Introduction to Computer Architecture

RISC-V ISA

R. Pacalet

Telecom Paris / EURECOM

Outline

Section 1

Overview of the RISC-V ISAs

A family of ISAs and extensions

RISC-V base ISAs and some extensions

Name	Description
RV32I	32 bits integer
RV32E	32 bits integer for embedded systems
RV64I	64 bits integer
RV128I	128 bits integer
...	
M	Integer multiplication and division extension
F	Floating point single precision extension
D	Floating point double precision extension
...	

Example: RV32IM: 32 bits base ISA plus multiplications and divisions

Main characteristics

- Open and free, easy to extend and customize
- Load/store architecture (register-register operations)
- Addressing mode: only displacements
  - Experiments on VAX (ISA with all modes):
    80% register indirect with small or 0 displacement
- Three-operands arithmetic and logic instructions
  - Two source registers and a destination register
  - One source register, one immediate operand and a destination register
- Assembly syntax: `op rd,rs1,rs2` or `op rd,rs1,imm`
- Signed integers represented in two's complement
- Little-endian instructions, little-, big- or bi-endian data
- Instructions length: 16-bits (C extension), 32-bits or more (multiple of 16)
- Control flow instructions do not use a control and status register

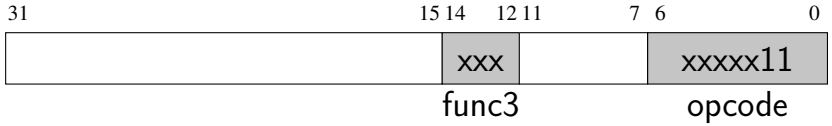
Section 2

The RISC-V 32 bits integer ISA with multiply and divide (RV32IM)

Main characteristics

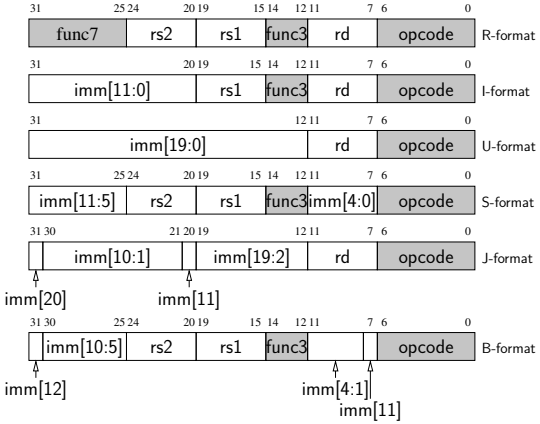
- Thirty-two 32-bits data registers `x0`, ... `x31`
  - `x0` constant: always read 0, write ignored
- 32-bits addresses, byte addressing ⇒ 4GiB address space
- 32 bits instructions
- One extra 32 bits register: Program Counter, `pc`
  - Holds address of current instruction
- Native data types: 1 byte, halfword (2 bytes) and word (4 bytes)
  - Signed and unsigned numbers, pointers (words), characters codes...
- Instructions alignment
  - An instruction uses four consecutive addresses
  - Instructions must be aligned on 32-bits words
  - The 2 LSBs of `pc` shall always be zero
- Data alignment constraints depend on system
  - Unsupported ⇒ exception
  - Allowed but possible performance impact

Instruction structure



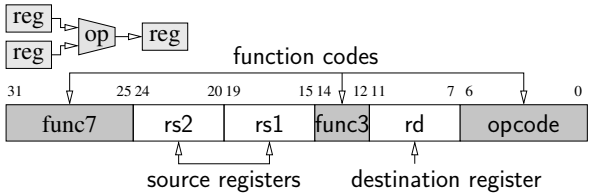
- Binary format
  - All instructions are 32 bits long
  - Bits `[6:0]` and `[14:12]` specify instruction length, format, behavior
  - Meaning of other bits depends on instruction format
- Assembly syntax: instruction keyword followed by operands
- Instructions grouped in categories. Examples:
  - Arithmetic and logic: `add x7,x6,x5` (addition)
  - Memory access: `lw x5,100(x6)` (load word)
  - Conditional branching: `beq x5,x6,comp` (branch to label `comp` if equal)

Instruction formats

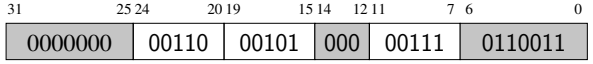


All RV32IM instruction formats

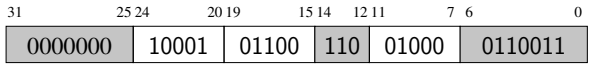
R-format: register-register operations




```
add x7,x5,x6 # x7 ← x5 + x6
```




```
or x8,x12,x17 # x8 ← x12 or x17
```



Example of RV32IM R-format instructions

R-format: addition, subtraction, shift, set, bitwise

R-format base arithmetic/logic operations (`opcode = 0110011`)

Operation		func3	func7	Behavior
`add`	`rd,rs1,rs2`	000	0000000	rd ← rs1 + rs2
`sub`	`rd,rs1,rs2`	000	0100000	rd ← rs1 - rs2
`sll`	`rd,rs1,rs2`	001	0000000	rd ← rs1 << rs2
`slt`	`rd,rs1,rs2`	010	0000000	rd ← (rs1 < rs2) ? 1 : 0
`sltu`	`rd,rs1,rs2`	011	0000000	rd ← (rs1 < rs2) ? 1 : 0 (unsigned)
`xor`	`rd,rs1,rs2`	100	0000000	rd ← rs1 XOR rs2
`srl`	`rd,rs1,rs2`	101	0000000	rd ← rs1 >> rs2
`sra`	`rd,rs1,rs2`	101	0100000	rd ← rs1 >> rs2 (arithmetic)
`or`	`rd,rs1,rs2`	110	0000000	rd ← rs1 OR rs2
`and`	`rd,rs1,rs2`	111	0000000	rd ← rs1 AND rs2

