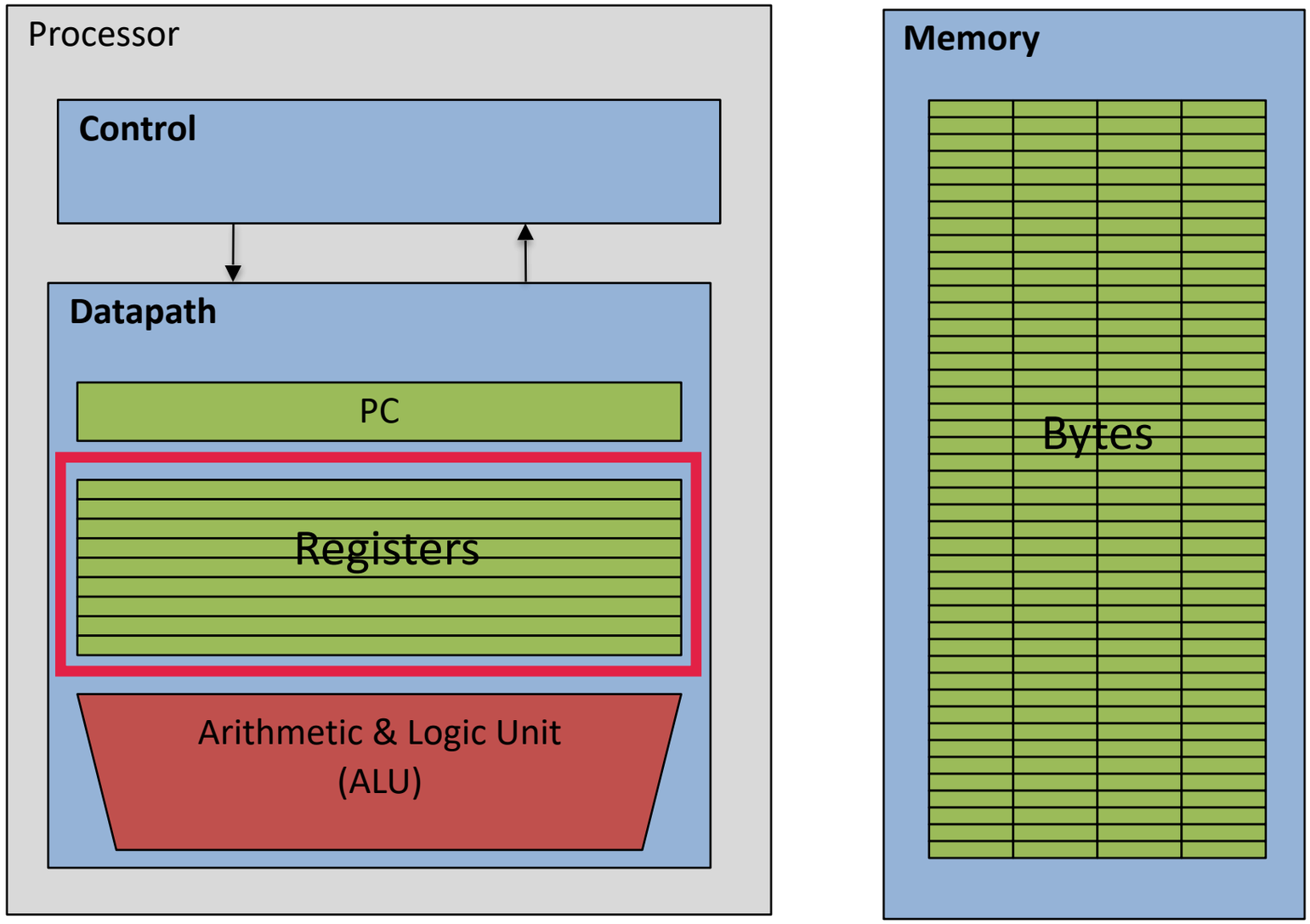
Jump and Link Register

- The same except the destination
  - Instead of PC + immediate it is rs + immediate
    - Same immediate format as I-type: 12 bits, sign extended
- Again, if you don't want to record where you jump to...
  - `jr rs == jalr x0 0(rs)`
  - `jr ra == ret`
- Two main uses
  - Returning from functions (which were called using JAL)
  - Calling pointers to function

Saving Registers

Caller function

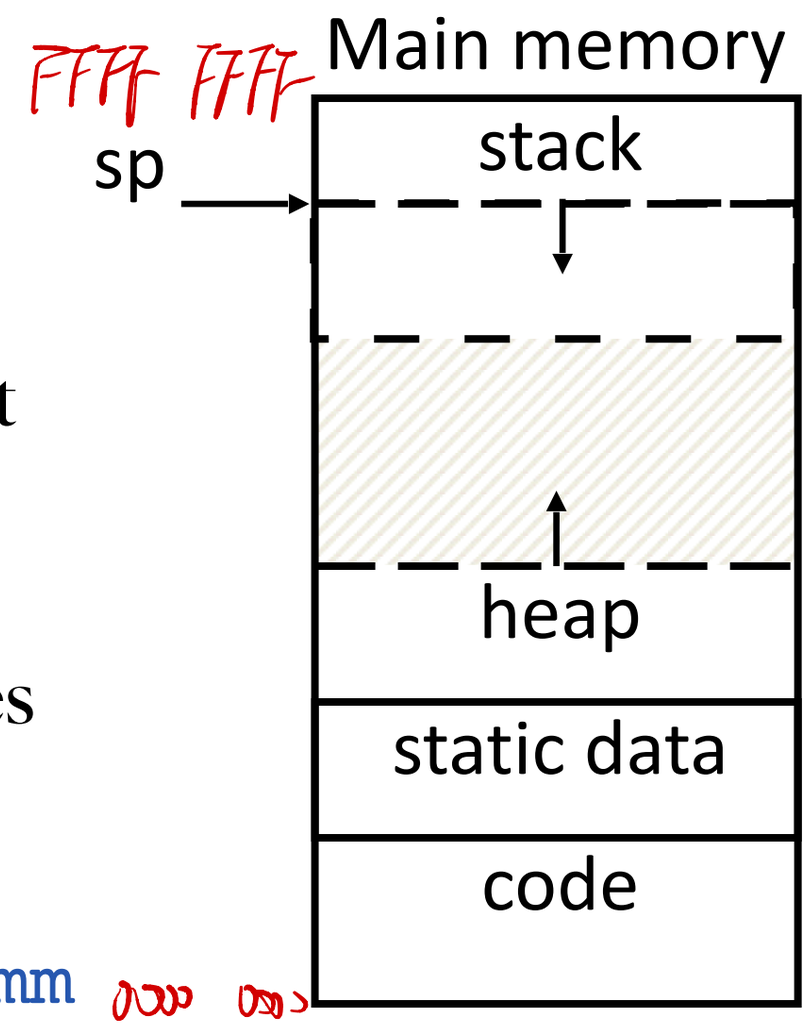
Callee function



One option: store all registers to memory and later load back

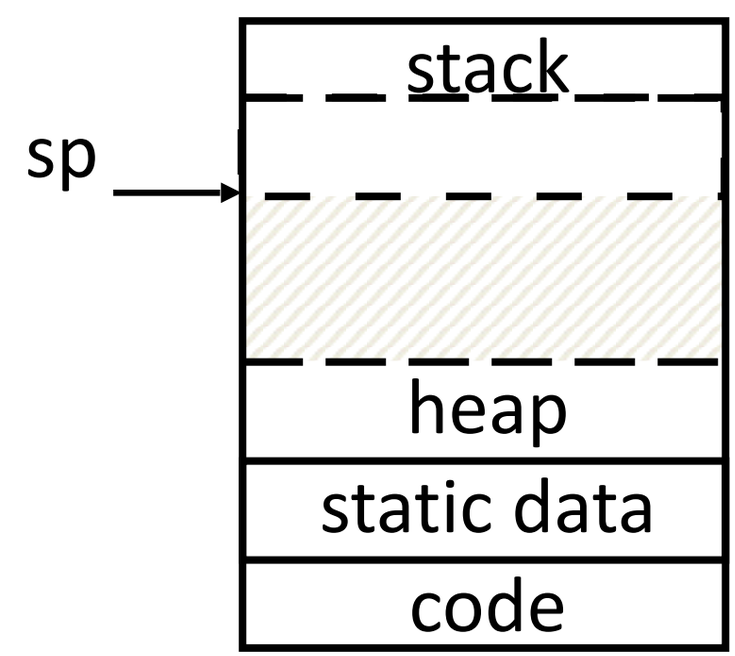
Where Are Old Register Values Saved to Restore Them After Function Call?

- Need a place to save old values before call function, restore them when return, and delete
- Ideal is stack: last-in-first-out queue (e.g., stack of plates)
  - Push: placing data onto stack
  - Pop: removing data from stack
- Stack in memory, so need register to point to it
- `sp` is the stack pointer in RISC-V (`x2`)
- Convention is grow from high to low addresses
  - Push decrements `sp`, Pop increments `sp`
  - `addi sp sp -imm; addi sp sp imm`



When to Save the Registers?

