Computer Organization
5-stage RISCV32I Processor
Class Project Phase 2: Power of Abstraction
Fall 2022

Objective: It has been said that technology is doubling every two years. To continue this growth, more efficient methods are required to write software and to develop hardware. **Abstraction** is the idea that a higher level hides details of a lower level. By hiding these details, productivity increases while design errors decrease. C-language is a higher level of **abstraction** to assembly language, and assembly language is a higher level of **abstraction** to machine code.

In this phase, you will write the same routine in c, assembly, and machine code. After you have completed all three implementations, you will compare how long it took you to implement the function in each language and how difficult it was to implement.

Key Learning Outcomes of this phase:
Abstraction: **Abstraction** is one of the eight great ideas in Computer Architecture. It is a technique to make the computer architect and programmer more productive. Another one of the eight great ideas in Computer Architecture is **Design for Moore's Law**, which states computer resources double every 18-24 months. Without the increased productivity from **Abstraction**, design time would lengthen dramatically and make it extremely difficult to utilize these increased resources every 18-24 months.

The remaining six of the eight great ideas in Computer Architecture include:

- **Make the Common Case Fast:** Based on the concept that enhancing performance on the common case will provide a better return than optimizing the rare case.
- **Performance via Parallelism:** Performance can be increased by executing more than one instruction simultaneously. If you can execute two instructions in parallel such as in a dual issue processor, twice as much work can be achieved, 2x the performance
- **Performance via Pipelining:** By dividing an instruction into tasks, where a task uses a specific cpu resource that can be implemented in a sequence, another instruction can use the cpu resource once the instruction before it moves to the next cpu resource. Effectively, parallelizing work by allowing each cpu resource to operate on a different instruction. You will explore this great computer architecture idea in a later phase.
- **Performance via Prediction:** In a pipeline processor, when the program flow changes due to a branch or jump operation, instructions in the pipeline after the branch or jump must be flushed (made into a No Operation (NOP)). A NOP performs no work, and thus reduces program performance. If you can predict whether the branch or jump will occur in an earlier pipeline stage, less NOPs will be inserted and effectively increasing program performance as long as there is a high degree of prediction accuracy. You will explore this great computer architecture idea in a later phase.

- **Hierarchy of Memories:** Processors are most effective by accessing fast memories for load and store operations. Fast memories are relatively small and high cost per memory bit. If a hierarchy of memory is implemented correctly, higher levels of memory which are further away from the processor can be large memories which are typically slow but low cost per memory bit, can appear to be as fast as the small expensive memories. This technique, caching, increases total memory by moving, predicting, data that will be required by the processor to the fast memory and moving it to the slow memory when it is less likely to be needed. You will explore this great computer architecture idea in a later phase.
- **Dependability via Redundancy:** Computers need to be fast and dependable. Processors can be made dependable by including redundancy to take over when a system fails or notify the system when a failure is detected

(The Eight Great Ideas in Computer Architecture are from the textbook “Computer Organization and Design: The Hardware and Software Interface, RISC-V Edition” by David Patterson and John Hennessy)

RISCV assembly: In this phase, you will be introduced to the RISCV assembly language through implementing the same function that you will have implemented in c. Instructions required for this assembly project includes arithmetic immediate as well as register-to-register operations, branches, and store operations.

Simulating an assembly or c-program: In phase 1, you made a simple alteration of an input assembly project. In this phase, you will also be writing a c-program and will learn how to provide all the c-project Compiler Operations settings to enable access to C-libraries and set the size of Heap Memory space.