Logical and arithmetic right shifts

⚠ `srl` ≠ `sra`

```
1 addi x5,x0,-1     # x5 <- 0xffffffff
2 addi x6,x0,30     # x6 <- 30
3 srl  x7,x5,x6     # x7 = ?
4 sra  x7,x5,x6     # x7 = ?
```

? Ex. 1: When do `srl` and `sra` produce same result
? Ex. 2: When do `srl` and `sra` produce different results

R-format: multiplication

- Variants with different signs of operands
- Both signed (`mulh`)
- Both unsigned (`mulhu`)
- `rs1` signed and `rs2` unsigned (`mulhsu`)

R-format multiply operations (`opcode = 0110011`)

Operation		func3	func7	Behavior
`mul`	`rd,rs1,rs2`	000	0000001	`rd` ← `rs1 * rs2` (32 LSB)
`mulh`	`rd,rs1,rs2`	001	0000001	`rd` ← `rs1 * rs2` (32 MSB)
`mulhsu`	`rd,rs1,rs2`	010	0000001	`rd` ← `rs1 * rs2` (32 MSB)
`mulhu`	`rd,rs1,rs2`	011	0000001	`rd` ← `rs1 * rs2` (32 MSB)

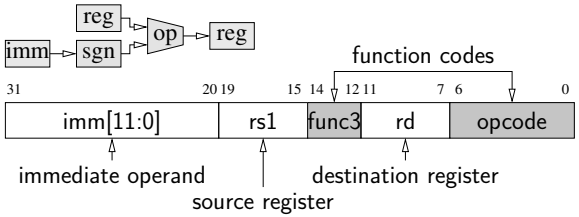
R-format: division, remainder

- Variants with different signs of operands and result
- Both operands signed (`div`, `rem`)
- Both operands unsigned (`divu`, `remu`)
- Sign of `rem` result same as dividend (`rs1`) :::

R-format divide operations (`opcode = 0110011`)

Operation		func3	func7	Behavior
`div`	`rd,rs1,rs2`	100	0000001	`rd` ← `rs1 / rs2`
`divu`	`rd,rs1,rs2`	101	0000001	`rd` ← `rs1 / rs2`
`rem`	`rd,rs1,rs2`	110	0000001	`rd` ← `rs1 mod rs2`
`remu`	`rd,rs1,rs2`	111	0000001	`rd` ← `rs1 mod rs2`

I-format: 12-bits immediate operand




```
addi x7,x6,-121 # x7 ← x6 - 121
```

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0
111110000111	00110	000	00111	0010011

```
ori x8,x12,0x11 # x8 ← x12 or 0x00000011
```

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0
000000010001	01100	110	01000	0010011

Example of RV32IM I-format instructions

I-format: arithmetic/logic operations

- `imm[11:0]`: 12 bits signed immediate operand
- s32: 32 bits sign-extended `imm[11:0]`

I-format arithmetic/logic operations (`opcode = 0010011`)

Operation		func3	Behavior
`addi`	`rd,rs1,imm`	000	`rd` ← `rs1 + s32`
`slti`	`rd,rs1,imm`	010	`rd` ← `(rs1 < s32) ? 1 : 0`
`sltiu`	`rd,rs1,imm`	011	`rd` ← `(rs1 < s32) ? 1 : 0` (unsigned)
`xori`	`rd,rs1,imm`	100	`rd` ← `rs1 XOR s32`
`ori`	`rd,rs1,imm`	110	`rd` ← `rs1 OR s32`
`andi`	`rd,rs1,imm`	111	`rd` ← `rs1 AND s32`

I-format: load

- Transfer data (w/o modifications) from memory to register
- `lb` (`lh`): one byte (half-word), sign-extended to 32 bits
- `lw`: one word (4 bytes)
- `lbu` (`lhu`): one byte (half-word), zero-extended to 32 bits

I-format load operations (`opcode = 0000011`)

Operation	`func3`	Behavior
`lb rd,imm(rs1)`	000	`rd ← SignExtend(mem[rs1 + s32][7:0])`
`lh rd,imm(rs1)`	001	`rd ← SignExtend(mem[rs1 + s32][15:0])`
`lw rd,imm(rs1)`	010	`rd ← mem[rs1 + s32][31:0]`
`lbu rd,imm(rs1)`	100	`rd ← ZeroExtend(mem[rs1 + s32][7:0])`
`lhu rd,imm(rs1)`	101	`rd ← ZeroExtend(mem[rs1 + s32][15:0])`

I-format: shift

- Special case: shift operations with immediate shift amount
  - `imm[11:5]`: used as alternate `func7` to distinguish logic and arithmetic right shifts
  - `u5`: 5 bits unsigned `imm[4:0]` (shift amount)

⚠ `srli` ≠ `srai`

I-format shift operations (`opcode = 0010011`)

Operation	`func3`	`imm[11:5]`	Behavior
`slli rd,rs1,imm`	001	0000000	`rd ← rs1 << u5`
`srli rd,rs1,imm`	101	0000000	`rd ← rs1 >> u5`
`srai rd,rs1,imm`	101	0100000	`rd ← rs1 >> u5` (arithmetic)