- We can save all of our registers before we call a function (before `jal`)
  - All registers would be saved by the caller
  - `sw/sb/sh rs (register to be saved) imm(sp)`
- We can save them before we actually use the registers (after `jal`)
  - All registers would be saved by the callee
  - `sw/sb/sh rs (register to be saved) imm(sp)`



Calling Convention

REGISTER NAME, USE, CALLING CONVENTION ④

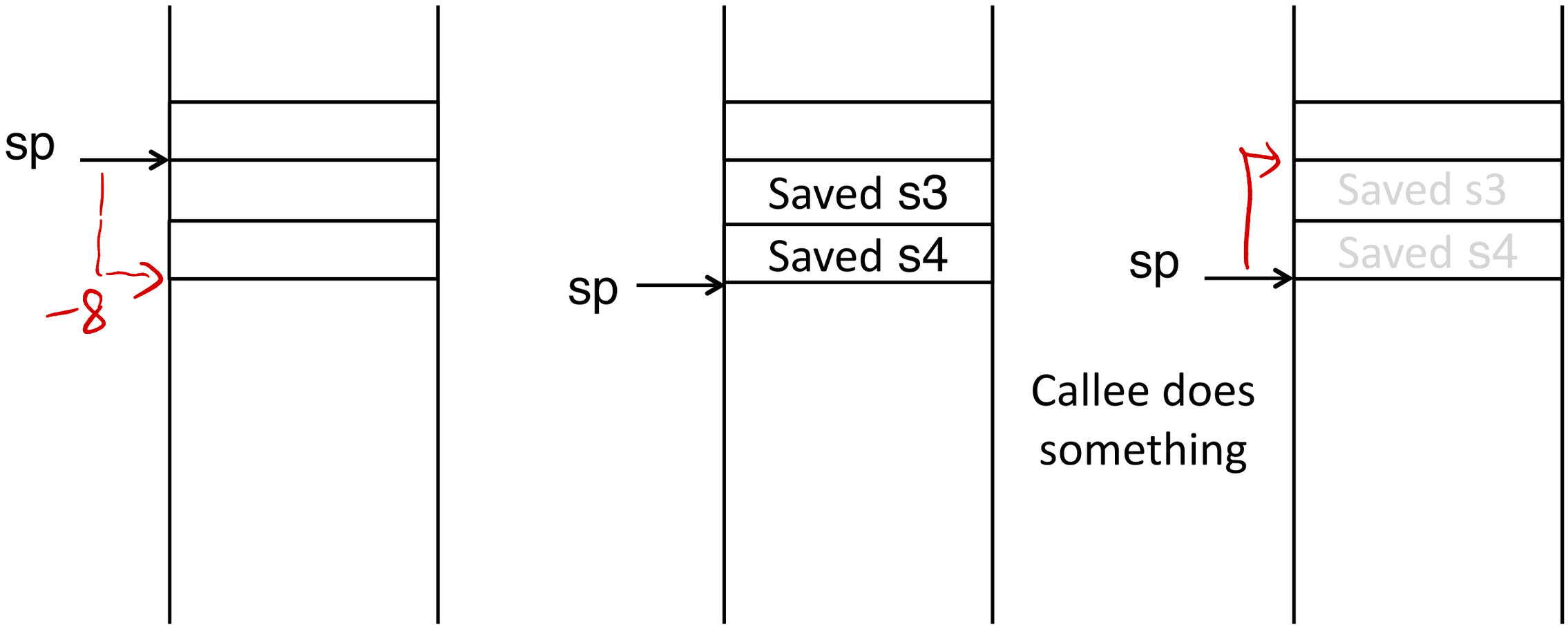
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

Caller saved registers:
If the caller has some temporary values in the registers that it wants to use after making a function call, it must save those values before JAL.

存memory中,结束后回返

Callee saved registers:
The callee saved registers should not change their values before and after the corresponding function call, i.e., callee is responsible to restore them if callee modifies them.

How to Save the Registers (Callee)?




```
addi sp sp -8
```

压栈

存2个值

2×4

```
sw s3 4(sp)
sw s4 0(sp)
```

```
lw s3 4(sp)
lw s4 0(sp)
addi sp sp 8
```

弹栈

Example

- Leaf function: a function that calls no function

```
int Leaf (int g, int h, int i, int j)
{
   int f; f = (g + h) - (i + j);
   return f;
}
```

- Parameter variables g, h, i, and j in argument registers `a0`, `a1`, `a2`, and `a3`, and f returned value in `a0` when returned
- Use temporary registers (`t0`-`t6`) for intermediate values

```
add t0,a0,a1   #g+h
add t1,a2,a3   #i+j
sub a0,t0,t1   #(g+h)-(i+j)
ret
```

Example

- Leaf function: a function that calls no function

```
int Leaf (int g, int h, int i, int j)
{
   int f; f = (g + h) - (i + j);
   return f;
}
```

- Assume using saved registers (`s0`-`s11`) for intermediate values

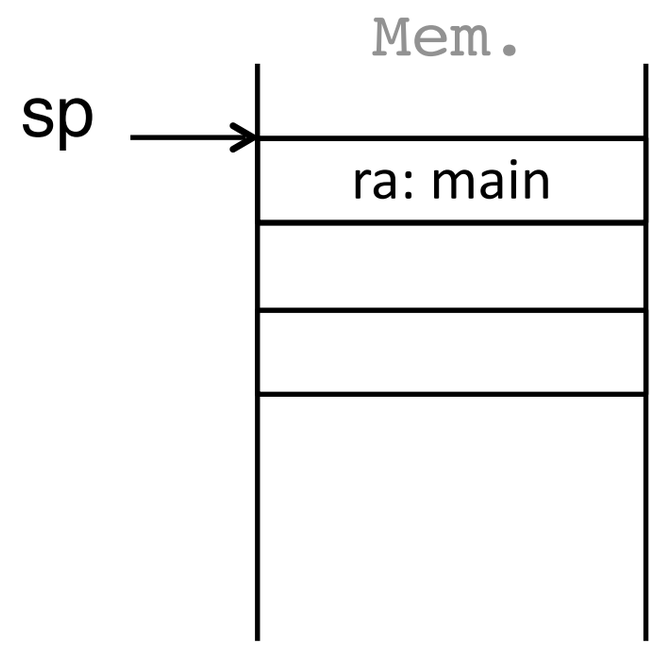
```
addi sp,sp,-8   #adjust sp to store
                 two intermediate
sw   s0,0(sp)
sw   s1,4(sp)
add  s0,a0,a1   #(g+h)
add  s1,a2,a3   #(i+j)
sub  a0,s0,s1   #(g+h)-(i+j)
lw   s0,0(sp)   #restore s0 & s1
lw   s1,4(sp)
addi sp,sp,8    #restore sp
ret
```

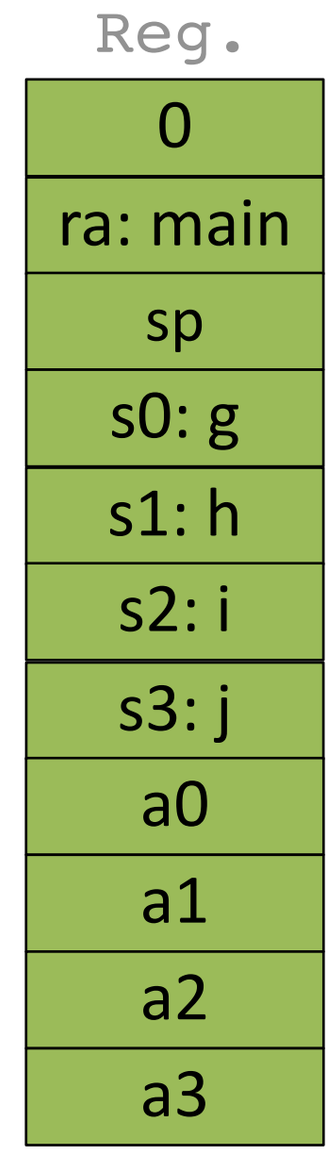
Example II

- A function that calls another function

```
int B(int g, int h, int i, int j) {
 int f = (g + h) - (i + j);
 return f;
}
int A(int x) {
 // do stuff
 int x = B(g, h, i, j);
 return (x * 2);
}
int main() {
 // do stuff
 A(x);
 // do stuff
}
```

callee + caller






```
# do stuff (omitted)
# save ra
addi sp,sp,-4
sw   ra,0(sp)
```

存ra

Example II

- A function that calls another function

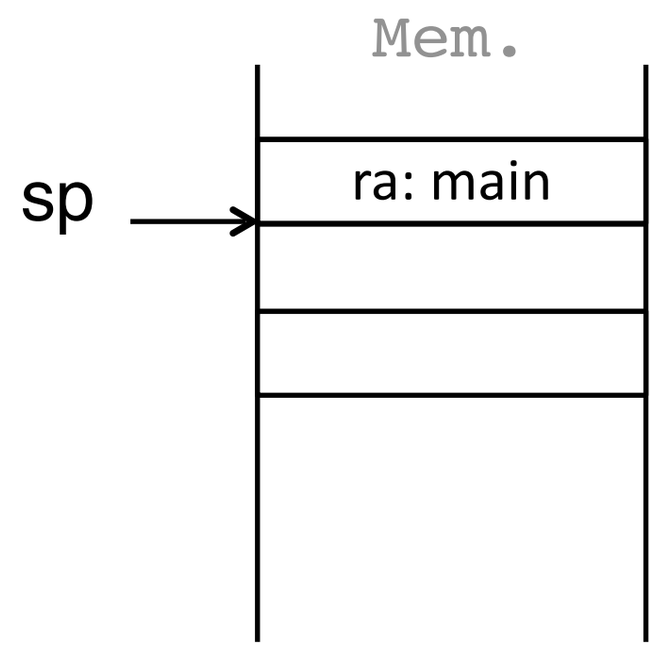
```
int A(int x) {
 // do stuff
 int x = B(g, h, i, j);
 return (x * 2);
}
```

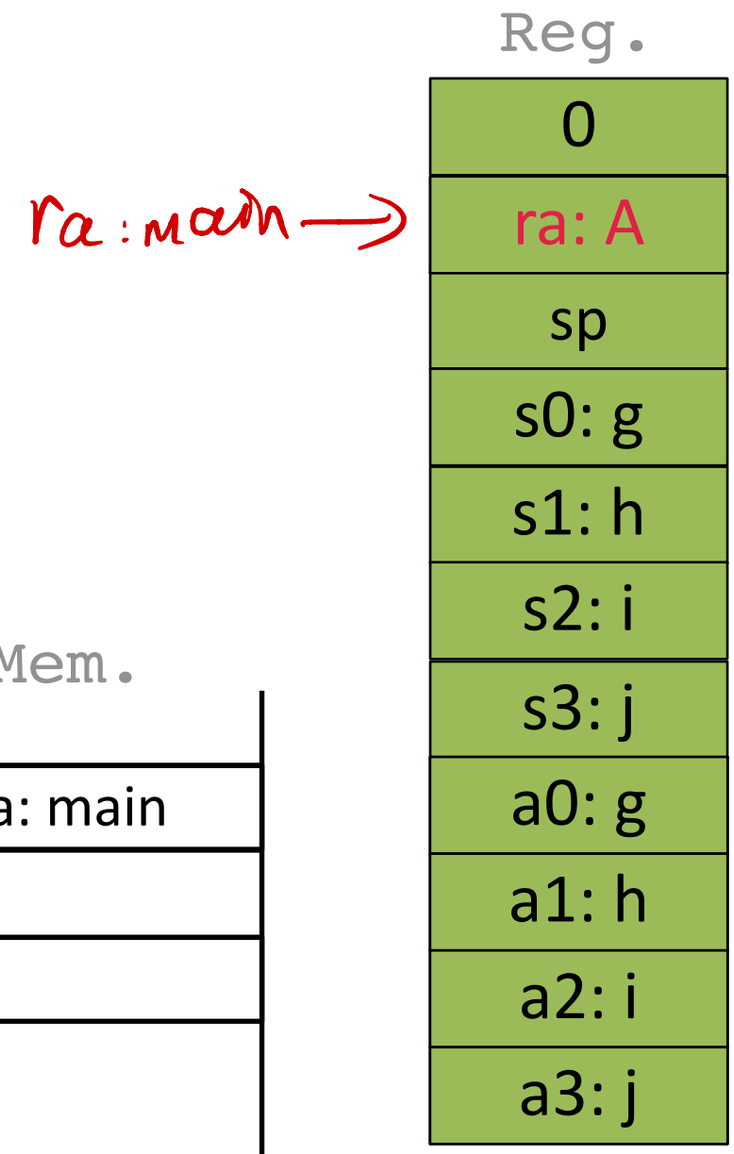
```
# do stuff (omitted)
# save ra
addi sp,sp,-4
sw   ra,0(sp)
# set up arguments
add  a0,s0,x0
add  a1,s1,x0
add  a2,s2,x0
add  a3,s3,x0
# jump to B
jal  ra,B
```

pre-load

g
h
i
j

从A跳转到B





Setup for function call (Prologue)

