ECE3700JFA23 Mid RC part1 T2-4

TA: Xu Weiqing

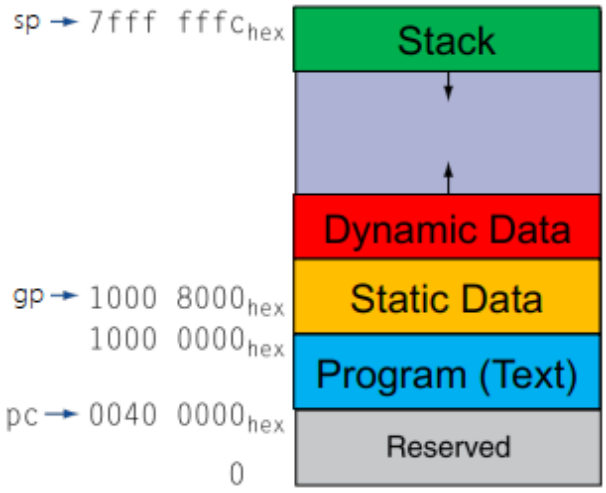
Register

1. RV32: 32×32bits register file
2. operands

- **x0 : the constant value 0 move & clear**
- **x1 (ra) : return address**
- **x2 (sp) : stack pointer**
- x3 (gp) : global pointer
- x4 (tp) : thread pointer
- **x5 – x7 , x28 – x31 : temporaries**
- **x8 : frame pointer**
- **x9 , x18 – x27 : saved registers**

- **x10 – x11 : function arguments/results**
- x12 – x17 : function arguments

Memory



1. PC (program counter)
    a. store the addr. of the instruction to be executed. (like pointer)
    b. PC ← PC + 4
2. Memory layout
    a. stack **x2**: sp 0x7ffffffc going down
    b. dynamic data: heap, going up
    c. static data: global/static variables **x3**: gp 0x10008000
    d. text: program (instructions) **PC**: 0x0040000
3. memory operands:
    a. load: mem → reg
    b. store: reg→mem;
4. **byte addressable** bit/word address?

5. Big & **Little Endian samllest-least**

   0x1020A0B0

	0xffff_0000	0xffff_0001	0xffff_0002	0xffff_0003
0xffff_0000	B0	A0	20	10

6. Integer Array

   Address of Array = Base Address + Offset = Base Address + (index × 4)

   &A[n] = &A[0] + 4n

Instructions

Before all, check the instructions syntax again…

R-type: `and rd, rs1, rs2`

I-type: `ori rd, rs1, imm` , `lw rd, imm(rs1)` , `jalr x0, 0(x1)`

S-type: `sw r2, imm(rs1)`

B-type: `blt rs1, rs2, LABEL`

U-type: `lui rd, imm`

J-type: `jal x1, imm`

As you are all at least familiar with basic instruction writing, just be careful with these things:

1. immediate: **signed constant data** (except _____)
2. Zero extension / signed extension:
   a. shift right: _____ / _____
   b. load: _____ / _____
3. Compare with / without sign: _____ / _____
4. jump instructions
   a. jal

```
jal rd, Label
```

do:

rd ← PC + 4 (rd ← we use x1)

PC ← PC + Imm<<1 (Imm ← we use Function Label)

Before jumping to the function:

```
jal x1, FunctionLabel
```

b. jalr

```
jalr rd, offset(rs1)
```

do:

rd ← PC + 4 (we use x0)

PC ← rs1 + Imm (rs1 ← we use x1, Imm ← offset ← we use 0)

Before leaving the function:

```
jalr x0, 0(x1)
```

Function Call

9 steps

1. parameters → reg x10-x17
2. control → function
3. storage & stack acquire
4. save (push) important reg → stack

5. operating…
6. result → reg x10-x11
7. load (pop) stack → important reg
8. return storage in stack
9. return to the place of function call → reg x1

syntax about stack usage:

```
addi sp, sp, -12
sw x9, 0(sp)
sw x18, 4(sp)
sw x19, 8(sp)
...
lw x19, 8(sp)
lw x18, 4(sp)
lw x9, 0(sp)
addi, sp, sp, 12
```

Caller VS Callee

Caller/Callee; Leaf&Non-leaf

Top principle: Do NOT lose any important data.

1. As a callee, I don’t want to miss up **saved registers** after I’m done.
2. As a caller, I don’t want to lose any important data **not in saved registers** after I call others.

Caller	**Callee**
Save the registers needed after function call Including: its arguments + temporary registers	**Save saved registers to stack before used; Don’t need to save temporary registers**
save return address (Non-leaf functions)	**require stack**
jal	**jalr**

- x10 – x11 : function arguments/results
- x12 – x17 : function arguments
- x5 — x7 , x28 — x31 : temporary registers

- x8 — x9 , x18 — x27 : saved registers

Frame (activation record): what's saved in this function. Check slides T3P29.

Encoding

R	funct7	rs2	rs1	funct3	rd	opcode
	7	5	5	3	5	7
I	Imm[11:0]		rs1	funct3	rd	opcode
	12		5	3	5	7
S	Imm[11:5]	rs2	rs1	funct3	Imm[4:0]	opcode
	7	5	5	3	5	7
B	Imm[11, 9:4]	rs2	rs1	funct3	Imm[3:0, 10]	opcode
	7	5	5	3	5	7
U	Imm[19:0]				rd	opcode
	20				5	7
J	Imm[19, 9:0, 10, 18:11]				rd	opcode
	20				5	7

rs1 / rs2	source register	5 bits	add x9, **x20**, **x21**	19~15, 24~20
rd	destination register	5 bits	add **x9**, x20, x21	11~7
Imm	Immediate	? bits	addi x9, x20, **21**	
opcode	operation code	7 bits	lb vs. addi	6~0
funct3	function code	3 bits	sb vs. sw	14~12
funct7	function code	7 bits	add vs. sub	31~25

R-type	**rs1, rs2 → rd**	`add` `and` `or` `sll` `slt` `sltu` `sra` `srl` `sub` `subw` `xor`	
I-type	**other imm**	`addi` `andi` `jalr` `lb` `lbu` `lw` `lh` `lhu` `ori` `slli` `slti` `sltiu` `srai` `srli` `xori`	0x800~0x7ff, -2048~2047
S-type	**store**	`sb` `sw` `sh`	
U-type	**lui, auipc**	`lui`	
(S)**B-type**	**branch**	`beq` `bge` `blt` `bne` `bgeu` `bltu`	Range: -2^**12** ~ 2^**12** **- 2 (Bytes)**

(U)**J-type**	**jal**	`jal`	Range: -2^**20** ~ 2^**20 - 2 (Bytes)**

What is the range of immediate in I/B/J-type? What is the range of address in some I/J/B-type instructions?

_____-type: Target PC = Current PC + Imm[11:0] * 2 (Bytes)

Design Principle

1. simplicity favors regularity L2P8
2. smaller is faster L2P11
3. make the common case fast L2P27
4. good design demands good compromises L4P9

Meanings. Examples. Reasons.

How to Read RISC-V Reference

Tips

Before midterm:

1. Reviewing lecture notes, especially those that didn't particularly catch your attention during previous study sessions and some knowledge that requires memorization.
2. Quickly skim through previous assignments and double-check the answers, even if you received full marks previously.
3. Go over RC slides and questions if necessary.

During midterm:

1. Clearly articulate your answers, avoiding ambiguity.

Good Luck!

References

ECE3700JFA23 lecture slides T2-4

ECE3700JFA23 RC1-3 slides