I-format: Jump And Link at Register (JALR)

- Control instruction used to call functions (and return)
- `jalr rd,imm(rs1)`
- Stores return address (address of next instruction): `rd ← pc + 4`
- Set `pc ← (rs1 + s32) AND 0xfffffffe`
  - Force LSB to zero as instruction addresses must at least be even
- Can jump anywhere in address space
- Exception if unaligned target address

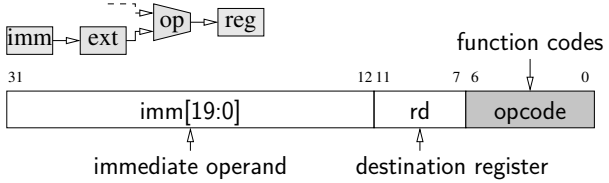
I-format jump and link operation (`opcode = 1100111`)

Operation	`func3`	Behavior
`jalr rd,imm(rs1)`	000	`pc ← (rs1 + s32) AND 0xfffffffe; rd ← pc + 4`

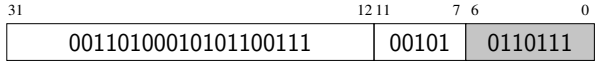
I-format: other instructions

- Access to Control and Status Registers (CSR)
- `ebreak`: debugger
- `ecall`: system calls
- `fence`: memory ordering

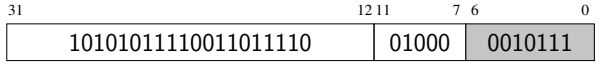
U-format: 20-bits immediate operand




```
lui x5,0x34567 # x5 ← 0x34567000
```




```
auipc x8,0xabcde # x8 ← pc + 0xabcde000
```



Example of RV32IM U-format instructions

U-format: set register content

U-format set-register-content operations

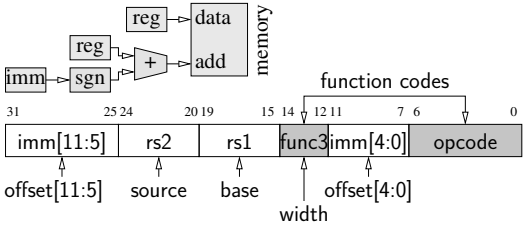
Operation	opcode	Behavior
`lui rd,imm`	0110111	`rd ← imm << 12`
`auipc rd,imm`	0010111	`rd ← pc + (imm << 12)`

- Example of `lui` use to store `0xfedcba98` in register `x5`
  - Two steps: `lui` for 20 MSBs, `addi` for 12 LSBs

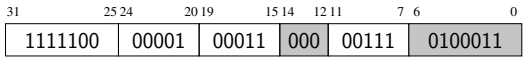
```
lui  x5,0xfedcc
addi x5,x5,0xa98
```

? Ex. 3: Why `0xfedcc` instead of `0xfedcb`

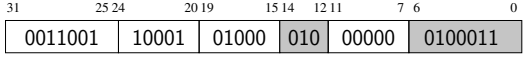
S-format: store with 12-bits immediate operand




```
sb x5,-121(x6) # mem[x6 - 121][7:0] ← x5[7:0]
```




```
sw x8,800(x17) # mem[x17 + 800] ← x8
```



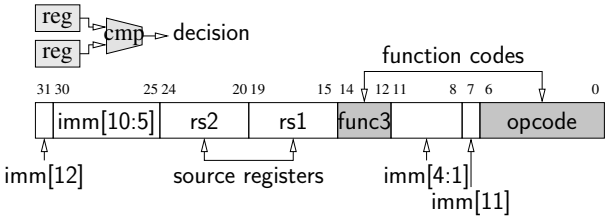
Example of RV32IM S-format instructions

S-format: store

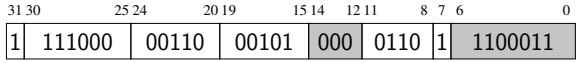
S-format store operations (`opcode = 0100011`)

Operation	func3	Behavior
`sb rs1,imm(rs2)`	000	`mem[rs2 + s32][7:0] ← rs1[7:0]`
`sh rs1,imm(rs2)`	001	`mem[rs2 + s32][15:0] ← rs1[15:0]`
`sw rs1,imm(rs2)`	010	`mem[rs2 + s32][31:0] ← rs1`

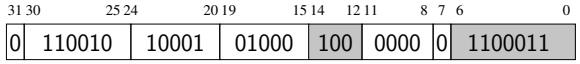
B-format: branch with 12-bits immediate operand




```
beq x5,x6,-122 # pc ← x5 == x6 ?  pc - 244 :  pc + 4
```




```
blt x8,x17,800 # pc ← x8 < x17 ?  pc + 1600 :  pc + 4
```



Example of RV32IM B-format instructions

B-format: conditional branch

- Conditionally select next instruction, no return
- Instruction length multiple of 2 bytes (16 bits) ⇒ `pc + 2 * s32`
- RV32IM 4 bytes (32 bits) instructions ⇒ `pc + 4`
- Signed (unsigned) comparison: `blt` (`bltu`), `bge` (`bgeu`)

B-format conditional branch operations (`opcode = 1100011`)