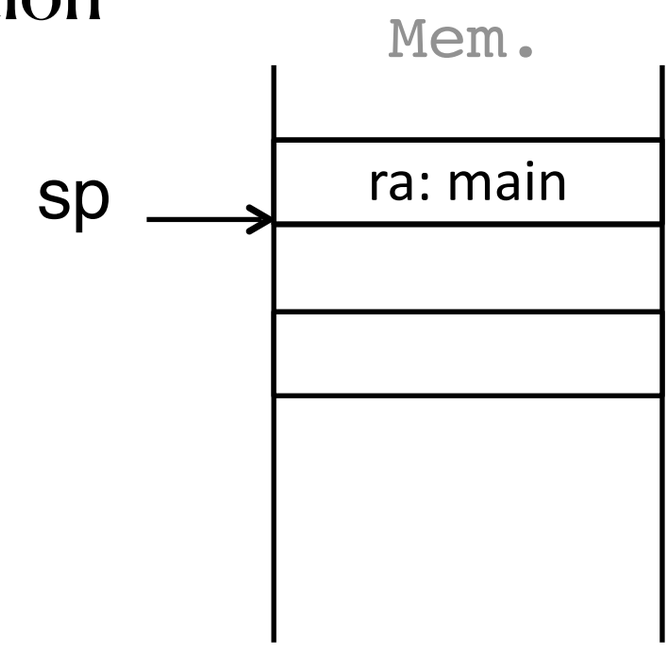
Example II

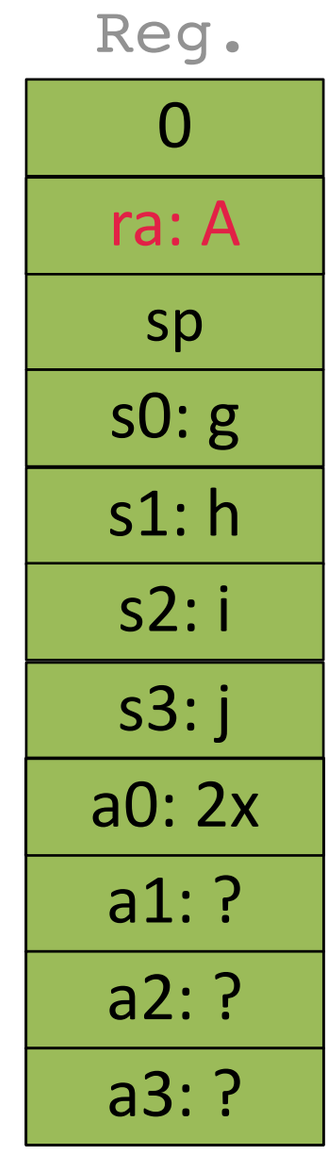
- A function that calls another function

```
int A(int x) {
 // do stuff
 int x = B(g, h, i, j);
 return (x * 2);
}
```

```
# do stuff (omitted)
# save ra
addi sp,sp,-4
sw   ra,0(sp)
# set up arguments
add  a0,s0,x0
add  a1,s1,x0
add  a2,s2,x0
add  a3,s3,x0
# jump to B
jal  ra,B
```

(Prologue)






```
# return from B
slli  a0,a0,1
# tear down from return
lw    ra,0(sp)
addi  sp,sp,4
# return to main
jr    ra
```

(Epilogue)

Calling Convention

REGISTER NAME, USE, CALLING CONVENTION ④

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

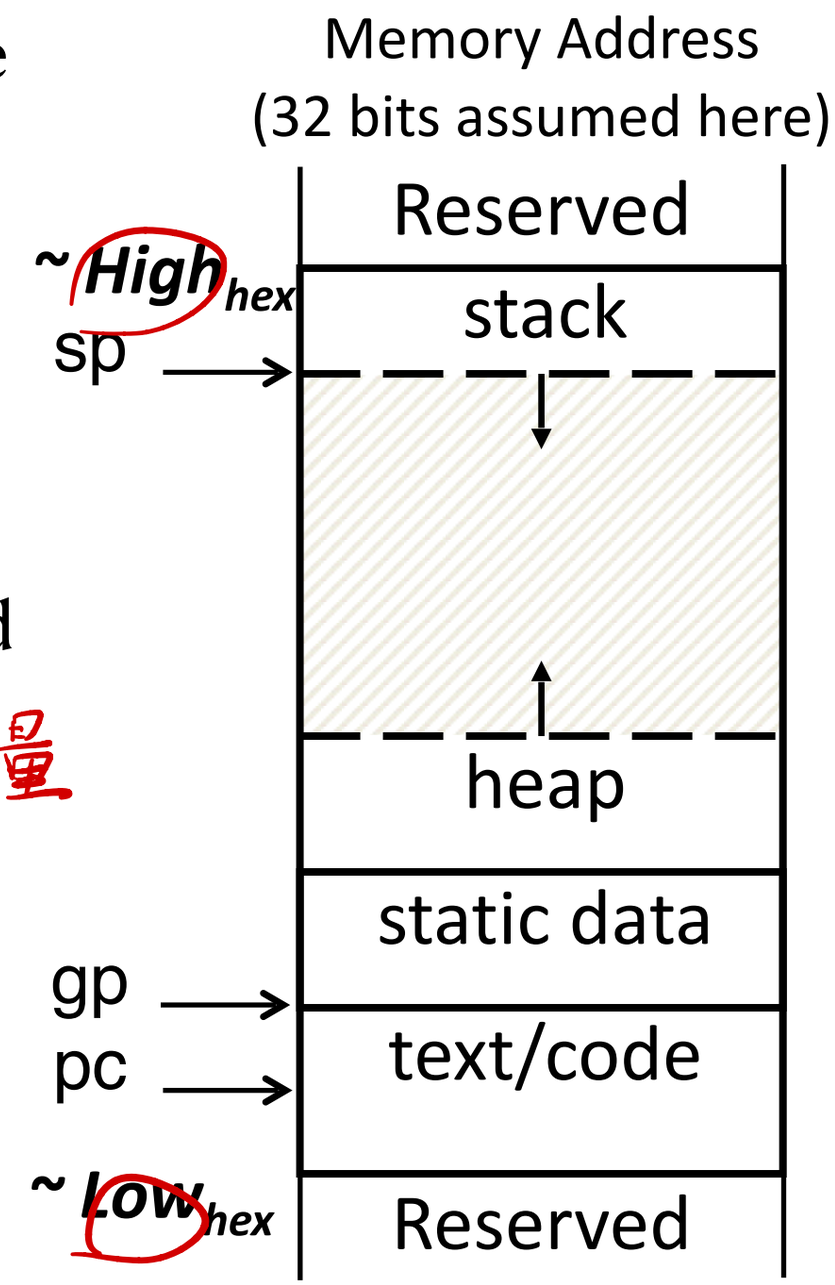
Call a Function

caller saver

1. Caller put parameters in a place where function can access them (a0-a7, or stack when registers not avail.), and then save caller-saved registers (temporaries, ax, ra, if necessary) to stack and addi sp
2. Transfer control to function (PC jump to function): JAL, ra is changed to where caller left
3. Acquire (local) storage resources needed for function: change sp (size decided when compiling); push callee-saved registers to stack (e.g., s0-s11)
4. Perform desired task of the function
5. Put result value in a place where calling code can access it (a0, a1), and restore callee-saved registers (s0-s11, sp)
6. Return control to point of origin, since a function can be called from several points in a program (jr); caller restores caller-saved registers

