2. Instruction Set Architecture

"Ideally, your initial instruction set should be an exemplar, …"

- Instruction set architecture (ISA) defines the interface between the hardware and software
- instruction set is the language of the computer
- RISCV instructions are 32-bits, instruction[31:0]

- RISC-V assembly[1] language notation
  - uses 64-bit registers, 64-bits refer to double word, 32-bits refers to word (8-bits is byte).
  - there are 32 registers, namely *x0-x31*, where *x0* is always zero
  - to perform arithmetic operations (add, sub, shift, logical) data must always be in registers
    - the number of variables in programs is typically larger than 32, hence 'less frequently used' [or those used later] variables are 'spilled' into memory [spilling registers]
    - registers are faster and more energy efficient than memory
    - for embedded applications where code size is important, a 16-bit instruction set exists, RISC-V compressed (e.g. others exist also ARM Thumb and Thumb2)

  - byte addressing is used, little endian (where address of 64-bit word refers to address of 'little' or rightmost byte, [containing bit 0 of word]) so sequential double word accesses differ by 8 e.g. byte address 0 holds the first double word and byte address 8 holds next double word. (Byte addressing allows the supports of two byte instructions)
  - memory contains 2^{61} memory words - using load/store instructions e.g. 64-bits available (bits 63 downto 0, 3 of those bits are used for byte addressing, leaving 61 bits)

  - Program counter register (PC) holds the address of the current instruction being executed by the processor
  - Other Notation:
    - R[*rs1*] refers to contents of register *rs1*
    - M[*addr*] refers to value stored at address *addr* in memory
    - *{imm, 1b'0}* denotes concatenating immediate, *imm*, with one bit of value 0), similarly *{imm, 12b'0}* uses 12 zero bits

- data memory is byte-addressable: Data is 64-bits long (thus address must be incremented by 8 to get next sequential data value)
  - instruction memory is byte-addressable: Instructions are 32-bits long (thus PC is incremented by 4 to get next instruction)
  - byte-addressable was chosen for supporting 16-bit instructions as well.

  - RISC-V is Harvard architecture : single main memory but separate instruction and data caches

Instructions have similar instruction format (rs1,rs2,rd, opcode fields in same position, some parts of immediate in same position*) , same length

opcode fields : *opcode, funct3, funct7*
source registers: *rs1, rs2*
destination register: *rd*
immediate field: imm

---

[1] In 1960s systems software was mostly programmed in assembly e.g. all OS before Unix were programmed in assembly

all quotes from H&P 2018

- *opcode* example: all *XXXi* instructions (such as addi, srli, srai, xori,…) have *opcode* 0010011
- *funct3* example: all *addX* instructions (such as add, addi, addiw, addw), have *funct3* 000
- *funct7* example: shift right logical (*srl*) and arithmetic (*sra*) instructions use *funct7* 000…0 and 010…0 respectively with same opcode and funct3 values (011011 and 101)

- consider the folllowing computation:

  d=b+c-e;

  - using assembly, this could be implemented as follows (variables *b,c,d,e* are assigned to registers *x6,x2,x3,x4* respectively and // separates comments on the right):

    add x5,x6,x2 // *a=b+c* , or x5 = x6 + x2, *a* is a temporary variable stored in x5
    sub x3,x5,x4 // *d=a-e , or x3 = x5 - x4*

- note order of operands: operation *rd, rs1, rs2* where R[*rd] = R[rs1] op R[rs2]* , *rd* is destination register
- These are R type instructions.

- ***R type instructions*** (R for registers) are coded in machine language as:

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5bits	3 bits	5 bits	7 bits

e.g.