Operation	`func3`	Behavior
`beq rs1,rs2,imm`	000	`pc ← rs1 == rs2 ? pc + 2 * s32 : pc + 4`
`bne rs1,rs2,imm`	001	`pc ← rs1 != rs2 ? pc + 2 * s32 : pc + 4`
`blt rs1,rs2,imm`	100	`pc ← rs1 < rs2 ? pc + 2 * s32 : pc + 4`
`bge rs1,rs2,imm`	101	`pc ← rs1 >= rs2 ? pc + 2 * s32 : pc + 4`
`bltu rs1,rs2,imm`	110	`pc ← rs1 < rs2 ? pc + 2 * s32 : pc + 4`
`bgeu rs1,rs2,imm`	111	`pc ← rs1 >= rs2 ? pc + 2 * s32 : pc + 4`

B-format branch: example of if/else

```
1 # compute absolute value of x10, result in x10
2 abs:
3   addi x5,x10,x0      # x5 <- x10
4   bge  x5,x0,abs_end  # goto abs_end if x5 >= 0
5   sub  x5,x0,x5       # x5 <- 0 - x5
6 abs_end:
7   addi x10,x5,x0      # x10 <- x5
```

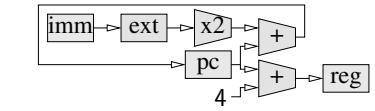
No automatic return mechanism

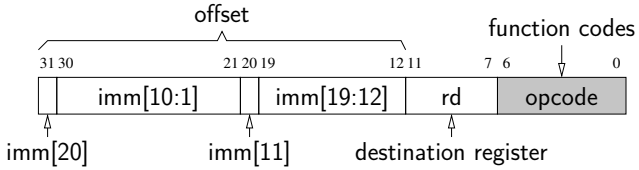
- There is no automatic way to "remember" where we come from
- Do not use for function calls
- Use for control flow (`if/else`, `loops...`)

B-format branch: example of for loop

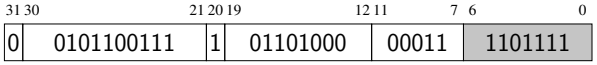
```
1  # compute hamming weight of x10, result in x10
2  hw:
3    addi x5,x0,32          # x5 <- 32
4    addi x6,x0,0           # x6 <- 0
5  hw_loop1:
6    andi x7,x10,1          # x7 <- x10 AND 1
7    add  x6,x6,x7          # x6 <- x6 + x7
8    srli x10,x10,1         # x10 <- x10 >> 1
9    beq  x10,x0,hw_end     # goto hw_end if x10 = 0
10   addi x5,x5,-1          # x5 <- x5 - 1
11   bne  x5,x0,hw_loop1    # goto hw_loop1 if x5 != 0
12 hw_end:
13   addi x10,x6,x0         # x10 <- x6
```

J-format: 20-bits immediate operand






```
jal x5,0x34567 # x5 ← pc + 4; pc ← pc + 0x68ace
```



Example of RV32IM J-format instructions

J-format: jump with 20-bits immediate operand

- The Jump And Link (JAL)
- Control instruction used to call functions (and return)
- `jal rd,imm`
- Stores return address (address of next instruction): `rd` ← `pc + 4`
- Set `pc` ← `pc + 2 * s32`
- Can jump at +/- 1 MB from `pc` in address space
- Exception if unaligned target address

J-format jump-and-link operation (`opcode = 1101111`)

Operation	Behavior
`jal rd,imm`	`rd` ← `pc + 4; pc` ← `pc + 2 * s32`

Section 3

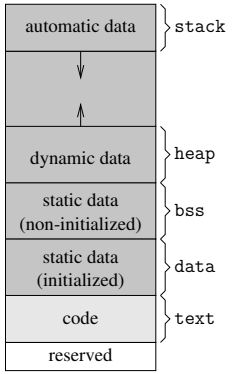
The 32 bits ILP32 ABI for RV32IM ISA (Int, Long, Pointers)

Application Binary Interface (ABI)

- Except `x0` (`zero`) general purpose registers are all the same
  - Could be used interchangeably
- All memory locations are the same
  - Could be used interchangeably
- But... software-software interactions
  - Application - application
  - Application - OS
  - Function - function...
- Example: function parameters, returned values, return address
- Conventions needed: the ABI
  - Memory layout and usage
  - Registers usage
  - Function calls and return...

Memory layout: tool chains organize memory in sections

- `text`: code, instructions
- `data`: initialized static and global variables
- `bss`: non-initialized static and global variables
- `heap`: dynamic memory allocation (malloc, free...)
- `stack`: LIFO, automatically allocated variables (non-static functions locals),...
- `text`, `data`, `bss`: fixed size, known at compile time
- `heap`, `stack`: size grows and shrinks at runtime
- Stack and heap grow towards each other...



Memory usage

Function calls allocate "stack frame" on stack

- Local variables
- Input parameters
- Results
- Stack pointer (register) points to low address of stack

Memory allocation in heap

- Store dynamically received/produced data
- Fragmentation
- Memory management, OS

RV32IM data registers and role according ILP32 ABI

Register	Name	Description	Saver
x0	zero	Constant 0	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer (don't touch)	-
x4	tp	Thread pointer (don't touch)	-
x5-x7	t0-t2	Temporary registers	Caller
x8	s0/fp	Saved register or frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments and return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporary registers	Caller

ILP32 ABI conventions for function calls

Return address and stack pointer

- Return address stored in `ra` by caller
  - ⍰ What if callee calls other functions
  - 👍 Overwrite `ra` ⇒ save `ra` on stack before call
- Stack pointer in `sp`
  - Low address of stack frame, preserved by callee
  - `addi sp,sp,-32` ⇒ allocate a 32 bytes stack frame
  - `addi sp,sp,32` ⇒ free a 32 bytes stack frame

