Computer Systems Design

Lesson 7

Basics of Assembly programming

Alexander Antonov, Assoc. Prof., ITMO University

Hangzhou, 2025

ITMO

Outline the lesson

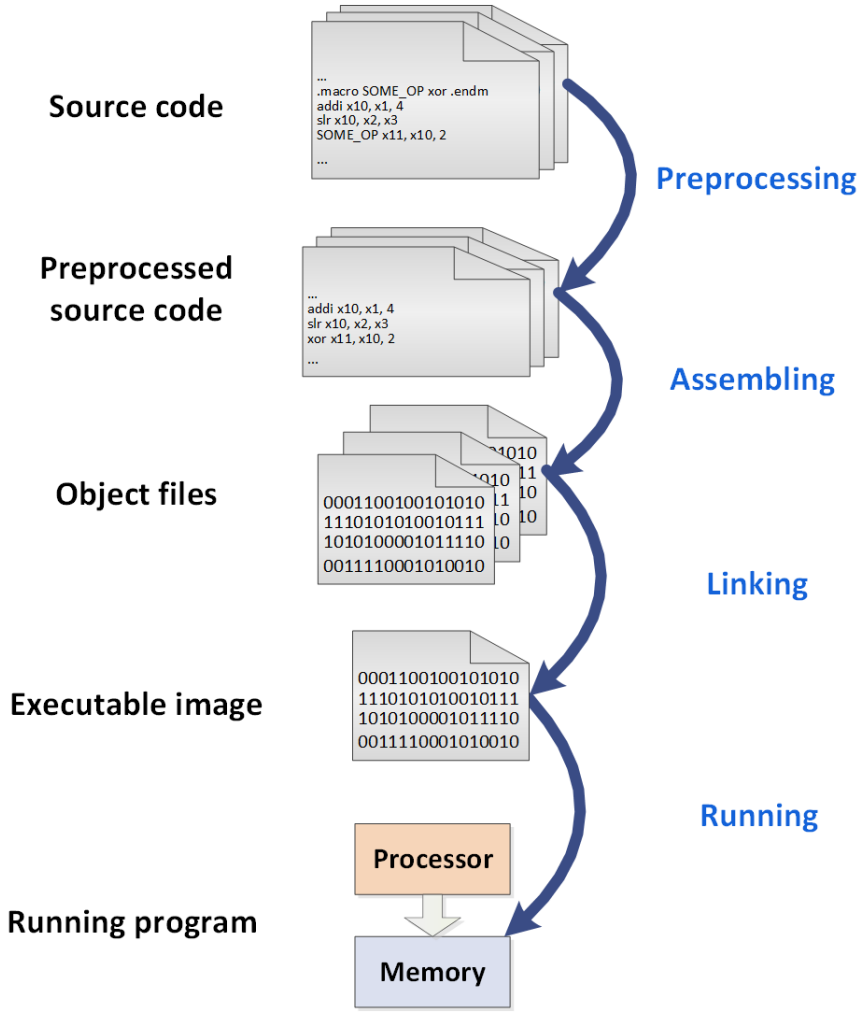
- Instructions and pseudo-instructions
- C vs. ASM idioms
  - constants and variables
  - branching, loops
  - procedures
  - callbacks
- Inline assembly

ITMO

Assembly language

Assembler languages: languages composed of mnemonic (textual) representations of processor instructions

- Specific assembler is tied to certain processor ISA
- Single assembler command represents one or several processor instructions
- No complex optimizations are implemented
- All binary code should be unambiguously understood ("decoded") by the processor



Pseudo-instructions

Pseudo-instructions: assembly instructions without exclusive correspondence to machine instructions

Simplify manual assembly programming

`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if ≠ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if ≠ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned

Table 25.2: RISC-V pseudoinstructions.