R[*rd*] = R[*rs1*] + R[*rs2*]

fields of R instruction include: 10-bits for the function, 5-bits for each register and 7-bits for the opcode. Largely used for arithmetic instructions using registers, e.g.

add x5,x6,x2 is 0000000 00010 00110 000 00101 0110011

- next consider the following computation:

  g = h + A[8] ;

  - A[] is a array of 100 double words, whose starting address or base address is stored in x22.
  - two instructions are required - one to load the array value A[8] and one to perform the addition, namely:

    ld x9, 64(x22) // x9=A[8], 8 is offset but
    // RISC-V is byte addressed so 8x8=64 is offset since
    // there are 8 bytes in each double word .
    // Also x22 is base register

all quotes from H&P 2018

```
add x20,x21,x9  // g,h are stored in registers x20,x21 rest
```

- the load instruction is an I type instruction

- ***I type instructions*** (I for immediate) are coded in machine language as:

immediate[11:0]	rs1	funct3	rd	opcode
12 bits	5bits	3 bits	5 bits	7 bits

e.g.

R[*rd*] = M[R[*rs1*] + *imm*](63:0)

used for loads and immediate arithmetic,
ld x9, 64(x22) is 0000 0100 0000 10110 011 01001 0000011

- note the immediate value 12-bit value has to be sign extended from 12-bits to the 64-bit double word size used in RISC-V, see examples below:
  - sign extension of 12-bit immediate 0000 0100 0000 to 64-bits is
    0000 0000 0000 0000 0000 0000 0000 0000
    0000 0000 0000 0000 0000 0000 0100 0000
  - also note that, sign extension of 1111 1111 1000 to 64-bits is
    1111 1111 1111 1111 1111 1111 1111 1111
    1111 1111 1111 1111 1111 1111 1111 1000

- consider the following computation:

```
A[30] = h + A[30] + 1;
```

  - the following assembly can perform this computation:
  -

```
ld x9, 240(x10)   // x9=A[30] , offset is 30X8=240 due to byte-addressing
add x9,x21,x9   // h is stored in x21
addi x9,x9,1      // adds immediate value, using I type instruction
                  // R[rd]=R[rs1]+imm
sd x9,240(x10)  // stores using S type instruction
```

- the store instruction is a S type instruction

- ***S type instructions*** (S for stores) are coded in machine language as:

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

note the 12-bit immediate is split in order to keep the location of *rs1* and *rs2* fields consistent (to minimize hardware complexity)
e.g.

M[R[*rs1*] + *imm*](63:0) = R[*rs2*](63:0)

used for store doublewords,
sd x9,240(x10) is 0000111 01001 01010 011 10000 0100011

all quotes from H&P 2018

SUMMARY so far:

R type instructions use 3 register operands (2 source registers and one destination register).
I type instructions use 2 register operands (1 source, 1 destination) and one 12-bit immediate.
S type instructions use 2 register operands (2 source) and one 12-bit immediate.
Note how the register fields align, along with the opcode and funct3 fields.

- Addressing can be done using registers, immediate values, immediate + register value, immediate + PC.
- Three other types of instructions are SB, U and UJ type instructions.

- the ***SB type instructions*** (conditional branch, fields like the S) are coded in machine language as:

immediate[12,10:5]	rs2	rs1	funct3	immediate[4:1,11]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

e.g.

if(R[*rs1*] != R[*rs2*]) PC = PC + { *imm*,1b'0}

bne instruction uses PC-relative addressing (PC =PC + immediate), used for branch not equal instruction

bne x10, x11, 2000 // if x10 != x11 , go to location 2000

- the ***U type instructions*** (upper immediate format) are coded in machine language as:

immediate[31:12]	rd	opcode
20 bits	5 bits	7 bits

e.g.

R[*rd*] = PC + { *imm*,12'b0}

instruction *auipc ,* or add upper immediate to PC (used for position independent code, such as for dynamic linked libraries, DLL)

- the ***UJ type instructions*** (unconditional jump, fields like the U) are coded in machine language as:

immediate[20,10:1,11,19:12]	rd	opcode
20 bits	5 bits	7 bits

e.g.