Parameters

- Input arguments passed in `a0-a7`
  - On caller stack frame if `a0-a7` not enough (at `sp`, `sp+4`...)
- Return results in `a0-a1`
  - On caller stack frame if `a0-a1` not enough
  - Address passed in `a0` by caller (new first argument)

ILP32 ABI conventions for function calls

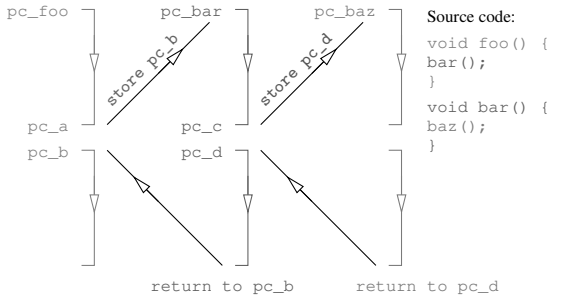
Other

- Temporary registers `t0-t6`
- Saved registers, preserved by callee `s0-s11`
- Pointer to constants and global variables `gp`
- Thread pointer `tp`

Just conventions

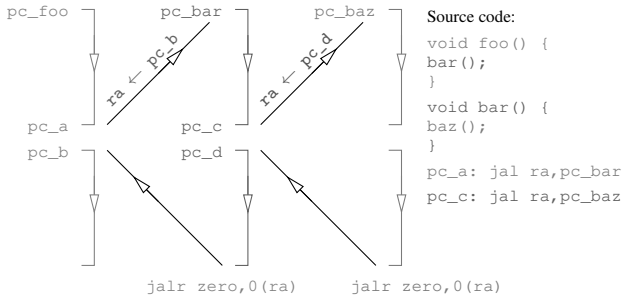
- Almost all these are just conventions
- They ensure good interactions between callers and callees
- Other conventions (ABI) are possible if carefully designed

Principles of function call and return



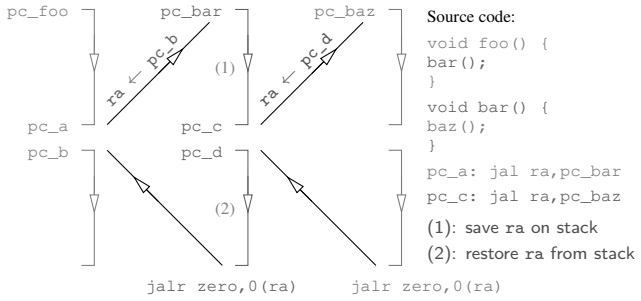
- `jal ra,pc_bar`: return address `pc+4` saved in `ra` and `pc` set to `pc_bar`
- `jalr zero,0(ra)`: jump to address in `ra`, discard `pc + 4`

Principles of function call and return



- `jal ra,pc_bar`: return address `pc+4` saved in `ra` and `pc` set to `pc_bar`
- `jalr zero,0(ra)`: jump to address in `ra`, discard `pc + 4`

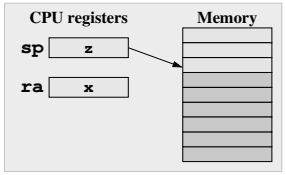
Principles of function call and return



- `jal ra,pc_bar`: return address `pc+4` saved in `ra` and `pc` set to `pc_bar`
- `jalr zero,0(ra)`: jump to address in `ra`, discard `pc + 4`

Example of function call and return

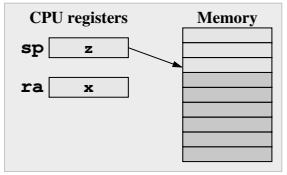
```
foo:  ...
      jal  ra,bar
a:    ...
bar:  addi sp,sp,-12
      sw   ra,0(sp)
      ...
      jal  ra,baz
b:    ...
      lw   ra,0(sp)
      addi sp,sp,12
      jalr zero,0(ra)
baz:  ...
      jalr zero,0(ra)
```



Initial state

Example of function call and return

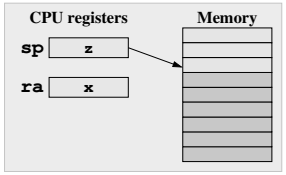
```
1 foo:  ...
      jal  ra,bar
a:    ...
bar:  addi sp,sp,-12
      sw   ra,0(sp)
      ...
      jal  ra,baz
b:    ...
      lw   ra,0(sp)
      addi sp,sp,12
      jalr zero,0(ra)
baz:  ...
      jalr zero,0(ra)
```



1. foo run

Example of function call and return

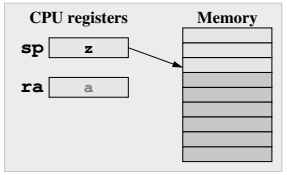
```
1 foo:  ...
2     jal  ra,bar
a:    ...
bar:  addi sp,sp,-12
      sw   ra,0(sp)
      ...
      jal  ra,baz
b:    ...
      lw   ra,0(sp)
      addi sp,sp,12
      jalr zero,0(ra)
baz:  ...
      jalr zero,0(ra)
```