pseudoinstruction	Base Instruction	Meaning
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, 0(rs)`	Jump register
`jalr rs`	`jalr x1, 0(rs)`	Jump and link register
`ret`	`jalr x0, 0(x1)`	Return from subroutine
`call offset`	`auipc x1, offset[31 : 12] + offset[11]` `jalr x1, offset[11:0](x1)`	Call far-away subroutine
`tail offset`	`auipc x6, offset[31 : 12] + offset[11]` `jalr x0, offset[11:0](x6)`	Tail call far-away subroutine
`fence`	`fence iorw, iorw`	Fence on all memory and I/O
`rdinstret[h] rd`	`csrrs rd, instret[h], x0`	Read instructions-retired counter
`rdcycle[h] rd`	`csrrs rd, cycle[h], x0`	Read cycle counter
`rdtime[h] rd`	`csrrs rd, time[h], x0`	Read real-time clock
`csrr rd, csr`	`csrrs rd, csr, x0`	Read CSR
`csrw csr, rs`	`csrrw x0, csr, rs`	Write CSR
`csrs csr, rs`	`csrrs x0, csr, rs`	Set bits in CSR
`csrc csr, rs`	`csrrc x0, csr, rs`	Clear bits in CSR
`csrwi csr, imm`	`csrrwi x0, csr, imm`	Write CSR, immediate
`csrsi csr, imm`	`csrrsi x0, csr, imm`	Set bits in CSR, immediate
`csrci csr, imm`	`csrrci x0, csr, imm`	Clear bits in CSR, immediate
`frcsr rd`	`csrrs rd, fcsr, x0`	Read FP control/status register
`fscsr rd, rs`	`csrrw rd, fcsr, rs`	Swap FP control/status register
`fscsr rs`	`csrrw x0, fcsr, rs`	Write FP control/status register
`frrm rd`	`csrrs rd, frm, x0`	Read FP rounding mode
`fsrm rd, rs`	`csrrw rd, frm, rs`	Swap FP rounding mode
`fsrm rs`	`csrrw x0, frm, rs`	Write FP rounding mode
`frflags rd`	`csrrs rd, fflags, x0`	Read FP exception flags
`fsflags rd, rs`	`csrrw rd, fflags, rs`	Swap FP exception flags
`fsflags rs`	`csrrw x0, fflags, rs`	Write FP exception flags

Table 25.3: RISC-V pseudoinstructions.

ISA spec, Vol. 1, unprivileged spec, Chapter 25

ITMO

C vs. ASM: constant initialization

“Short” constant

```
unsigned int some_var = 134;
```

```
li  a4,134   [addi a4, x0, 134]
```

“Long” constant (longer than 12-bit imm in I-type instruction)

```
unsigned int some_var = 3489110677; // 0xcff79a95
```

```
lui    a5,0xcff7a    # constructing 0xcff79... part
addi   a5,a5,-1387   # constructing 0x.....a95 part
```

ITMO

C vs. ASM: working with main memory

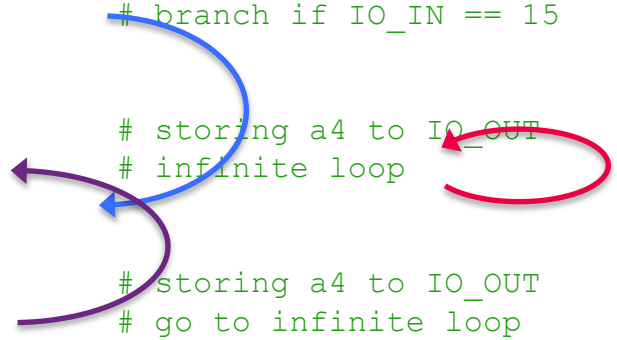
```c
// memory-mapped registers at addresses 0x80000000 and 0x80000004
#define IO_IN          (*(volatile unsigned int *)(0x80000000))
#define IO_OUT         (*(volatile unsigned int *)(0x80000004))

IO_OUT = IO_IN + 211;
```

```
lui a4,0x80000      # writing high part of IO_* address to a4
lw  a5,0(a4)        # loading IO_IN memory cell to a5

addi a5,a5,211      # adding 211
sw  a5,4(a4)        # storing a5 to IO_OUT memory cell
```

C vs. ASM: branching

```c
// memory-mapped registers at addresses 0x80000000 and 0x80000004
#define IO_IN          (*(volatile unsigned int *)(0x80000000))
#define IO_OUT         (*(volatile unsigned int *)(0x80000004))

if (IO_IN == 15) {
    IO_OUT = IO_OUT + 10;
} else {
    IO_OUT = IO_OUT + 20;
}
while(1) {} // infinite loop
```

```
518:    lui  a5,0x80000
51c:    lw   a3,0(a5)        # loading IO_IN to a3
520:    li   a4,15
524:    beq  a3,a4,538       # branch if IO_IN == 15
528:    lw   a4,4(a5)
52c:    addi a4,a4,20
530:    sw   a4,4(a5)        # storing a4 to IO_OUT
534:    j    534             # infinite loop
538:    lw   a4,4(a5)
53c:    addi a4,a4,10
540:    sw   a4,4(a5)        # storing a4 to IO_OUT
544:    j    534             # go to infinite loop
```