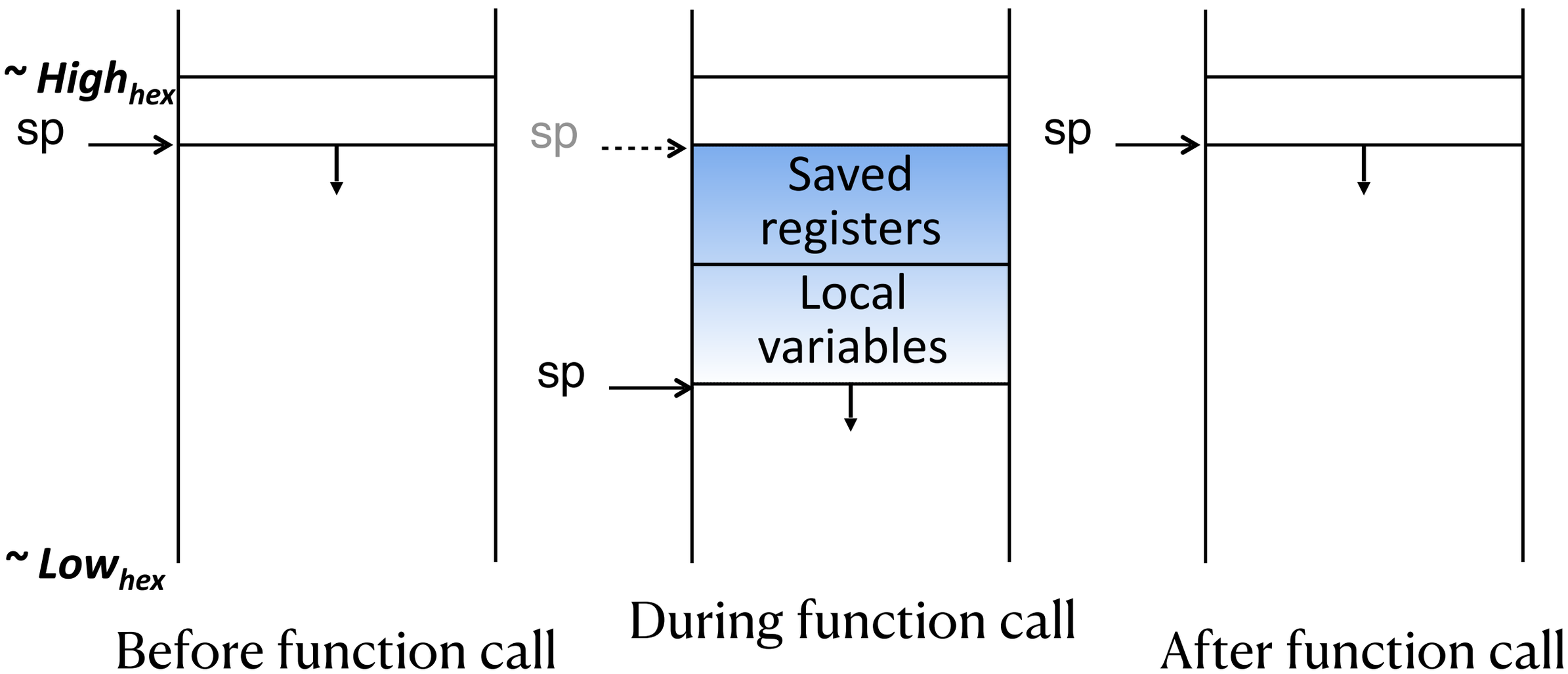
C Memory Management

- To simplify, assume one program runs at a time
- A program's address space contains 4 regions:
  - Stack: saved registers & local `auto` variables cannot fit into regs. → Stack
  - heap: space requested for dynamic data via `malloc`(); resizes dynamically, grows upward
  - static data: variables declared outside 全局變量 functions, does not grow or shrink. Loaded when program starts, can be modified.
  - code (a.k.a. text): loaded when program starts, does not change
  - 0x0 unwritable/unreadable (NULL pointer)



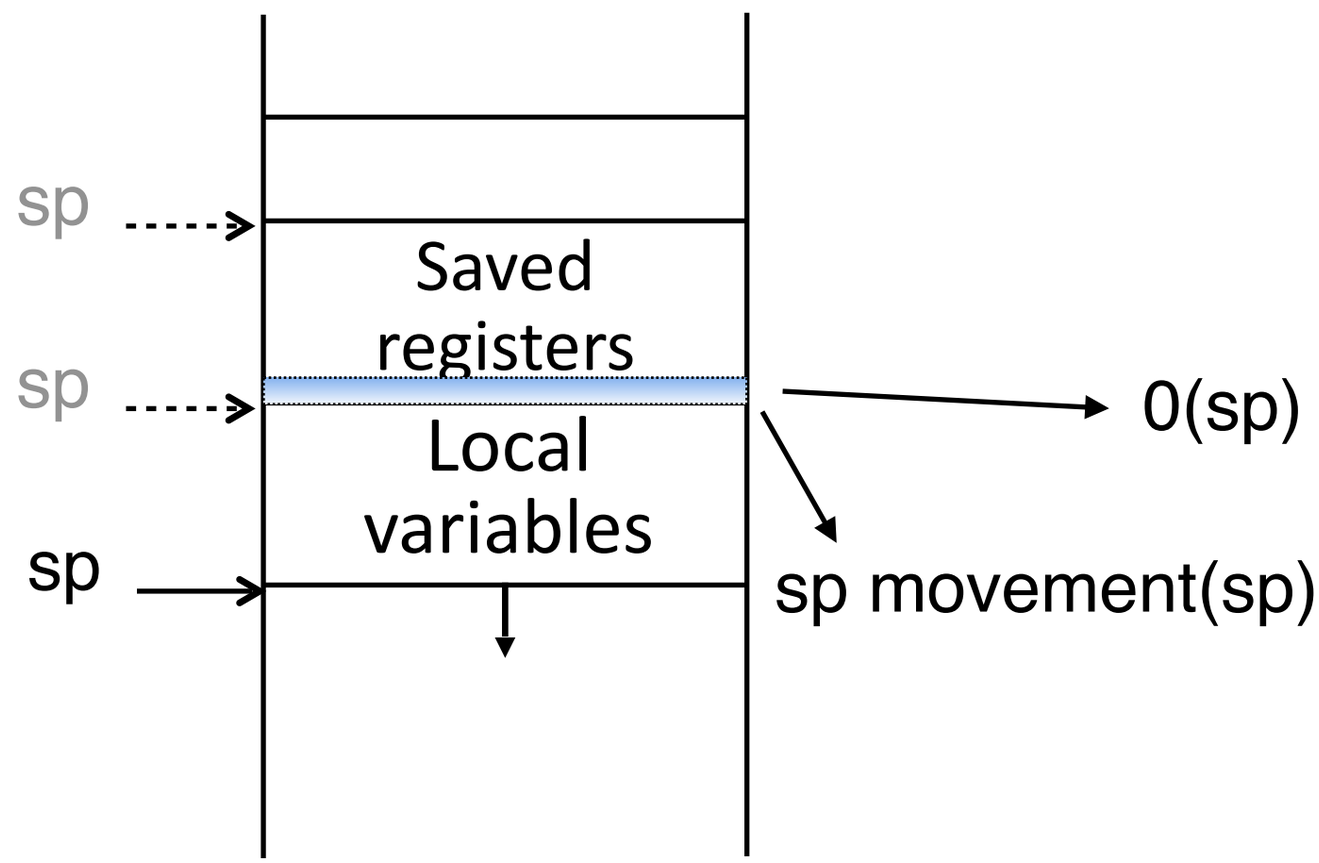
Procedure/Function Frame

- Also called activation record
- Containing procedure/function's saved registers & local variables



Procedure/Function Frame

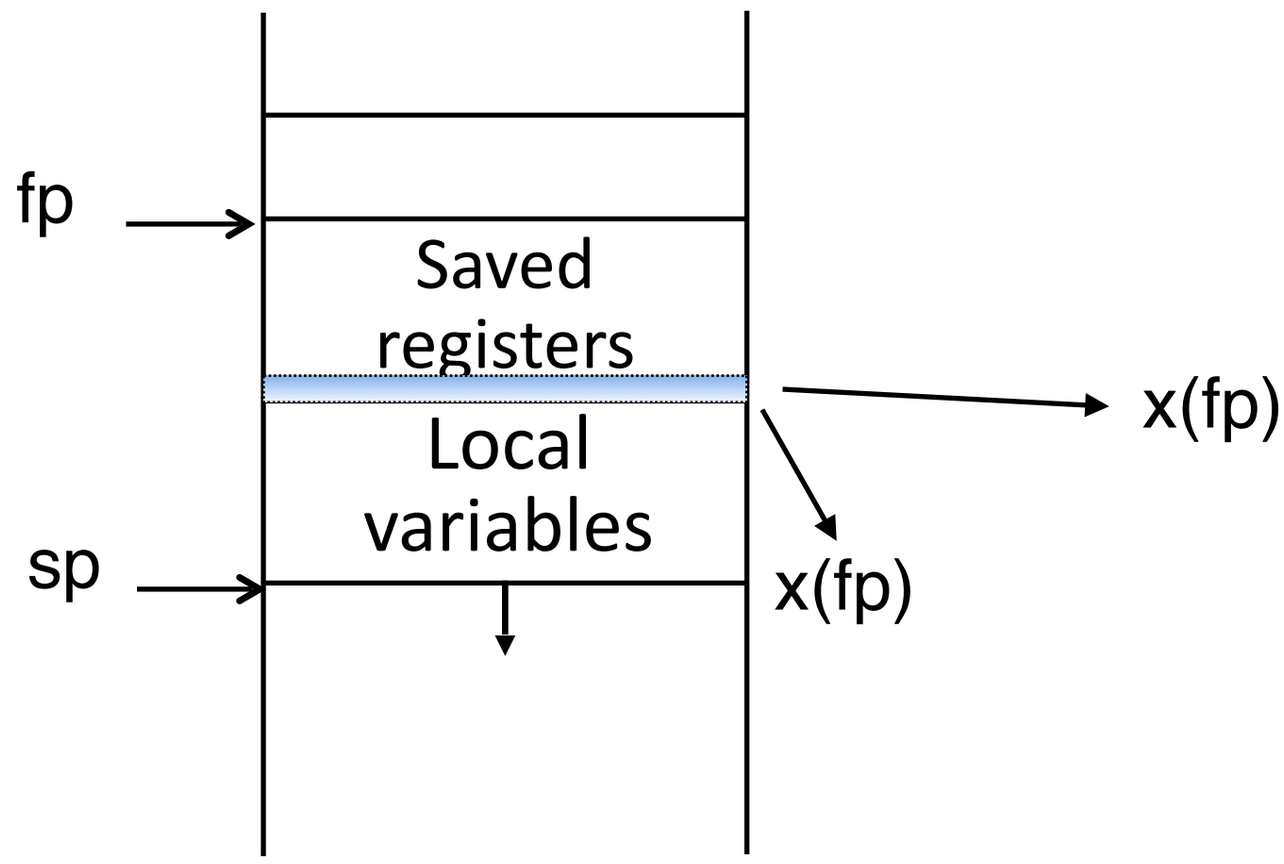
- Also called activation record
- Containing procedure's saved registers & local variables



During function call

Optional Frame Pointer (`s0`)