1. foo run
2. foo call bar

Example of function call and return

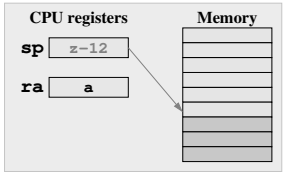
```
1 foo:  ...
2     jal  ra,bar
a:    ...
3 bar:  addi sp,sp,-12
      sw   ra,0(sp)
      ...
      jal  ra,baz
b:    ...
      lw   ra,0(sp)
      addi sp,sp,12
      jalr zero,0(ra)
baz:  ...
      jalr zero,0(ra)
```



1. foo run
2. foo call bar
3. bar increase stack

Example of function call and return

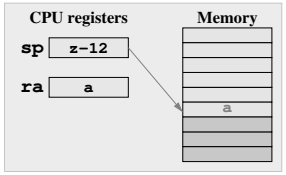
```
1  foo:  ...
2        jal  ra,bar
   a:    ...
3  bar:  addi sp,sp,-12
4        sw   ra,0(sp)
         ...
         jal  ra,baz
   b:    ...
         lw   ra,0(sp)
         addi sp,sp,12
         jalr zero,0(ra)
   baz:  ...
         jalr zero,0(ra)
```



1. foo run
2. foo call bar
3. bar increase stack
4. bar push ra on stack

Example of function call and return

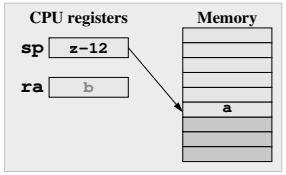
```
1  foo:  ...
2        jal  ra,bar
   a:    ...
3  bar:  addi sp,sp,-12
4        sw   ra,0(sp)
5        ...
         jal  ra,baz
   b:    ...
         lw   ra,0(sp)
         addi sp,sp,12
         jalr zero,0(ra)
   baz:  ...
         jalr zero,0(ra)
```



1. foo run
2. foo call bar
3. bar increase stack
4. bar push ra on stack
5. bar run, call baz

Example of function call and return

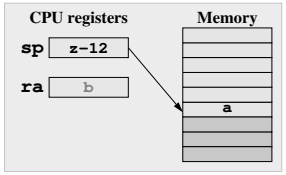
```
1  foo:  ...
2        jal  ra,bar
   a:    ...
3  bar:  addi sp,sp,-12
4        sw   ra,0(sp)
5        ...
         jal  ra,baz
   b:    ...
         lw   ra,0(sp)
         addi sp,sp,12
         jalr zero,0(ra)
6  baz:  ...
         jalr zero,0(ra)
```



1. foo run
2. foo call bar
3. bar increase stack
4. run push ra on stack
5. bar run, call baz
6. baz run

Example of function call and return

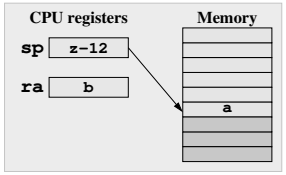
```
1  foo:  ...
2        jal  ra,bar
   a:    ...
3  bar:  addi sp,sp,-12
4        sw   ra,0(sp)
5        ...
         jal  ra,baz
   b:    ...
         lw   ra,0(sp)
         addi sp,sp,12
         jalr zero,0(ra)
6  baz:  ...
7        jalr zero,0(ra)
```



1. foo run
2. foo call bar
3. bar increase stack
4. run push ra on stack
5. bar run, call baz
6. baz run
7. baz return to bar with ra

Example of function call and return

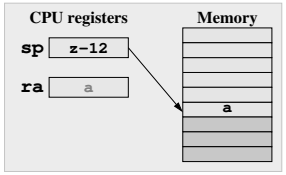
```
1  foo:  ...
2        jal  ra,bar
   a:    ...
3  bar:  addi sp,sp,-12
4        sw   ra,0(sp)
5        ...
         jal  ra,baz
8  b:    ...
         lw   ra,0(sp)
         addi sp,sp,12
         jalr zero,0(ra)
6  baz:  ...
7        jalr zero,0(ra)
```



1. foo run
2. foo call bar
3. bar increase stack
4. bar push ra on stack
5. bar run, call baz
6. baz run
7. baz return to bar with ra
8. bar run, pop return address to ra

Example of function call and return

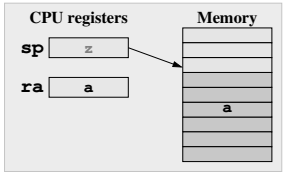
```
1  foo:  ...
2        jal  ra,bar
   a:    ...
3  bar:  addi sp,sp,-12
4        sw   ra,0(sp)
5        ...
         jal  ra,baz
8  b:    ...
         lw   ra,0(sp)
9        addi sp,sp,12
         jalr zero,0(ra)
6  baz:  ...
7        jalr zero,0(ra)
```



1. foo run
2. foo call bar
3. bar increase stack
4. bar push ra on stack
5. bar run, call baz
6. baz run
7. baz return to bar with ra
8. bar run, pop return address to ra
9. bar shrink stack

Example of function call and return

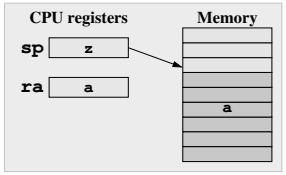
```
1  foo:  ...
2        jal  ra,bar
   a:    ...
3  bar:  addi sp,sp,-12
4        sw   ra,0(sp)
5        ...
         jal  ra,baz
8  b:    ...
         lw   ra,0(sp)
9        addi sp,sp,12
10       jalr zero,0(ra)
6  baz:  ...
7        jalr zero,0(ra)
```



1. foo run
2. foo call bar
3. bar increase stack
4. bar push ra on stack
5. bar run, call baz
6. baz run
7. baz return to bar with ra
8. bar pop return address to ra
9. bar shrink stack
10. bar return to foo with ra