Due Dates:

- This is a 1-week phase
- Due date: Monday September 19th 2022 at 11:59pm

Instructions:

- You will use your RISC-V project from Phase 1, ia_riscv32i.
    - If you don’t already have it, you can re-import ia_riscv32i through the following git clone command using one of the terminal windows
    - git clone https://github.com/tamisil/ia_riscv32i.git
- Import the assembly_abstraction project using the following git clone command
    - git clone https://github.com/tamisil/assembly_abstraction.git
- Import the c_abstraction project using the following git clone command
    - git clone https://github.com/tamisil/c_abstraction.git
- C-programming language is a higher level of **abstraction** compared to assembly language

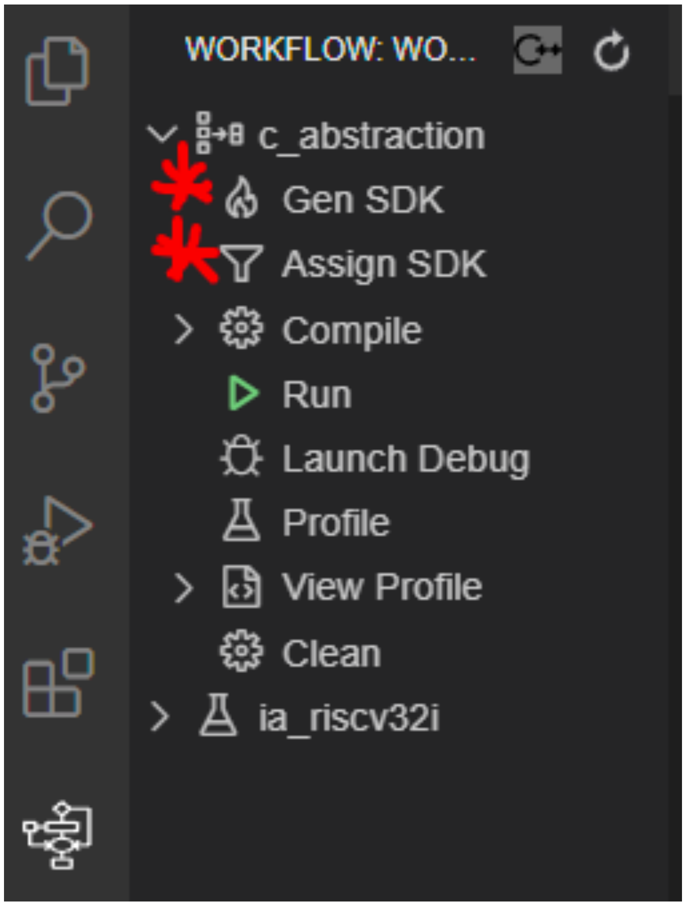
- In reviewing the definition of **abstraction**, it is the concept that a higher level hides details of a lower level. By hiding these details, productivity increases while design errors decrease.
- Let's take a look at an example:
  - c-programming:
    - result = b - a;
  - assembly equivalent:
    - Note: a in memory is located at memory address 0x100, b is located at 0x104, and result is located at 0x108
    - load x3, 0x100(x0) // x0 in RISCV is defined as 0
    - load x4, 0x104(x0)
    - sub x5, x4, x3 // result is in x5 registers
    - sw x5, 0x108(x0)
- What details is the c-line of code hiding?
  - Remembering the physical memory address for a, b, and result
  - Remembering the register locations for a, b, and result
  - Loading a and b from memory
  - Storing result back to memory
- As a program gets larger and larger, hiding these details becomes more significant providing increased productivity. Relying on the c-compiler and linker to manage the variables and the register locations, automation, enables the programmer to focus on the application development and not the programming details to enable the program to execute