R[*rd*] = PC + 4; PC = PC + { *imm*,1b'0}

instruction *jal,* used for jump and link instruction (where 'link' means it will return to where it was called, by placing address or link into register x11 which holds the return address)- typically used for procedure calls

jal x11, 2000 // go to location 2000

---

all quotes from H&P 2018

- Longer branches can be achieved using unconditional branch instructions in RISC-V

```
beq x10,x0,L1  // uses 12-bit immediate to jump to L1

can be replaced with the two instructions below

bne x10,x0,L2
jal x0,L1          // uses 20-bit immediate to jump to L1
L2: …
```

- in RISC-V, 20-bit immediates, e.g. in jal, and 12-bit immediates, e.g. in conditional branch, are concatenated with a lsb '0' bit using {imm,1b'0} since they are half-word aligned (to support the possibility of 16-bit instructions)

- note the position of immediate data below (similar positions to minimize hardware complexity)

	funct7		rs2		rs1 , funct3	rd		opcode
I	imm[11]	**imm[10:5]**	**imm[4:1]**	imm[0]				opcode
SB	imm[12]	**imm[10:5]**				**imm[4:1]**	imm[11]	opcode
S	imm[11]	**imm[10:5]**				**imm[4:1]**	imm[0]	opcode
U	imm[31]	imm[3**0**:2**5**]	imm[2**4**:2**1**]	imm[20]	**imm[19:12]**			opcode
UJ	imm[20]	**imm[10:5]**	**imm[4:1]**	imm[11]	**imm[19:12]**			opcode

all quotes from H&P 2018

Code examples:

"Compilers frequently create branches and labels where they do not appear in the programming language…"

RISC-V logical operations include shift left/right/right-arithmetic, and/or/not/xor.

Example 1:

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

assume *f* through *j* correspond to values in registers *x19* through *x23*

```
bne x22, x23, Else          // go to Else if i !=j
add x19, x20, x21           // f=g+h
beq x0, x0, Exit            // go to Exit
Else: sub x19, x20, x21     // f=g-h
Exit:
```

Example 2:

```
while (save[i] == k)
                                    i += 1;
```

assume value of *i* is in register *x22* and value of *k* is in register *x24*. Assume base of array *save[]* is in *x25.*

```
loop: slli x10, x22, 3   // x10 = i*8 , slli: R[rd]=R[rs1]<<imm
add x10, x10, x25        // x10 now has address of save[]
ld x9, 0(x10)            // load save[i] into x9
bne x9, x24, Exit        // if save[i] != k go to exit
addi x22, x22, 1         // i++
beq x0, x0, Loop         // go to loop
Exit:
```

Note 2.1. first 3 lines are a 'basic block' == series of instructions / code with no ifs/loops/ branches.

Note 2.2. There are also unsigned branch instructions such as *bltu (branch less than unsigned, if (R[rs1]<R[rs2]) PC=PC+{imm,1b'0})*

Some architectures alternatively set a sign bit (condition codes, flags) in a status register which is then checked for the branch instructions (e.g. ARM architecture), but this may cause more complexity/dependencies during pipelining. Or some architectures may set a value in a temporary register which can be checked for a branch instruction (e.g. MIPS architecture), but this may require more instructions.

2.8 Procedures:

"Procedures…to concentrate on just one portion of the task at a time;…"

By convention in RISC-V:

- **8 parameter registers *x10-x17* in which to pass parameters or return values.**
- **one return address register *x1* holds the return address to return to the point of origin.**

In RISC-V there are special 'procedure' instructions:

- jump and link instruction, *jal* (jump and link, UJ type) for procedures, which branches to an address and saves the address of the following instruction (PC+4) to *rd=x1.*

  jal x1, procedure-address // jump and link
  R[*rd*]=PC+4; PC=PC+{imm,1b'0}

- indirect jump and link instruction, *jalr* (I type), jumps to the address stored in a register, e.g.

  jalr x0, 0(x1) //procedure-address = contents of x1 + imm
  R[*rd*]=PC+4; PC=R[*rs1*]+imm

Note if temporary registers are needed in the procedure, in addition to the parameter variable registers, then registers are spilled/temporarily-stored to memory and must be restored on return from the procedure. Stacks are used to spill registers in RISC-V.