Example of function call and return

```
1  foo:  ...
2        jal  ra,bar
11 a:    ...
3  bar:  addi sp,sp,-12
4        sw   ra,0(sp)
5        ...
         jal  ra,baz
8  b:    ...
         lw   ra,0(sp)
9        addi sp,sp,12
10       jalr zero,0(ra)
6  baz:  ...
7        jalr zero,0(ra)
```



1. foo run
2. foo call bar
3. bar increase stack
4. bar push ra on stack
5. bar run, call baz
6. baz run
7. baz return to bar with ra
8. bar run, pop return address to ra
9. bar shrink stack
10. bar return to foo with ra
11. foo run

ILP32 ABI for this course (no arguments/results in stack)

In function header

- Allocate a stack frame: `addi sp,sp,-N`
  - Constraint: `N` multiple of 16
  - Stack frame addresses: from `sp` to `sp+N-1`
- Save `ra` in stack frame: `sw ra,0(sp)`
- If function will modify saved registers (e.g., `s0`), store them first in stack frame
  - `sw s0,4(sp)...`
- `N` value depends on number of registers stored in stack frame

In function footer

- Restore saved registers from stack frame: `lw s0,4(sp)...`
- Restore `ra` from stack frame: `lw ra,0(sp)`
- De-allocate stack frame and restore `sp`: `addi sp,sp,N`
  - Same `N` as in header

ILP32 ABI for this course (no arguments/results in stack)

In function body

⚠ Modify only non-saved registers or saved registers stored in stack frame

- Before calling another function (including itself - recursive programming)
  - ⚠ Non-saved registers not preserved accross function calls
  - 👍 Save non-saved registers **in use** in saved registers or in stack frame
    ```
    add s0,zero,t2
    sw t5,8(sp)...
    ```
  - Prepare arguments in `a0...a7`
    ```
    addi a0,zero,17...
    ```
- After call
  - Returned value found in `a0` (and `a1`)
  - Restore non-saved registers in use from saved registers or stack frame
    ```
    addi t2,zero,s0
    lw t5,8(sp)...
    ```


Section 4

Extra assembly language features

Pseudo-instructions

- Consecutive instructions at consecutive memory addresses (4-bytes step)
- Except for pseudo-instructions
  - Assembled as true instructions
  - Or as sequences of true instructions

Assembly	Address	Code
`add a0,a0,a1`	0x00400000	0x00b50533
`add a2,a2,a3`	0x00400004	0x00d60633
`sub a0,a0,a2`	0x00400008	0x40c50533
`jalr zero,0(ra)`	0x0040000c	0x00008067

Pseudo-instructions

- Consecutive instructions at consecutive memory addresses (4-bytes step)
- Except for pseudo-instructions
  - Assembled as true instructions
  - Or as sequences of true instructions

Assembly	True instruction
`bgez a0,myFunc_end`	`bge x10,x0,0x00000010`
`not a0,a0`	`xori x10,x10,0xffffffff`
`nop`	`addi x0,x0,0`
`ret`	`jalr x0,0(x1)`

Pseudo-instructions

- Consecutive instructions at consecutive memory addresses (4-bytes step)
- Except for pseudo-instructions
  - Assembled as true instructions
  - Or as sequences of true instructions

Assembly	True instructions
`la a1,myFunc`	`auipc x11,0`
	`addi x11,x11,0xfffffff0`

Directives for data allocations and symbols definitions

Symbols, data values

Implementation of static variable: label, allocation at current position

```
1 .eqv    BIG8 0xff          # define symbol
2 .eqv    HI "Hello world"   # define symbol
3 .byte   BIG8,-10,0x55      # store byte values
4 .half   0xffff,0xaaaa      # store aligned half-words values
5 .word   0xffffffff         # store aligned word values
6 .asciz "dead beef"         # store string with null termination
7 .ascii  HI                 # store string without null termination
```


Directives for alignments, spacing, section selection

Alignments, spacing

```
1 mySpace:
2 .align 2 # align next data item on 2^2 (=4) bytes boundary
3 .space 7 # allocate 7 consecutive bytes
```

Switch between data and text sections

- Optional address

```
1 .data 0x10010800 # next items in data section at given address
2 .byte 0xaa
3 .text            # next items stored in code section
4 add t0,t2,s6
```

Section 5

Examples of C snippets and corresponding RV32IM assembly

Examples of C code and corresponding RV32IM assembly code

C code (comments: // ...)

```
1 // g, h, i, j in  a0=x10, a1=x11, a2=x12, a3=x13
2 int foo(int g, int h, int i, int j) {
3   int f; // in a0=x10
4   f = (g + h) - (i + j);
5   return f;
6 }
```

Assembly code (comments: # ...)

```
1 foo:
2   add  a0,a0,a1   # a0 <- g+h
3   add  a2,a2,a3   # a2 <- i+j
4   sub  a0,a0,a2   # a0 <- (g+h)-(i+j)
5   jalr zero,0(ra) # return at 0+ra, discard pc+4
```


Example of C code and corresponding RV32IM assembly code

Assembly code

```
1 foo:
2   add  a0,a0,a1   # a0 <- g+h
3   add  a2,a2,a3   # a2 <- i+j
4   sub  a0,a0,a2   # a0 <- (g+h)-(i+j)
5   jalr zero,0(ra) # return at 0+ra, discard pc+4
```