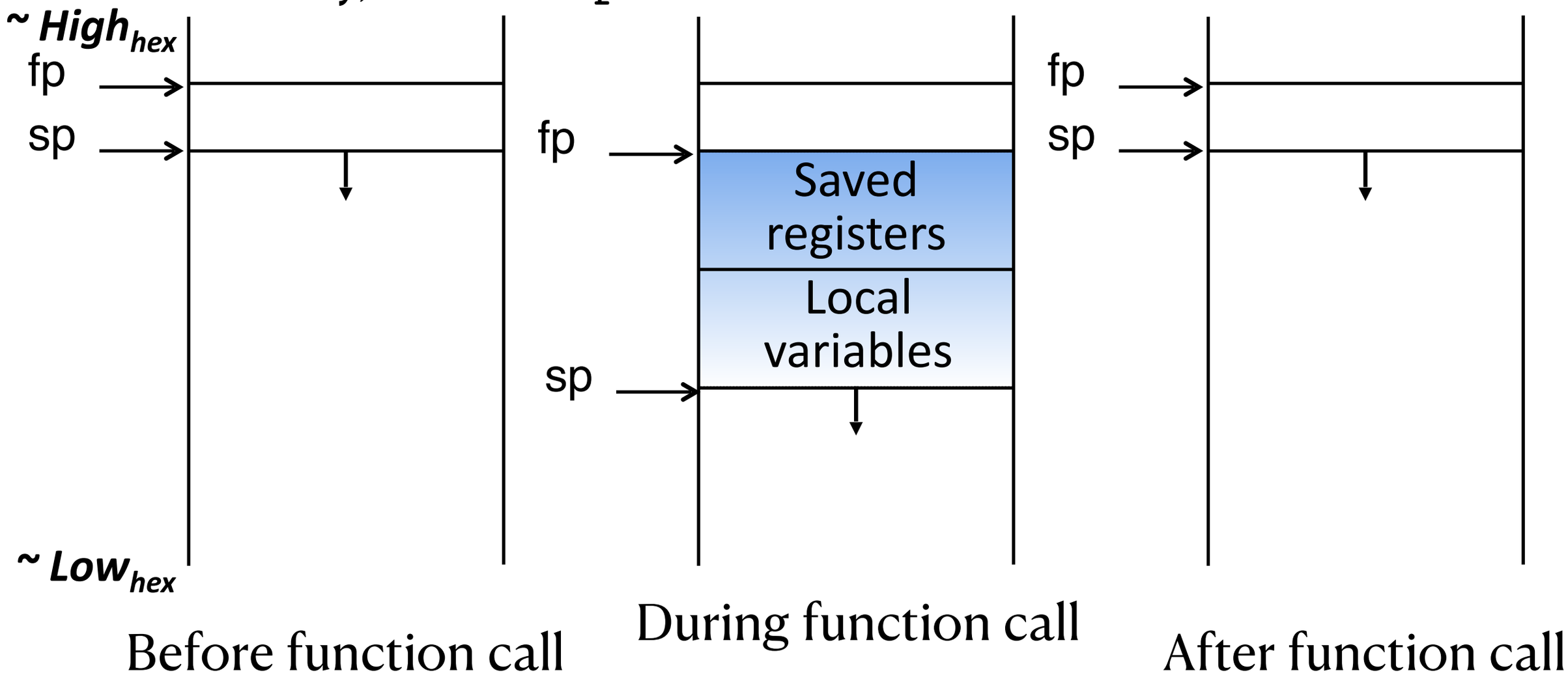
- Frame pointer does not change during a single procedure call



During function call

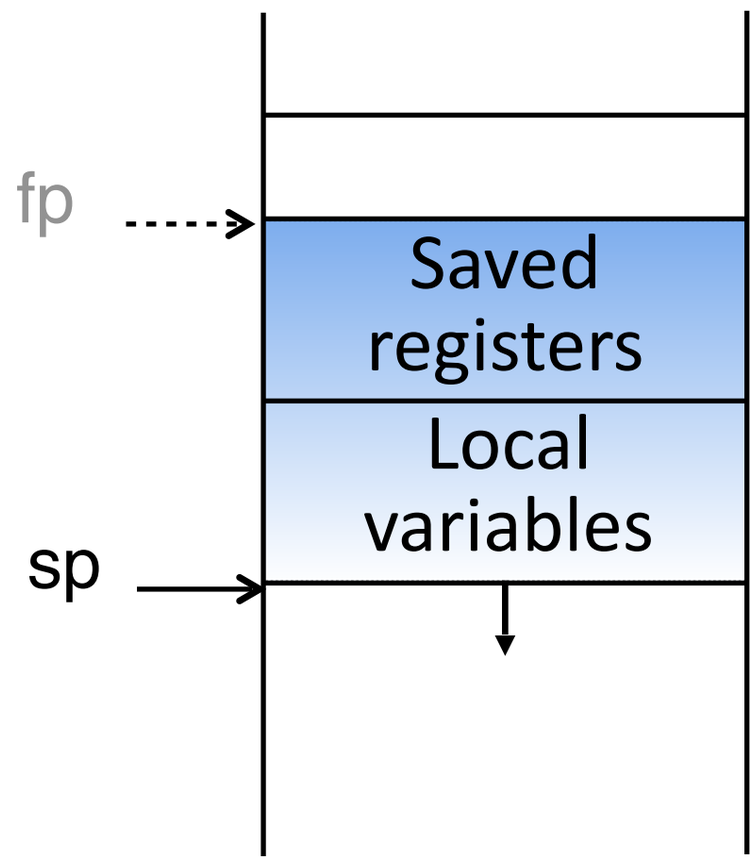
Frame Pointer

- Frame pointer is optional. Ignore for hand-written assembly.
- It is simply a saved register `s0` when not used as `fp`
- Alternatively, reduce `sp` movement



Stack for Local Variables

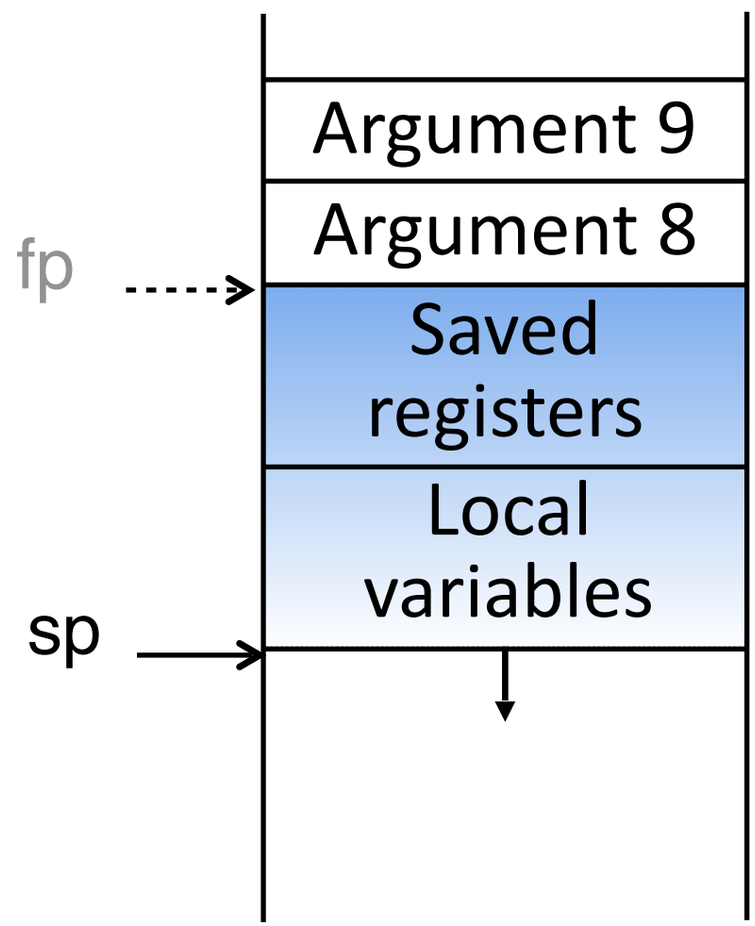
- For example a large array
- Require enough space and usually padding
  - In the standard RISC-V calling convention, `sp` is always 16-byte aligned.



During function call

Stack for Excessive Arguments

- What if we have 10 arguments? Only 8 argument registers (`a0-a8`).
- Use caller's procedure frame, and `fp` to access



During function call

信息科学与技术学院
School of Information Science and Technology

CS 110
Computer Architecture
RISC-V Instruction Formats

Instructors:

Siting Liu & Chundong Wang

Course website: https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2023/index.html

School of Information Science and Technology (SIST)

ShanghaiTech University

2023/2/6

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

Compiler

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

Assembler

Machine Language Program (RISC-V)

Machine Interpretation

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

Architecture Implementation

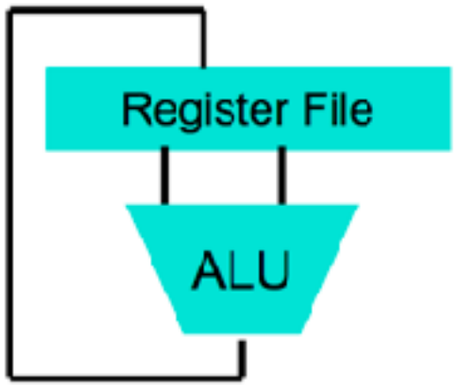
Logic Circuit Description (Circuit Schematic Diagrams)

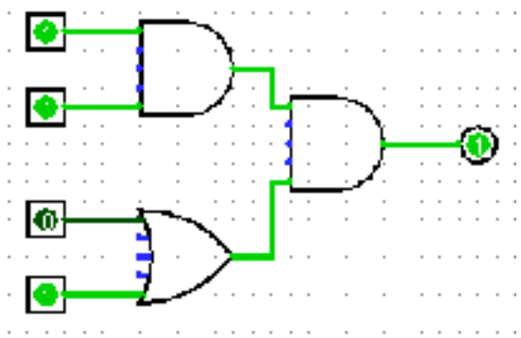
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw    xt0, 0(x2)
lw    xt1, 4(x2)
sw    xt1, 0(x2)
sw    xt0, 4(x2)
```

机械码