- **the stack pointer is *x2*, also called *sp*.**
- **stack grows from higher to lower addresses.**

So values are pushed onto the stack by decrementing the *sp* and popped by incrementing the *sp*.

EXAMPLE 3.1

```
long long int leaf_example( long long int g, long long int h, long long int i, long long int j)
{
long long int f;

f = (g+h)-(i+j);
return f;
}
```

Assume

```
jal x1, leaf_example          // the caller of the procedure
```

was executed in main program (*x1* is return address register).

Also assume incoming parameters to procedure, specifically variables g,h,i,j , are stored in registers *x10, x11, x12, x13* .
Also assume the procedure will use *x5, x6 and x20* as temporary registers. Thus we must first save the existing *x5,x6 and x20* values into the stack before the procedure computations.

```
leaf_example:              // the called procedure or callee

    // push[save] 3 values onto the stack, since procedure needs these 3 registers
    addi sp, sp, -24
    sd x5, 16(sp)          // ***
    sd x6, 8(sp)           // ***
    sd x20,0(sp)

    // next do procedure computations
    //
```

```
add x5, x10, x11
add x6, x12, x13
sub x20, x5, x6
//
// procedure computations are done
addi x10, x20,0     // returns f in register x10

// pop[restore] three values from the stack
ld x20, 0(sp)
ld  x6, 8(sp)         // ***
ld x5,16(sp)          // ***
addi sp, sp, 24

// now return, recall x1 is return address register, x0 is zero register so can ignore
jalr x0,0(x1)
```

since it took some code to do the register saving (8 instructions, see push and pop), it's helpful to know if these registers need to be saved or not. For example the caller can do some register saving if needed and the callee (or called procedure) can also do some saving of registers if needed. Hence RISC-V separates 19 registers into two groups

1. ***x5-7 and x28-31*** **are not preserved by the callee** (called procedure) on a procedure call. thus **if the caller needs the data in these registers, the caller must push these onto the stack before the call**.
2. ***x8-9 and x18-27*** **are saved and restored by the callee, if used by callee** (these registers must be preserved on a procedure call). the callee pushes return address register x1 and these if to be used inside procedure.

In example above *x5 and x6* need not be saved to stack according to caller. In other words the caller would save registers x5 and x6 if they needed these values, before the procedure is called. Thus the procedure does not have to worry about saving the previous values of these registers (so one can remove the *sd x5,..* , *sd x6,.. , ld x5,…, ld x6,…* (see ***) from the *leaf_example* procedure above), this would save 4 instructions. Register *x20* still needs to be stored/pushed to stack by callee.

Note the term 'leaf' procedures is used when the procedure does not call other procedures. This is unlike the next example which uses a recursive procedure.

EXAMPLE 3.2

Factorial procedure illustrating nested calls (recursion) :

```
long long int fact(long long int n)
{
 if (n<1) return (1);
 else return ( n*fact(n-1) );
}
```

Assume the variable n is stored in register *x10.* The procedure saves two register values onto the stack (the return address and *x10*)

```
fact:
addi sp, sp, -16
```

```
sd x1, 8(sp)
sd x10, 0(sp)

addi x5, x10, -1
bge x5, x0, L1

addi x10,x0,1
addi sp, sp, 16
jalr x0, 0(x1)          // return to caller

L1:
addi x10, x10, -1
jal x1, fact

addi x6, x10, 0         //
ld x10, 0(sp)           //
ld x1, 8(sp)               //
addi sp, sp, 16               //

mul x10, x10, x6        // n*fact(n-1)
jalr x0, 0(x1)          // return to caller
```

Note recursion algorithms typically have larger overheads than iterative algorithms.