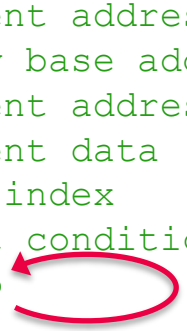


ITMO

C vs. ASM: arrays and loops

```c
#define ARR_SIZE 256

int array [ARR_SIZE];

for (int i=0; i<ARR_SIZE; i++) {
    array[i] = i;       // initializing array with counter values
}
while(1) {} // infinite loop
```

```
 51c:    li    a5,0                # writing index of element to a5
 520:    li    a3,256              # writing array size to a3
 524:    slli  a4,a5,0x2           # writing element address offset to a4
 528:    addi  a2,sp,<array addr>  # writing array base address to a2
 52c:    add   a4,a2,a4            # writing element address to a4
 530:    sw    a5,0(a4)            # writing element data
 534:    addi  a5,a5,1             # incrementing index
 538:    bne   a5,a3,524 <main+0xc> # checking exit condition
 53c:    j     53c <main+0x24>     # infinite loop
```



ITMO

Procedures

Procedure: reusable block of code than can be called from multiple locations

Typically has ***input arguments*** and ***output value***

```
int FindMax(int src1, int src2) {      // procedure (callee)
    if (src1 > src2) return src1;
    else return src2;
}

int DoSmth1()                          // parent procedure (caller 1)
{
    IO_OUT = FindMax(1, 2);            // procedure call
    IO_OUT = FindMax(24, 7);           // procedure call
    IO_OUT = FindMax(19, 138);         // procedure call
    // some other code ...
}

int DoSmth2()                          // parent procedure (caller 2)
{
    IO_OUT = FindMax(14, 15);          // procedure call
    // some other code ...
}
```

ITMO

Recall: Application Binary Interface (ABI)

Convention of register usage by programs

Defines ***recommended*** role of general-purpose registers for procedure calls, stacks, etc.

Followed by programmers and compiler developers

ISA spec, Vol. 1, unprivileged spec, p. 137

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

ITMO

“Jump and Link” instructions

To return from functions ***right after the calling point***, this location has to be saved

“Jump and link” instructions allow to save PC+4 in register (to know where to return when procedure finished)

Jump and link (JAL)

```
jal rd, <procedure offset>
```

Performs:

- rd <- PC + 4
- PC <- PC + procedure offset (20 bits)

Jump and link register (JALR)

```
jal rd, rs, <procedure offset>
```

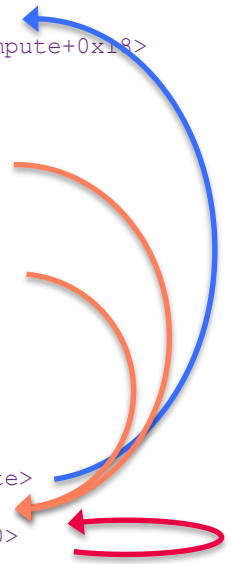
Performs:

- rd <- PC + 4
- PC <- rs + procedure offset (12 bits)

C vs. ASM: procedures

```c
#define IO_IN0          (*(volatile unsigned int *)(0x80000000))
#define IO_IN1          (*(volatile unsigned int *)(0x80000004))
#define IO_OUT          (*(volatile unsigned int *)(0x80000008))

int Compute(int x0, int x1) {       // procedure (callee)
    if (x0 > 10) return (x0 >> 3) | (x1 + 456);
    else return (x0 << 2) & (x1 - 211);
}

int main()                                      // procedure (caller)
{
    IO_OUT = Compute(IO_IN0, IO_IN1);
    while(1) {} // infinite loop
}
```

```
000002d8 <Compute>: ## procedure: callee
 2d8:  00a00793            li     a5,10
 2dc:  00a7da63            bge    a5,a0,2f0 <Compute+0x18>        # branching depending on (x0 > 10)
 2e0:  1c858593            addi   a1,a1,456
 2e4:  40355513            srai   a0,a0,0x3
 2e8:  00b56533            or     a0,a0,a1
 2ec:  00008067            ret                                    # returning to caller
 2f0:  f2d58593            addi   a1,a1,-211
 2f4:  00251513            slli   a0,a0,0x2
 2f8:  00b57533            and    a0,a0,a1
 2fc:  00008067            ret                                    # returning to caller

00000540 <main>:    ## procedure: caller
 540:  80000437            lui    s0,0x80000                      # saving IO_* base address to s0
 544:  00442503            lw     a0,0(s0)                        # saving argument 0 to a0
 548:  00042583            lw     a1,4(s0)                        # saving argument 1 to a1
 54c:  00112623            sw     ra,12(sp)                       # saving return address
 550:  d81ff0ef            jal    ra,2d8 <Compute>                # calling procedure
 554:  00a42023            sw     a0,8(s0)                        # writing return value from a0 to IO_OUT
 558:  0000006f            j      558 <main+0x20>                 # infinite loop
```