```
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
```





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			rs		rd	1110011	CSRRW
csr			rs		rd	1110011	CSRRS
csr			rs		rd	1110011	CSRRC
csr			zimm	101	rd	1110011	CSRRWI
csr			zimm	110	rd	1110011	CSRRSI
csr			zimm	111	rd	1110011	CSRRCI

Not in CS110

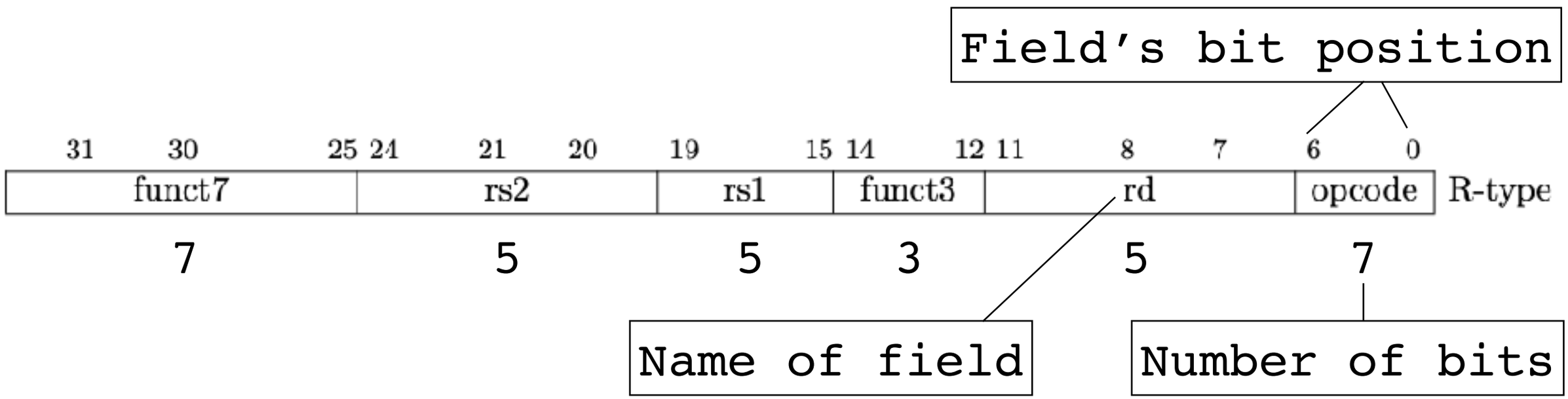
Instruction Formats

~ High Bit ***~ Low Bit***

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] imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]				rd		opcode	U-type
imm[20] imm[10:1]		imm[11]	imm[19:12]	rd		opcode	J-type

- All 32-bit in length (not the case in RVC)
- Generally, fields are aligned if present (`rs1, rs2, rd, funct3, funct7, opcode`)
- Different number/type of operands/result

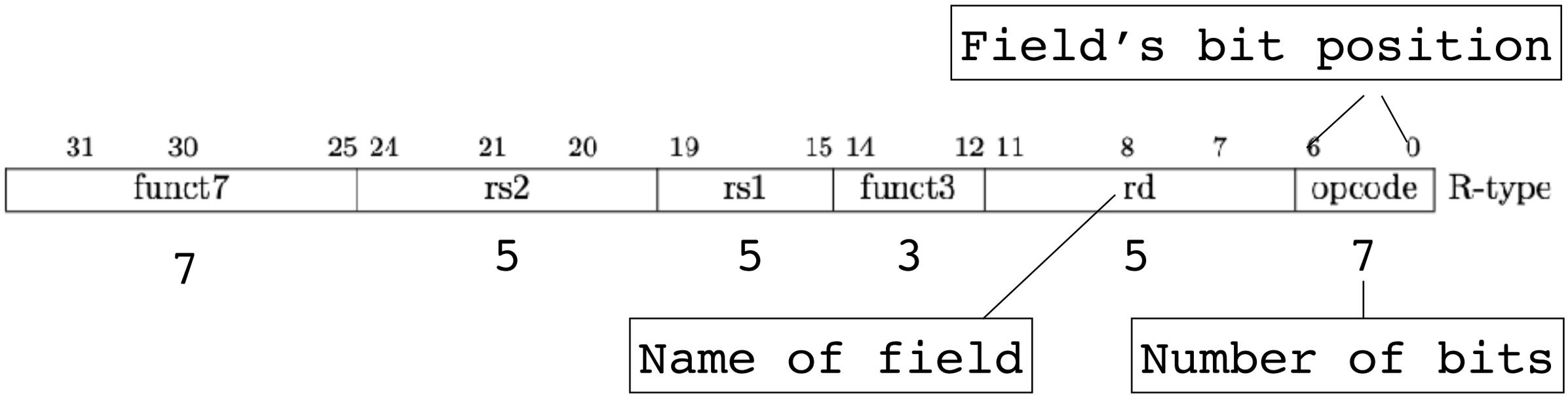
R-Format



Assembly: Operation rd, rs1, rs2

rd, rs1, rs2 unsigned numbers, represent No. of regs.

R-Format



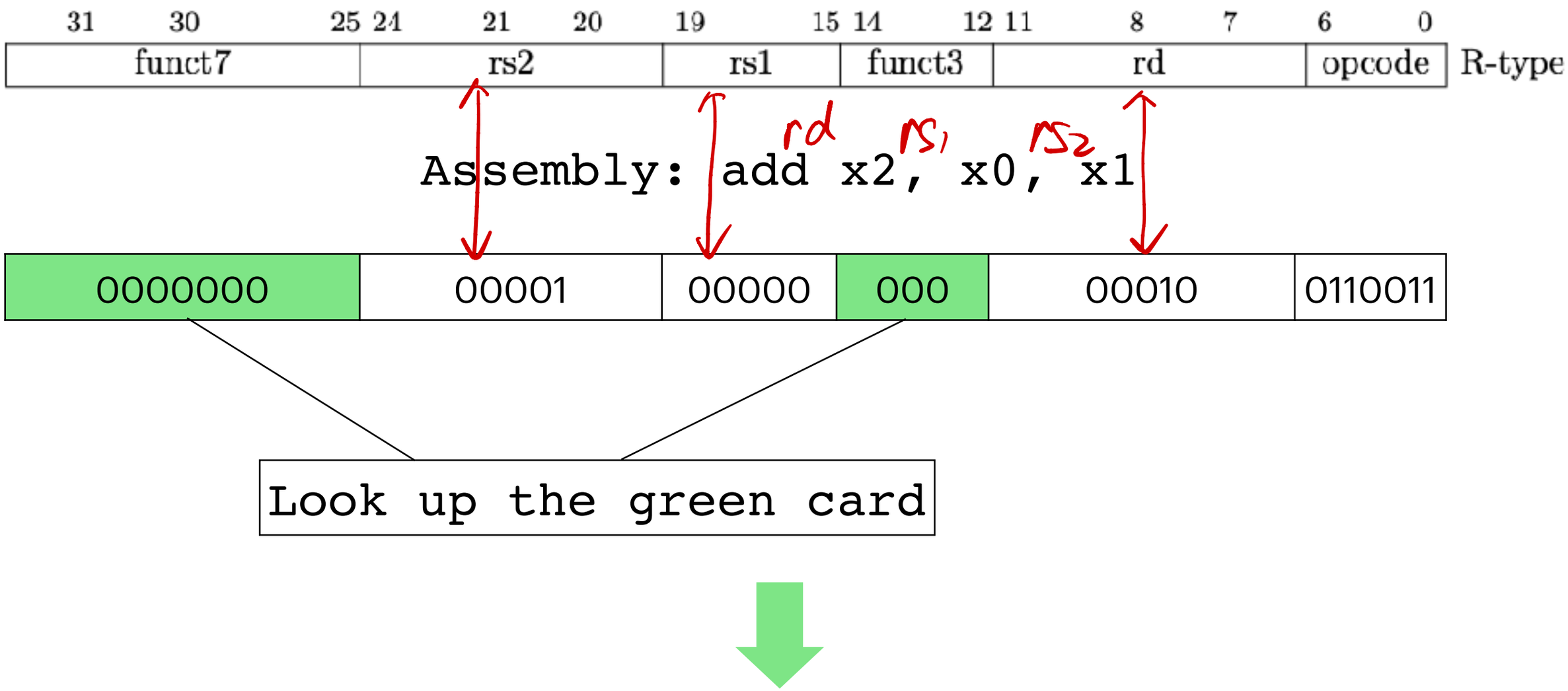