Though the following example uses tail recursion algorithm it can be implemented efficiently using an iterative algorithm;

```
long long int sum( long long int n, long long int acc)
{
if (n>0)
    return sum( n-1, acc+n );
else
    return acc;
}
```

Assume *x10 = n* and *x11 = acc* and the result is stored in *x12*.

```
sum:
ble x10, x0, sum_exit
add x11, x11, x10
addi x10, x10, -1
jal x0, sum                 // go to sum

sum_exit:
addi x12, x11, 0
jalr x0, 0(x1)              // return
```

Static variables in C, may be pointed to by a global pointer, ***gp***, by convention register ***x3*,** in some RISC-V compilers. Automatic variables are only local to a procedure.

Static variables and dynamic data structures need space in memory as well.

- Machine code is stored in the **'text segment'** in lower part of memory.
- The static data (constants, global variables, arrays) is stored above the machine code.

- Dynamic data structures (such as linked lists etc) are stored next in the memory section referred to as the **heap**, which grows up in memory.
- The stack is above it, growing down towards heap.

Space on the heap is controlled by the programmer using *malloc()* (which allocates space and returns a pointer to it) and *free()* (which releases space to which the pointer points) in C. This is unlike in Java which automatically does memory allocation and garbage collection (freeing).

Linux uses the following address space shown next:

addresses	*memory contents*
SP => 0000 003f ffff fff0	Stack || \V /\ || Dynamic Data (Heap)
0000 0000 1000 0000	Static Data (constants, global variables)
PC=> 0000 0000 0040 0000	machine code “text segment”
0	reserved

Fig 2.13: RISC-V memory allocation for program and data

Stack is also used to store variables local to a procedure which don’t fit into registers (including local arrays, structures).

> The part of the stack holding a procedure’s variables/registers is called a procedure frame or activation record.

Some RISC-V compilers use a frame pointer, ***fp,*** or register ***x8*** to point to first double word of frame.

> A **frame pointer** is a value denoting the location of saved registers and local variables for a given procedure.

For example if the user has to pass more than 8 parameters as input to the procedure, they’d use registers x10 through x17 and the *fp* to point to the remaining input parameters stored in memory. The *fp* is a stable pointer within the procedure for local memory references. In previous examples only the *sp* was utilized on entry and exit from procedure for simplicity. Stack frame example below showing local data allocated by callee procedure frame used to manage stack storage. The *fp* points to first double word of the frame. The *sp* points to top of stack. The *sp* is adjusted to make room for all saved registers etc. Since the *sp* may change during program execution, it’s easier for procedure to use the more stable *fp* to reference variables.

- the *sp* always points to top of stack
- when the *fp* is used, it is initialized using the address in *sp* on a call and *sp* is restored using *fp*.
- *fp* points to first double word of the frame

	higher memory addresses
	…
	…
FP =>	saved argument registers (if any)
	saved return addresses (if any) saved saved registers (if any)
SP=>	local arrays and structures (if any)
	lower memory addresses

2.9 ASCII

ASCII is typically used to represent characters and is based on a 8-bit representation. Upper and lower case alphabets differ by 32. For example ‘a’ is represented by 0x61 and ‘A’ is represented by 0x41. ASCII includes formatting characters such as null which is 0x00, backspace as 0x08, etc.

Example of loading constants into registers:

load constant 0x 00 00 00 00 00 3D 05 00 into register
using load upper immediate instruction, *lui* and *addi* :

```
lui x19, 976              // 976 is 0x3D0
addi x19, x19, 1280       // 1280 is 0x500
```

Instruction *lui* copies immediate to bits 31:12 of register along with zero bits in 11:0 bits of register, and sign extends to msb e.g.