Checkpoint 1: Writing the routine in c

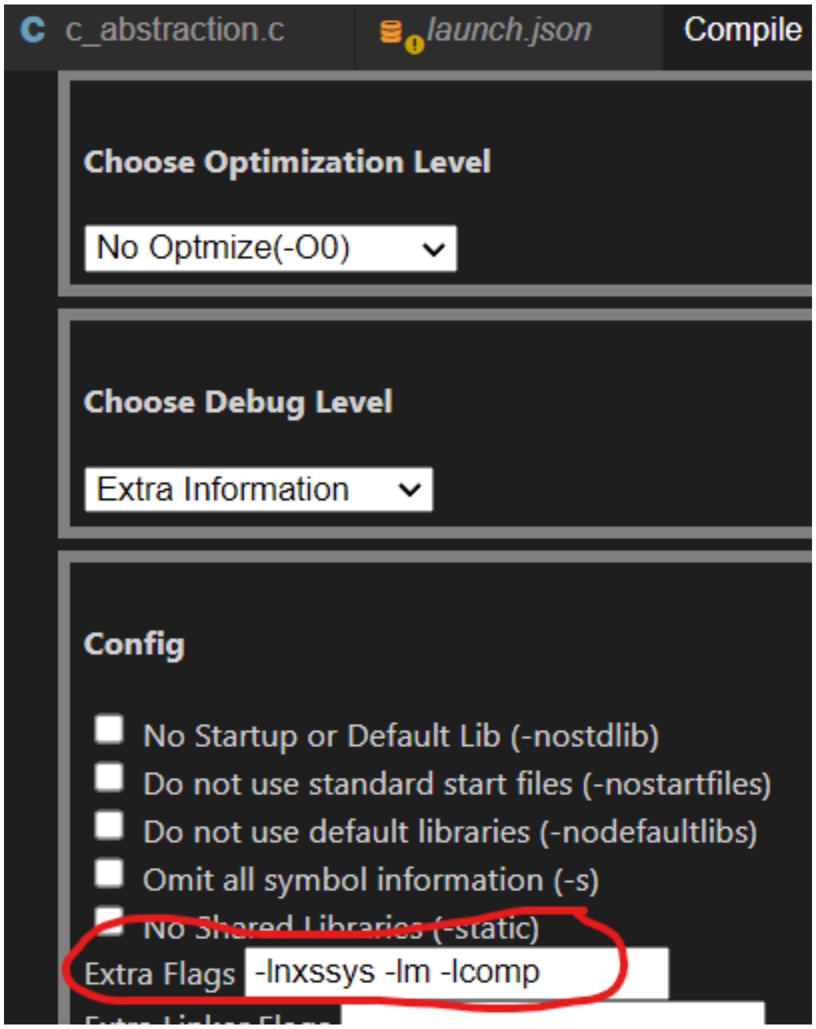
- Using the c_abstraction project imported, write the c-program for the below function for main.
  - For ease of stepping through your c-program, do not change the optimization level of the c_abstration project that you imported. It will be set to optimization level -o0 or no optimization
- Function to program
  - unsigned int letter = 'your first initial in lowercase' // Keith would be 'k'
  - int result = 0 // declare and initialize result
  - Int i = 0 // declare loop variable
  - loop until variable i increments by 1 from 0 to letter
    - within the loop, result = result + i + letter;
    - Note: The c supported in Codasip Studio does not support variable declarations within for loop statements.
    - Example: for (int i = 0; i < max_value; i++) is not valid. This for loop will need to be implemented using two lines of c-code:
      - int i;
      - for (i = 0; i< max_value; i++)
  - Add a printf statement to print final result

        - printf(“Result = %d\n”, result);
        - Note: Do not copy and paste the printf command. From experience, there appears to be hidden special characters that copy over which are not visible and will result in compile errors. Type the command as shown above.
    - return result   // for c-program, it would be the return of main, for assembly, after the loop, you can call the ‘halt’ assembly instruction

- With the c program written, switch to the Workflow perspective , and generate and assign the SDK(ia) as you did in Phase 1.



- Before you build your project, you must inform the linker to include the libraries that will enable the printf function. Click on the cog to the right of the c_abstraction Compile command to set the Compiler Configuration to the following:



c_abstraction.c | launch.json | Compile

Choose Optimization Level

No Optmize(-O0)