ITMO

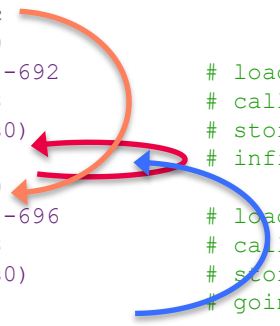
C vs. ASM: callbacks (C)

Sometimes it is useful to pass ***references to functions***

```
int call_function_by_reference(int (call_vector)(void)) {
    return call_vector();
}

int func0() {
    return IO_IN0 + 3;
}

int func1() {
    return IO_IN1 + 5;
}

int main()
{
    if (IO_IN0 == 0) IO_OUT = call_function_by_reference(&func0);
    else IO_OUT = call_function_by_reference(&func1);

    while(1) {}             // infinite loop
}
```

C vs. ASM: callbacks (ASM)

```
000002a4 <func0>:
 2a4:  800007b7            lui    a5,0x80000
 2a8:  0007a503            lw     a0,0(a5)            # loading value at address 0x80000000 to a0
 2ac:  00350513            addi   a0,a0,3
 2b0:  00008067            ret

000002b4 <func1>:
 2b4:  800007b7            lui    a5,0x80000
 2b8:  0047a503            lw     a0,4(a5)            # loading value at address 0x80000004 to a0
 2bc:  00550513            addi   a0,a0,5
 2c0:  00008067            ret

000002f8 <call_function_by_reference>:
 2f8:  00050313            mv     t1,a0
 2fc:  00030067            jr     t1                  # calling passed function reference

00000540 <main>:
 540:  ff010113            addi   sp,sp,-16
 544:  00812423            sw     s0,8(sp)
 548:  80000437            lui    s0,0x80000
 54c:  00042783            lw     a5,0(s0)            # loading value at address 0x80000000 to a5
 550:  00112623            sw     ra,12(sp)
 554:  00079c63            bnez   a5,56c
 558:  00000517            auipc  a0,0x0
 55c:  d4c50513            addi   a0,a0,-692          # loading reference 0x2a4 <func0> to a0
 560:  d99ff0ef            jal    ra,2f8              # calling <call_function_by_reference>
 564:  00a42423            sw     a0,8(s0)            # storing result to IO_OUT
 568:  0000006f            j      568                 # infinite loop
 56c:  00000517            auipc  a0,0x0
 570:  d4850513            addi   a0,a0,-696          # loading reference 0x2b4 <func1> to a0
 574:  d85ff0ef            jal    ra,2f8              # calling <call_function_by_reference>
 578:  00a42423            sw     a0,8(s0)            # storing result to IO_OUT
 57c:  fedff06f            j      568                 # going to infinite loop
```



Inline assembly in C

C and assembly execution models are very close to each other

C and assembly entry can be ***mixed*** in programs

```c
#define IO_IN0          (*(volatile unsigned int *)(0x80000000))
#define IO_IN1          (*(volatile unsigned int *)(0x80000004))
#define IO_OUT          (*(volatile unsigned int *)(0x80000008))

int main() {
    //IO_OUT = Compute(IO_IN, IO_OUT);

    int x0 = IO_IN0;
    int x1 = IO_IN1;
    int y  = 0;

    asm volatile (
        " addi %2, %0, 5 \n\t"
        " xor %2, %2, %1 \n\t"
        " andi %2, %2, 288 \n\t"
        : "=r" (y)
        : "r" (x0), "r" (x1));

    IO_OUT = y;

    while(1) {} // infinite loop
}
```

```
00000518 <main>:
 518:     lui   a5,0x80000
 51c:     lw    a4,0(a5)        # 80000000
 520:     lw    a3,4(a5)
 524:     addi  a3,a4,5
 528:     xor   a3,a3,a4
 52c:     andi  a3,a3,288
 530:     sw    a4,8(a5)
 534:     j     534 <main+0x1c>
```

Thank you for the lesson!

Alexander Antonov, Assoc. Prof., antonov@itmo.ru

Hangzhou, 2025