Remarks

- Compiler used the conventions of ILP32 ABI
  - Function arguments passed in registers `a0, ..., a3` (`x10, ..., x13`)
  - Function result returned in register `a0` (`x10`)
  - Function returns at address in `ra` (`x1`)
- `foo:` is a label
  - Used to call function (`jal foo`)
  - Substituted with real address by tool chain

Check your understanding

? Ex. 4: Assembly coding: translate from C to RV32IM assembly

```
1 int acc(void) {
2   int i, sum;
3   sum = 0;
4   for(i = 0; i <= 100; i++) {
5     sum += i;
6   }
7   return sum;
8 }
```

Arrays

C code

```c
// T: base address of array
void bar(int T[10], int h, int i) {
  T[i] = h + T[i];
}
```

Assembly code

```
bar:
   slli a2,a2,2     # i <- i<<2 (= i*4)
   add  a0,a0,a2    # T <- T+i
   lw   t0,0(a0)    # t0 <- T[i]
   add  t0,t0,a1    # t0 <- t0+h
   sw   t0,0(a0)    # T[i] <- t0
   jalr zero,0(ra) # return
```


Control: `if then else`

C code

```c
int baz(int g, int h, int i, int j) {
  int f;
  if (i == j) f = g + h;
  else f = g - h;
  return f;
}
```

Assembly code

```
baz:
  bne  a2,a3,baz_else # if i =! j go to baz_else
  add  a0,a0,a1       # f <- g+h
  jal  zero,baz_end   # go to baz_end
baz_else:
  sub  a0,a0,a1       # f <- g-h
baz_end:
  jalr zero,0(ra)     # return
```


Control: loop

C code

```c
// stock: base address of array
int qux(int i, int j, int k, int stock[10]) {
  while(stock[i] == k) i = i + j;
  return i;
}
```

Assembly code

```
qux:
   slli t0,a0,2       # t0 <- i<<2 (= i*4)
   add  t1,a3,t0      # t1 <- stock+t0
   lw   t2,0(t1)      # t2 <- stock[i]
   bne  t2,a2,qux_end # if stock[i] != k got to qux_end
   add  a0,a0,a1      # i <- i + j
   jal  zero,qux      # goto qux
qux_end:
   jalr zero,0(ra)    # return
```


Wrap-up: a recursive function

? Ex. 5: Code `Fact` in RV32IM according ILP32 ABI (assume no overflow)

```c
unsigned int Fact(unsigned int n) {
  if(n < 3) return n;
  return n * Fact(n - 1);
}
```

Section 6

Conclusion

Further Reading

Not a textbook but almost

- Computer Organization and Design RISC-V Edition
  (The Hardware Software Interface)
  Second Edition, 2021, David A. Patterson and John L. Hennessy
  Available from EURECOM library

Other

- J.L. Hennessy, D.A. Patterson
  - Computer Organization and Design, various editions
  - Computer architecture: a quantitative approach, 2011
- W. Stallings
  - Computer organization and architecture: designing for performance, 2015
- Note: some have French versions
- And a lot more in the library...

Questions?

Section 7

Solutions of exercises

Logical and arithmetic right shifts

- ? Ex. 1: When do `srl` and `sra` produce same result
  - When shifting by zero positions or...
  - ✔ ...when shifted value greater or equal zero (MSB = 0)
- ? Ex. 2: When do `srl` and `sra` produce different results
  - When shifting by non-zero positions and...
  - ✔ ...when shifted value less than zero (MSB = 1)

U-format: set register content

- Example of `lui` use to store `0xfedcba98` in register `x5`
  - Two steps: `lui` for 20 MSBs, `addi` for 12 LSBs

```
lui  x5,0xfedcc
addi x5,x5,0xa98
```

- ? Ex. 3: Why `0xfedcc` instead of `0xfedcb`
  - Because in `addi x5,x5,0xa98`, `0xa98` is negative (MSB = 1)
  - After signed extension to 32 bits: `0xfffffa98`
  - `0xfedcb000 + 0xfffffa98 = 0xfedcaa98`
  - ✔ `0xfedcc000 + 0xfffffa98 = 0xfedcba98`

Ex. 4: Assembly coding

```
1 acc:
2   xor  a0,a0,a0        # sum (a0) <- 0
3   addi t0,zero,101     # t0 <- 101
4   xor  t1,t1,t1        # i (t1) <- 0
5 acc_loop:
6   add  a0,a0,t1        # sum <- sum + i
7   addi t1,t1,1         # i <- i + 1
8   bne  t0,t1,acc_loop  # iterate if not done
9   jalr zero,0(ra)      # return
```

Ex. 5: Code `Fact` in RV32IM according ILP32 ABI (assume no overflow)

```
1  # Argument n passed in a0, return value in a0
2  Fact:
3    addi sp,sp,-16       # allocate 16 bytes stack frame
4    sw   ra,0(sp)        # store ra
5    sw   s0,4(sp)        # store s0
6    addi t0,zero,3       # t0 <- 3
7    bltu a0,t0,FactEnd   # goto FactEnd if a0 < t0
8    addi s0,a0,0         # s0 <- a0
9    addi a0,a0,-1        # a0 <- a0 - 1
10   jal  ra,Fact         # a0 <- Fact(a0)
11   mul  a0,s0,a0        # a0 <- s0 * a0
12 FactEnd:
13   lw   s0,4(sp)        # restore s0
14   lw   ra,0(sp)        # restore ra
15   addi sp,sp,16        # de-allocate stack frame, restore sp
16   jalr zero,0(ra)      # return
```