Choose Debug Level

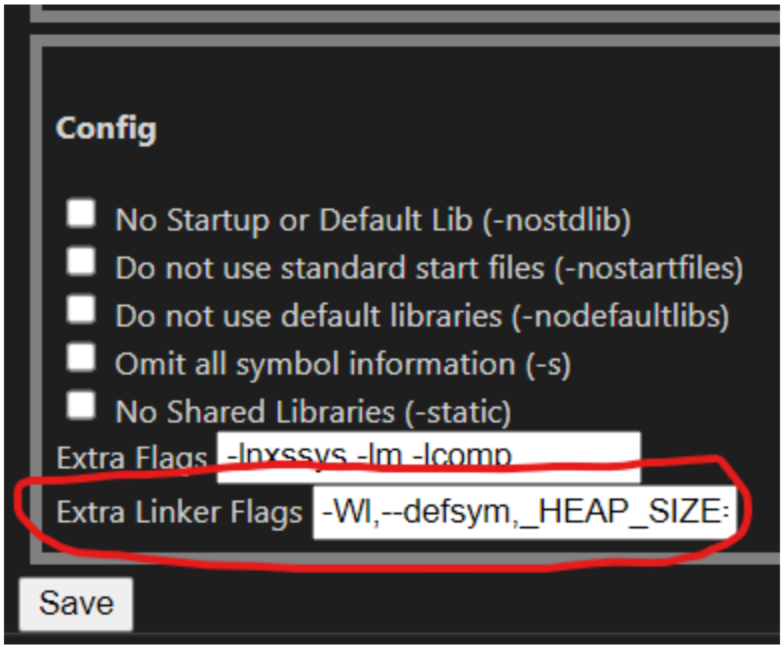
Extra Information

Config

- [ ] No Startup or Default Lib (-nostdlib)
- [ ] Do not use standard start files (-nostartfiles)
- [ ] Do not use default libraries (-nodefaultlibs)
- [ ] Omit all symbol information (-s)
- [ ] No Shared Libraries (-static)

Extra Flags: `-lnxssys -lm -lcomp`

- Heap memory is memory that is persistent memory that will not be reallocated to another function unless specifically unallocated first. Data in Heap memory can be passed from one function to another through a pointer. Stack memory is temporary memory and is only valid as long as the program remains inside the function or its nested function. As soon as the program exits the function that it was allocated, the Stack memory is freed or unallocated so that it can be used by another function.
- The printf function requires persistent or Heap memory to properly execute. You will need to notify the linker during the program compile to inform how much heap memory to allocate
  - Type in the following in the Extra Linker Flags dialog box. The W is a capital W. You must type it in exactly as shown below.
    - -Wl,--defsym,_HEAP_SIZE=0x2000