R[rd] = {32b’imm<31>, imm, 12’b0}

Note if immediate of instruction *addi* had ‘1’ for most significant bit , then it would be signed extended during the addition. To undo the sign extension, a solution would be to add ‘1’ to bit 12 of *lui* immediate (this would remove the effect of the sign extension and negative number).

e.g. constant 0x80 00 00 00 sign extended and added to 2^{12}:
0xFF FF FF FF 80 00 00 00 + 0x00 00 00 01 00 00 00 00 = 0x00 00 00 00 80 00 00 00

2.10 Addressing

PC relative branching used for (un)conditional branching or jumping, supporting branching within +/- 2^{10} words of current instruction (using 12-bit immediate minus 2 bits for byte

addressing) or jump within +/- 2^{18} words of current instruction (using 20-bit immediate minus 2 bits for byte addressing).

For very long jumps to any 32-bit address, a 2 instruction sequence is used, specifically *lui* to write bits 12 through 31 to a temporary register and *jalr* to add the lower 12-bits of address to the temporary register and jump to the sum.
Additionally longer branching distances can be used by choosing different instruction formats such as:

```
beq x10, x0, L1     // beq supports only 12-bit immediate
```

can be replaced by

```
bne x10, x0, L2
jal x0, L1          // jal supports larger 20-bit immediate for longer branching distance
L2: …
```

Addressing (of operand/instruction) modes:

- immediate addressing:
  - operand is constant within instruction, e.g. *addi*
- register addressing:
  - operand is a register, e.g. *add*
- base or displacement addressing:
  - operand is at memory location whose address is sum of register and constant in instruction
- PC-relative addressing:
  - branch address is sum of PC and a constant in instruction

2.11 Synchronization

Parallelism needs synchronization. For example a task must know when another task is completed writing data so it is safe for the first task to read the data. Thus tasks need to synchronize to prevent problems from occurring such as data races. Synchronization is typically built using software routines that rely on hardware-supplied synchronization instructions. Lock and unlock synchronization operations can create regions where only a single processor can operate, called mutual exclusion. In order to support synchronization in multiprocessors a set of hardware primitives with the ability to atomically read and modify a memory location is needed.

- **In processors the ability to being able to uninterruptedly read and modify a memory location is necessary (this is referred to as atomically read and modify).**

Normally the ability to atomically read and modify a location along with a check to ensure the read and write were performed atomically is provided in the hardware primitives.

For example an atomic swap (or exchange) operation is required. This operation interchanges a value in a register for a value in memory. If two processors try to do an atomic swap, the hardware will only allow one to do it at time.
Two instructions in RISC-V can accomplish this

- load-reserved double word (*lr.d*)
- store-conditional double word (*sc.d*)

If the contents of memory location specified by *lr.d* are changed before the store-conditional to same address then the *sc.d* fails and does not write to memory. If *sc.d* succeeds the contents

of a register are set to 0 else some non-zero value. Below the contents of *x23* and memory location specified by address in *x20* are atomically exchanged.

```
again:
lr.d x10, (x20)          // R[x10]=M[ R[x20] ] reserved
sc.d x11,x23, (x20)      // if reserved, M[ R[x20] ] = R[ x23], R[x11]=0, else R[x11]=1
bne x11,x0, again
addi x23, x10, 0         // R[x23]=R[x10]
```

2.12 Tools

Compiler
(input C program[2], e.g. filename.c => filename.i, filename.i => filename.s)
=> assembler
(input assembly program e.g. filename.s=> filename.o)
=> linker
(input: object-machine language module output from assembler, filename.o,
and object-library routine (machine language), e.g. dynamically linked library routines is x.so)
(e.g. filename.o => a.out , a.out is default executable filename, originally referring to 'assembler.out')
or Linux ELF (executable and linking format)…
=> loader (input executable machine language program) (e.g. ./a.out)

Note there are some assembly **pseudoinstructions** supported by the RISC-V assembler such as:
li x9, 123 which is equivalent to *addi x9, x0, 123* ; also *mv x10,x11* equivalent to *addi x10,x11,0* ; also *and x9,x10,15* equivalent to *andi x9,x10,15.*

To produce binary machine code the labels in assembly must be converted to addresses. This is performed with a **symbol table** (symbol, address).