```
Assembly: Operation rd, rs1, rs2
```

funct7/funct3/opcode fields:

所有指令 opcode相同 → Rtype

- Opcode: `0b0110011` for RV32I R-format arithmetic/logic operations
- funct7/funct3 together decide the type of operation

R-Format Example



Machine code: concatenate all fields

0000_0000_0001_0000_0000_0001_0011_0011

0x00100133

R-Format—All Instructions

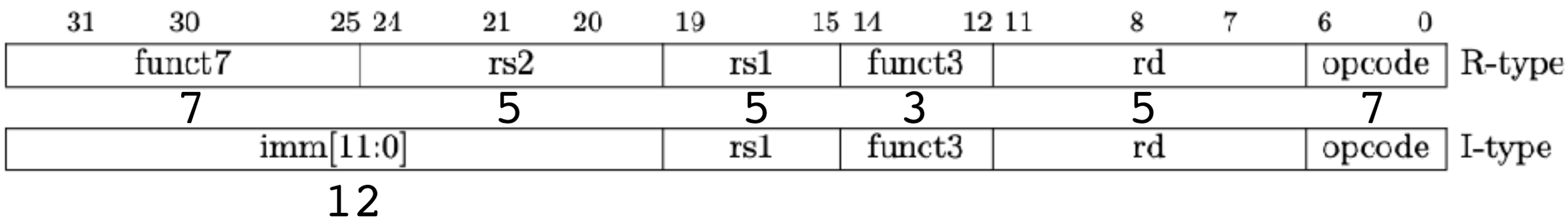
Assembly: Operation rd, rs1, rs2

funct7 funct3

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

funct7/funct3/opcode together decide the operation

I-Format



Assembly: `Operation rd, rs1, imm`

Register-immediate type signed

- `imm`: 12 bits, hold values for `[-2048, 2047]`
- `imm` sign-extended before operations (sign-extension done in hardware)
- Opcode `0b0010011` for I-type arithmetic/logic operations

I-Format Example I

	5	3	5	7	
imm[11:0]	rs1	funct3	rd	opcode	I-type
12					

Assembly: addi x2, x0, 1234

010011010010	00000	000	00010	0010011

Look up the green card

Machine code: concatenate all fields

0100_1101_0010_0000_0000_0001_0001_0011

0x4d200113

I-Format Example II

	5	3	5	7	
imm[11:0]	rs1	funct3	rd	opcode	I-type
12					

Assembly: addi x2, x0, -1234

101100101110	00000	000	00010	0010011

2's complement

Look up the green card

Machine code: concatenate all fields

1011_0010_1110_0000_0000_0001_0001_0011

0xb2e00113

I-Format

imm[11:0]	rs1	funct3	rd	opcode	I-type
12	5	3	5	7	

0000000	shamt[4:0]	src	001	dest	0010011	SLLI
0000000	shamt[4:0]	src	101	dest	0010011	SRLI
0100000	shamt[4:0]	src	101	dest	0010011	SRAI

Register-immediate type

- `imm`: 12 bits, hold values for `[-2048, 2047]` (Not for shift operations)
- `shamt` not sign-extended before operations for shifts
- Opcode `0b0010011` for I-type arithmetic/logic operations
- Shift is specialized, since shift more than 31 bits is meaningless

I-Format Arithmetic & Logic

imm[11:0]	rs1	funct3	rd	opcode	I-type
12	5	3	5	7	

imm	src	000	dest	0010011	ADDI
imm	src	010	dest	0010011	SLTI
imm	src	011	dest	0010011	SLTIU
imm	src	100	dest	0010011	XORI
imm	src	110	dest	0010011	ORI
imm	src	111	dest	0010011	ANDI

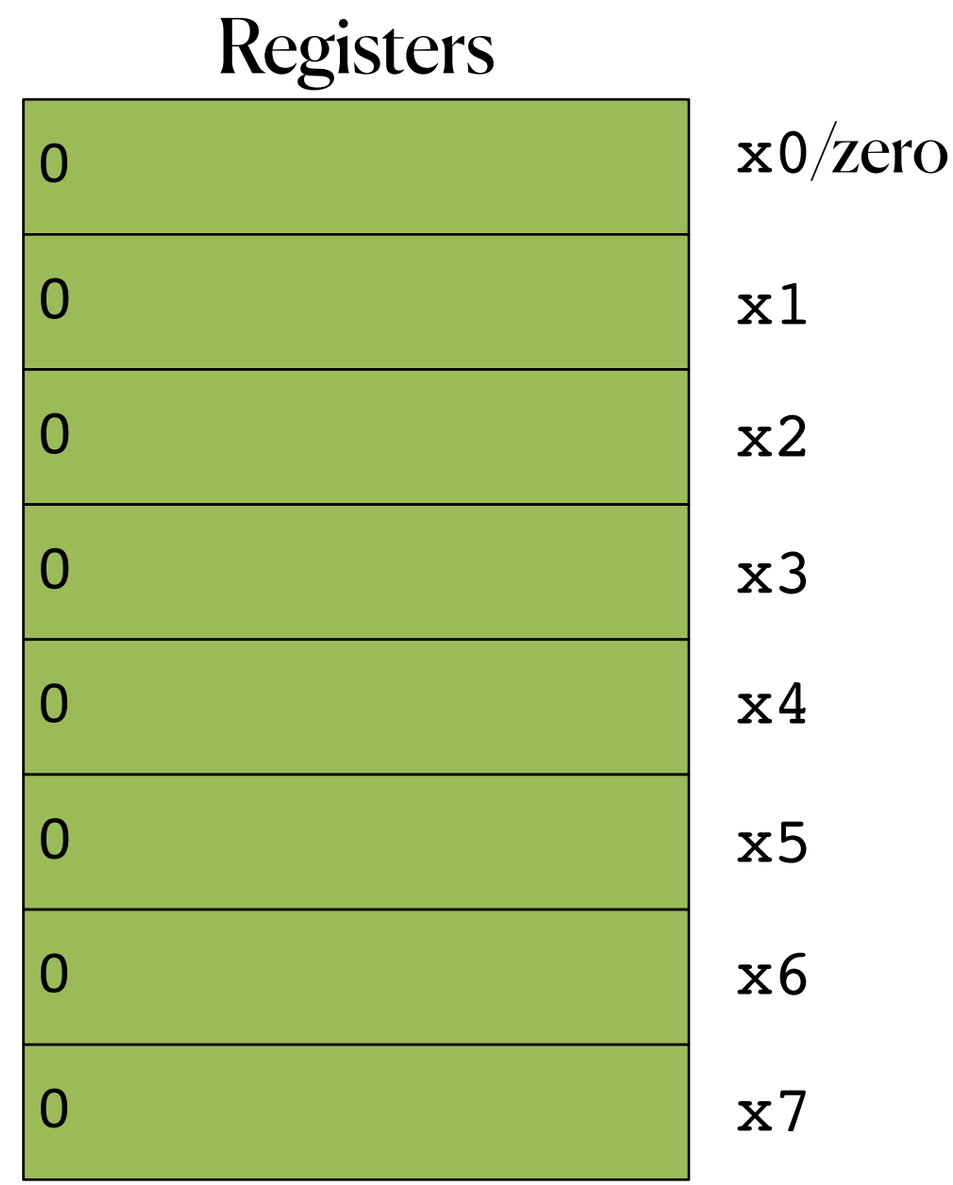
0000000	shamt[4:0]	src	001	dest	0010011	SLLI
0000000	shamt[4:0]	src	101	dest	0010011	SRLI
0100000	shamt[4:0]	src	101	dest	0010011	SRAI

Same as corresponding R-type funct3

Correction!!!

- `addi x1, x0, -1`
- `or  x2, x2, x1`
- `add  x3, x1, x2`
- `slt  x4, x3, x1`
- `sra  x5, x3, x4`
- `sub  x0, x5, x4`

- Register zero (`x0`) is 'hard-wired' to 0;
- By **convention** RISC-V has a specific no-op instruction... ⇒PC向下移4
  - **`addi x0 x0 0`**
  - You may need to replace code later: No-ops can fill space, align data, and perform other options
  - Practical use in jump-and-link operations (covered later)



I-Format Load

imm[11:0]	rs1	funct3	rd	opcode	I-type
12	5	3	5	7	

`Assembly: lw/lhu/lh/lb/lbu rd, (imm)rs1`

- Opcode: `0b0000011` for RV32I R-format load operations
- funct3:
  - First bit indicates signed(`0`)/unsigned(`1`)
  - Last 2 bits indicates w(`10`)/h(`01`)/b(`00`)

I-Format Load

imm[11:0]	rs1	funct3	rd	opcode	I-type
12	5	3	5	7	

Assembly: lw/lhu/lh/lb/lbu rd, imm(rs1)

imm	src	000	dest	0000011	LB
imm	src	001	dest	0000011	LH
imm	src	010	dest	0000011	LW
imm	src	100	dest	0000011	LBU
imm	src	101	dest	0000011	BHU

2’s complement

I-Format Load Example

imm[11:0]	rs1	funct3	rd	opcode	I-type
12	5	3	5	7	

Assembly: lw/lhu/lh/lb/lbu rd, imm(rs1)

imm	src	000	dest	0000011	LB
imm	src	001	dest	0000011	LH
imm	src	010	dest	0000011	LW
imm	src	100	dest	0000011	LBU
imm	src	101	dest	0000011	BHU

2's complement

rd FFFF... rs1

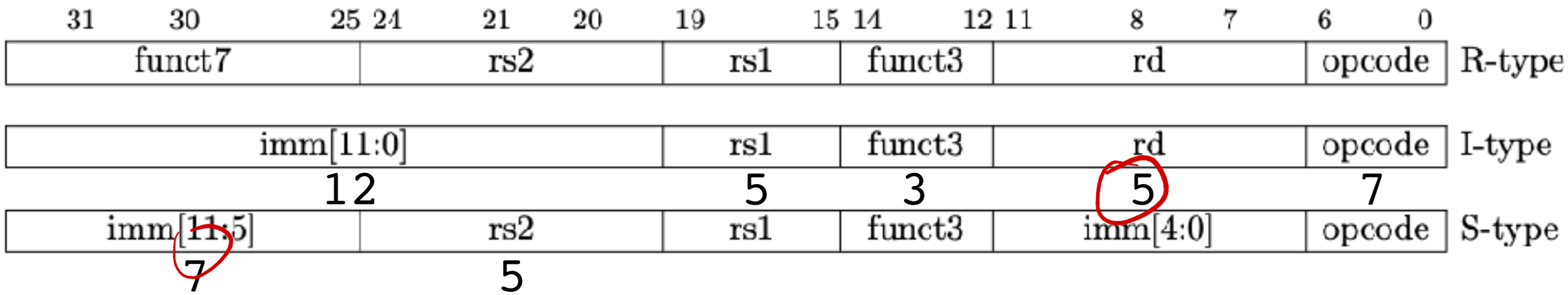
Assembly: lbu x18, -1(x17)

FFF	10001	100	10010	0000011

Machine code:1111_1111_1111_1000_1100_1001_0000_0011

0xFFF8c903

S-Format Store




```
Assembly: sw/sh/sb rs2, imm(rs1)
```

- Opcode: `0b0100011` for RV32I S-format store operations
- funct3:
  - Last 2 bits indicates w(`10`)/h(`01`)/b(`00`)
  - First bit `0`

S-Format Store Instructions

		5	3	5	7	
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
7	5					

```
Assembly: sw/sh/sb rs2, imm(rs1)
```

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

S-Format Store Example

		5	3	5	7	
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
7	5					

Assembly: sw x20, -31(x21)

1111111	10100	10101	010	00001	0100011

Machine code:1111_1111_0100_1010_1010_0000_1010_0011

0xFF4aa0a3

B-Format Conditional Branch

			5	3	5		7	
imm[11:5]		rs2	rs1	funct3	imm[4:0]		opcode	S-type
7		5						
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type

Assembly: `bne/beq/blt/bltu/beg/begu rs1, rs2, label`

- Opcode: `0b1100011` for RV32I B-format branch operations
- How to represent label?

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)

C Loop Mapped to RISC-V Assembly

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

```asm
# Assume x8 holds pointer to A
# Assign x10=sum
   add x9, x8, x0   # x9=&A[0]
   add x10, x0, x0 # sum=0
   add x11, x0, x0 # i=0
   addi x13,x0, 20 # x13=20
Loop:
   bge x11,x13,Done
   lw x12, 0(x9)    # x12=A[i]
   add x10,x10,x12 # sum+=
   addi x9, x9,4    # &A[i+1]
   addi x11,x11,1   # i++
   j Loop
Done:
```

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 $\pm 2^{11}$ 'unit' addresses from the PC
- Recall
  - Each instruction is 4-byte wide (4-byte aligned)
  - Address is multiple of 4, least significant 2 bits "`00`"
  - Can use bits `[13:2]` for imm
  - Can specify $\pm 2^{13}$ 'unit' addresses from the PC
  - But, to support RVC (16-bit/2-byte instruction) extension, `[12:1]` for imm/offset, can specify $\pm 2^{12}$ 'unit' addresses from the PC
- Opposite to it, absolute addressing (use full address)