The object file for Unix consists of six parts:

- object file header:
  - size and position of other pieces in file;
- text segment:
  - machine language code
- static data segment:
  - data allocated for life of program (static is allocated for throughout the program, dynamic can grow or shrink as needed)
- relocation information:
  - identifies instructions/data dependent on absolute addresses when program is loaded into memory
- symbol table:
  - remaining undefined labels (external references)
- debugging information:
  - info on how modules were compiled for debugger (associating machine instructions with C source files for readable data structures etc)

Executable file output is same format as object file except generally there are no unresolved references. However it is possible to have partial linked files such as library routines with unresolved addresses, thus resulting in object files.

---

[2] open source gnu C compiler, gcc, in 1987.

Linker[3] resolves all undefined labels using relocation information and symbol table in each object module. These references occur in branch instructions and data addresses.
Loader has the operating system read it to memory (instructions and data to memory, parameters to stack, sets *sp*) and start the program (branches to startup routine to copy parameters into argument registers and calls main routine). On return from the main program, an exit system call terminates the program.

Consider example below where procedure A needs address of data A and address of procedure B.

object file header	name text size data size	**procedure A** 0x100 0x20	
text segment	address	instruction	
	0 4 …	ld x10, 0(x3) jal x1, 0 …	ld X call procedure B
data segment			
	0 …	X …	
relocation information	address	instruction type	dependency
	0 4	ld jal	X B
symbol table	label	address	
	X B		
object file header	name text size data size	**procedure B** 0x200 0x30	
text segment	address	instruction	
	0 …	sd x11, 0(x3) …	sd Y
data segment			
	0 …	Y …	
relocation information	address	instruction type	dependency

[3] linking used to be done manually in the 1940s

	0 …	sd …	Y …
symbol table	label Y	-	

Executable file is shown below (assuming register x3 is initialized to 0x0000 0000 1000 0000):

	text size data size	0x300 0x50
text segment	address	instruction
	0000 0000 0040 0000 0000 0000 0040 0004 … 0000 0000 0040 0100 …	ld x10, 0(x3) jal x1, 252_{10} … sd x11, 32_{10}(x3) ….
data segment	address	
	0000 0000 1000 0000 … 0000 0000 1000 0020 …	X value … Y value …

As shown on page 10 (text page 106) in Fig 2.13, text segment starts at address 0x0000 0000 0040 0000 and static data segment starts at 0x0000 0000 1000 0000. Since procedure_A requires text size of 0x100 bytes (as determined in its object file header), procedure_B code can start at address 0x0000 0000 0040 0100. Since procedure_A's data size is 0x20 bytes, procedure_B's data starts at address 0x0000 0000 1000 00020. This requires procedure_B's *sd* instruction to use immediate value of 32 (0x20).

Dynamically linked libraries (DLL)

Libraries can be statically or dynamically linked[4]. Statically linked libraries have disadvantages:
- library routines are part of the executable, so newer libraries releases are not incorporated automatically
- all routines called anywhere in executable are loaded, even if they are not executed (e.g. error routines may never be called …)

Libraries can be large, for example standard C library on RISC-V running Linux is 1.5MB.

Dynamic linked libraries are not linked and loaded until the program is run. To avoid linking all called library routines, which may be again be very large, a lazy procedure linkage approach is used.
Using the lazy procedure, a library routine is linked only after it is called. This works by using levels of indirection. The program calls a dummy entry and indirectly branches to code that places a number in a register to identify the library routine. Then it branches to the dynamic

[4] Since 1980s most systems support shared libraries and dynamic linking

linker/loader which finds the routine and remaps it changing the address and indirect branch location to point to that routine address whose data holds the routine address
e.g. it resolves the actual address of the routine

jalr sets *PC= R[rs1]+imm,* hence the value loaded into register *rs1* is updated so it can jump to the routine directly next time it's called.