```
Disassembly of section .
0000000000000000 <ltmp0:
   0: ff c3 00 d1
   4: fd 7b 02 a9
   8: fd 83 00 91
   c: 08 00 80 52
  10: e8 0f 00 b9
  14: bf c3 1f b8
  18: 48 9a 80 52
  1c: a8 83 1f b8
```

B-Format Conditional Branch

7	5	5	3	5	7	
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type

```
Assembly: bne/beq/blt/bltu/beg/begu rs1, rs2, label
```

- Opcode: `0b1100011` for RV32I B-format branch operations
- Label: PC-relative addressing, concatenate `imm[12]`, `imm[11]`, then `imm[10:5]` and `imm[4:1]` as offset (sign-extended)

B-Format Conditional Branch Example

7	5	5	3	5	7	
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type

```
rs1 = 01011

rs2 = 01101

opcode = 1100011

funct3 = 101

imm/offset
= 6 instructions
= 24 bytes

0|0|000000|1100|0
Bit 12        Bit 0
```

```
# Assume x8 holds pointer to A
# Assign x10=sum
   add x9, x8, x0   # x9=&A[0]
   add x10, x0, x0 # sum=0
   add x11, x0, x0 # i=0
   addi x13,x0, 20 # x13=20
Loop:
PC → bge x11,x13,Done
   lw x12, 0(x9)    # x12=A[i]
   add x10,x10,x12 # sum+=
   addi x9, x9,4    # &A[i+1]
   addi x11,x11,1   # i++
   j Loop
Done:  # some instruction
```

B-Format Conditional Branch Example

7	5	5	3	5	7		
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type

```
rs1 = 01011    0  000000  01101  01011  101  1100  0  1100011

rs2 = 01101
```

Machine code

```
opcode = 1100011
```

0x00d5dc63

```
funct3 = 101

imm/offset

= 6 instructions
= 24 bytes
```

```
    0|0|000000|1100|0
Bit 12          Bit 0
```

B-Format Branch Instructions

7	5	5	3	5	7	
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type

```
Assembly: bne/beq/blt/bltu/beg/begu rs1, rs2, imm/offset
```

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

Further on Conditional Branch

- To support RVC (16-bit/2-byte instruction) extension, `[12:1]` for imm/offset, can specify $\pm 2^{12}$ 'unit' addresses from the PC
- Equivalent to $\pm 2^{10}$ 32-bit instructions
- What if jump to farther away?

```
beq x10, x0, far
# next instruction
```

```
bne x10, x0, next
next: # next instruction
j far
```

j gets 20-bit imm

J-Format Jump Instruction

1	10	1	8	5	7	
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode	J-type

`Assembly: jal rd, label`

- Recall `jal` does 2 things:
  - Store `PC+4` to `rd` as return address
  - Jump to `label` = `PC + offset(imm)`

- Label translated by assembler to a 20-bit offset (encoded similar to Branch offset)
- Can access $\pm 2^{20}$ 'unit' addresses from the PC
- $\pm 2^{18}$ 32-bit instructions

unit ÷4 → bit

I-Format Jump Instruction

imm[11:0]	rs1	funct3	rd	opcode	I-type
12	5	3	5	7	

`Assembly: jalr rd, rs1, imm`

- Recall `jalr` does 2 things:
  - Store `PC+4` to `rd` as return address
  - Jump to `label` = `rs + offset(imm)`
- `imm` can hold values between `[-2048,2047]`
- Unlike JAL, include the last 0 using I-format

Corner Cases

20	5	7	
imm[31:12]	rd	opcode	U-type

```
li x5, 0xABCDEFEF
```

```
lui x5, 0xABCDE
```

```
addi x5, x5, 0xFEF
```

!!!!

符号位拓展

0xFFFF FFEF

```
lui x5, 0xABCDF
```

补上1

```
addi x5, x5, 0xFEF
```

Registers

0xABCDE000	x5
	x5

ABCDF000
FFFFFFEF
ABCD[D]FEF

差了1

This is automatically handled by `li`. !!

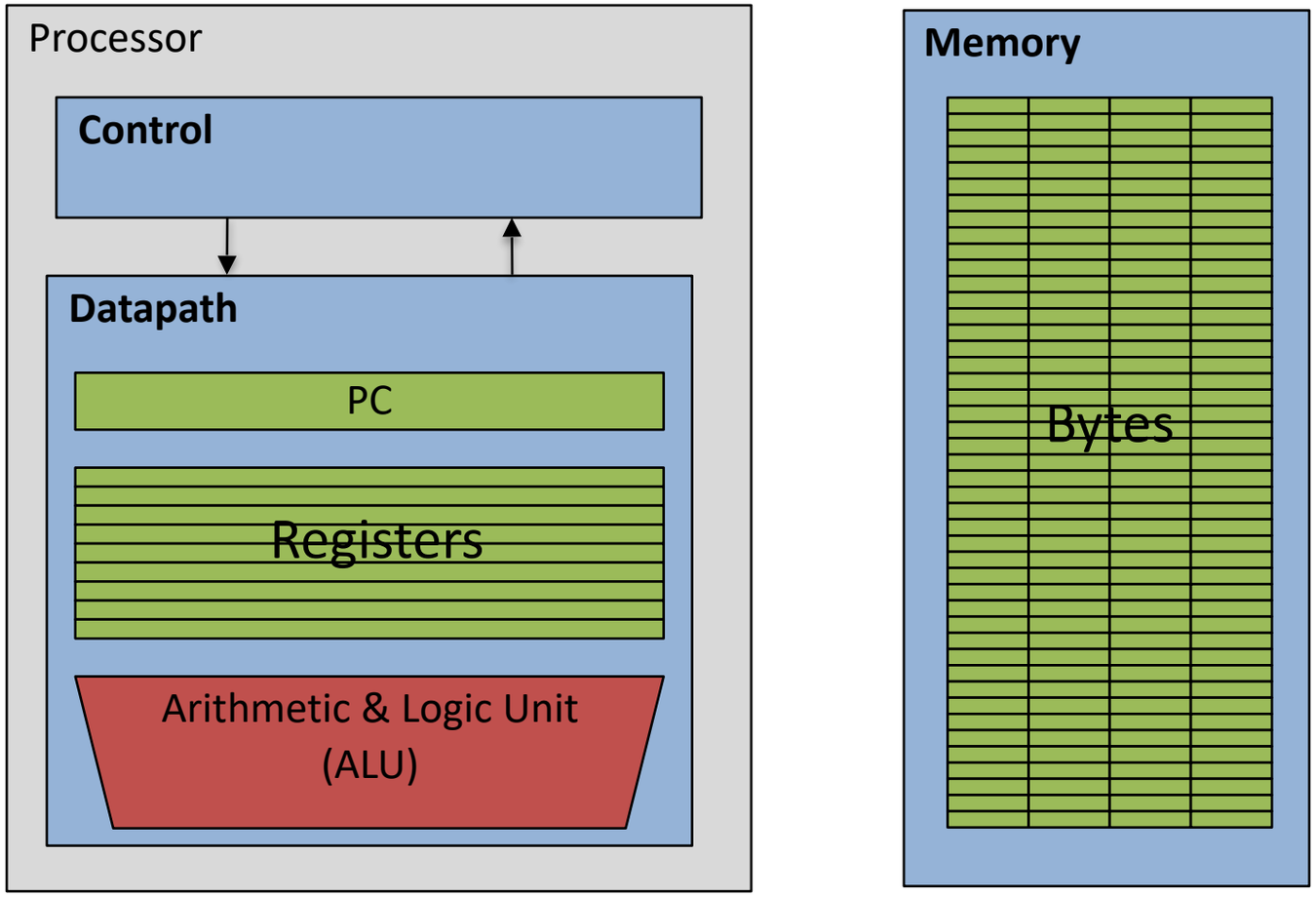
li自动,无需人为考虑

True or False

- If we move all of code, the branch immediate field does not change.

True

Because it utilizes PC-relative addressing/offset



U-Format (Something New)

20	5	7	
imm[31:12]	rd	opcode	U-type

imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode	J-type

```
Load upper immediate: lui rd, imm;
rd = imm.<<12
```

- Can be used to create long immediate to registers along with `addi`
  - Previously, it was 12-bit, e.g., `addi x1, x0, 2047`

```
lui x5, 0xABCDE
+
addi x5, x5, 0x123
```

load immediate

执行时拆为2条

```
li x5, 0xABCDE123
```

A pseudo-instruction

Registers

0xABCDE000	x5
0xABCDE123	x5

U-Format (Something New)

20				5	7	
imm[31:12]				rd	opcode	U-type
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode	J-type

Add upper immediate PC: auipc rd, imm

- rd = PC + (imm. << 12)

```
auipc x5, 0xABCDE
```

0xABCDE000 + PC	x5

- lui opcode: 0b0110111
- auipc opcode: 0b0010111

LUI and AUIPC

- Call function with 32-bit absolute address

```
lui  x5, <hi20bits>
jalr ra, x5, <lo12bits>
```

- Jump PC-relative with 32-bit offset

```
auipc  x5, <hi20bits>
jalr   ra, x5, <lo12bits>
```

- Obtain PC value

```
auipc  x5, 0
```

- Store/load with PC-relative 32-bit offset/32-bit absolute address

```
auipc  x5, <hi20bits>/lui  x5, <hi20bits>
sw/lw  rd, (<lo12bits>)x5
```

Instruction Decoding

- Given an instruction, how to interpret
- Reverse the procedure translating an instruction to machine code
  - Look up `opcode/funct3/funct7` to identify type & operation
  - Find out `rs1/rs2/rd/imm` value, whichever presents
- This is done by hardware in CPU

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] | imm[10:5]	rs2	rs1	funct3	imm[4:1] | imm[11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20] | imm[10:1] | imm[11]		imm[19:12]		rd	opcode	J-type

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			rs		rd	1110011	CSRRW
csr			rs		rd	1110011	CSRRS
csr			rs		rd	1110011	CSRRC
csr			zimm	101	rd	1110011	CSRRWI
csr			zimm	110	rd	1110011	CSRRSI
csr			zimm	111	rd	1110011	CSRRCI

Not in CS110