It then branches to it. When it's finished executing the routine, it returns to the calling site and on the next call to that routine the code jumps indirectly to the routine without any extra overhead. The overhead is mainly associated with the first time a routine is called since afterwards only a single indirect branch is needed.

2.13 **SUMMARY**

- Implement assembly code for the following C routine, swap (note: size_t is unsigned integer type returned by sizeof operator):

```
void swap( long long int v[ ], size_t k)
{
        long long int temp;

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

```
Assembly
// assume v and k are passed in x10,x11
// inside routine temp variable is stored in x5
// also registers x6,x7 are used
// x5-7 do not need to be preserved inside
// routine by convention
swap:
  slli x6,x11,3     // reg x6 = k * 8
  add  x6,x10,x6    // reg x6 = v + (k * 8)
  ld   x5,0(x6)     // reg x5 (temp) = v[k]
  ld   x7,8(x6)     // reg x7 = v[k + 1]
  sd   x7,0(x6)     // v[k] = reg x7
  sd   x5,8(x6)     // v[k+1] = reg x5 (temp)
  jalr x0,0(x1)     // return to calling routine
```

- Implement assembly code for the following C routine, sort (bubble sort), assuming base address of v[] is *x10*, n is in *x11*, i is in *x19* and j is in *x20*:

```
void sort (long long int v[], size_t n){
      size_t i, j;
      for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
          swap(v,j); }
      }}
```

Solution:

saving registers	sort: addi sp, sp, -40 sd x1, 32 (sp) sd x22, 24 (sp) sd x21, 16 (sp) sd x20, 8 (sp) sd x19, 0 (sp)	# #room on sp for 5 values #save return address #save 4 more register values
move parameters since swap uses x10 and x11	addi x21, x10, 0 addi x22, x11, 0	#x21 <= x10 #x22 <= x11
outer loop	addi x19, x0, 0 forfirst: bge x19, x22, exit1	#load immediate # #goto exit1 if i>=n

Inner loop	addi x20, x19, -1
	forsecond:
	blt x20, x0, exit2
	slli x5, x20,3
	add x5, x21, x5
	ld x6, 0(x5)
	ld x7, 8(x5)
	ble x6, x7, exit2
		# j= i - 1
		#
		# go to exit 2 if j < 0
		#x5 = j *8
		#x5 = v + j*8
		#x6 = v[j]
		#x7 = v[j+1]
		# goto exit2 if x6<x7

pass parameters	addi x10, x21, 0
	addi x11, x20, 0
	jal x1, swap
		#x10 <= x21, v swap param
		#x11 <= x20, j swap param
		# call swap

inner loop	addi x20, x20, -1 jal x0, forsecond	# j -=1 #goto forsecond
	exit2: addi x19,x19,1 jal x0, forfirst	# # i++ # goto forfirst

	exit1:
	ld x19, 0(sp)
	ld x20, 8(sp)
	ld x21, 16(sp)
	ld x22, 24(sp)
	ld x1, 32(sp)
	addi sp, sp, 40
		#restore variables from
		#stack
		
		# restore return address
		# restore sp

	jalr x0, 0(x1)	#return to calling routine

- There are other instructions in RISC-V including set instructions (which write a 1 or 0 to a register), and 32-bit word instructions (which operate on lower 32-bit words ignoring upper word unless sign extension occurs). Totalling 51 instructions in base architecture
- Extensions to RISC-V instructions include multiply (M, 13), atomic (A, 22), single-precision floating point (F, 30), double-precision (D, 32) floating point and compressed instructions (C, which utilize equivalent instructions to base instructions but only 16-bits long, 36).

[not responsible for section 2.14-2.19]