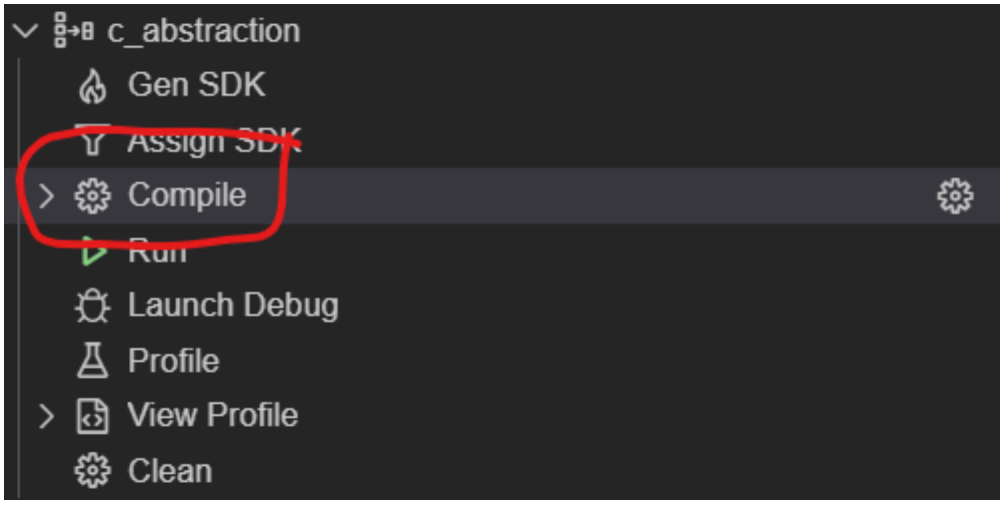


Config

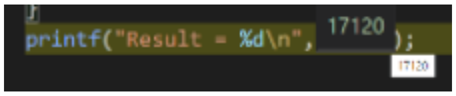
- [ ] No Startup or Default Lib (-nostdlib)
- [ ] Do not use standard start files (-nostartfiles)
- [ ] Do not use default libraries (-nodefaultlibs)
- [ ] Omit all symbol information (-s)
- [ ] No Shared Libraries (-static)

Extra Flags	-lnxssys -lm -lcomp
Extra Linker Flags	-Wl,--defsym,_HEAP_SIZE:

- With the program compiled and the compiler settings updated, you can now compile your project.

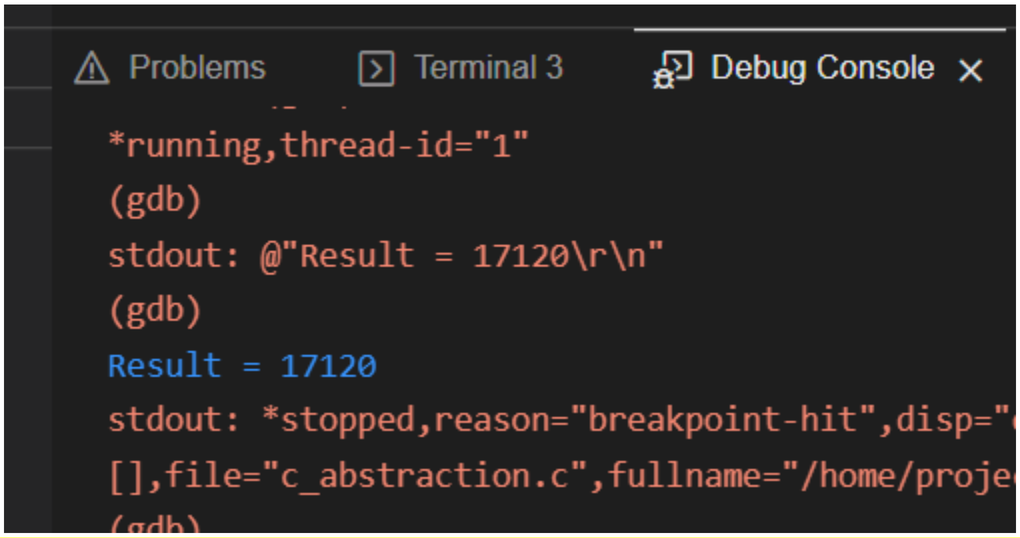


- As you did in Phase 1, place breakpoints in your c-program so that when you run the debugger, you can step through your project and see each line of c-code executed. Use the debugger to debug your program.
  - Place one breakpoint at your printf statement
- To determine the result, click run on your code or step through your code to the printf statement. Place your cursor over the variable result in the printf statement. The value of this variable should appear.



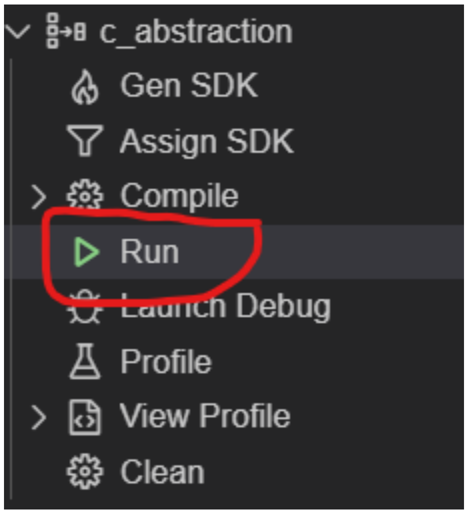
```
printf("Result = %d\n", 17120 );
```

- Once you have a working program, in the debug console, you can scroll up and see the printf statement in blue.

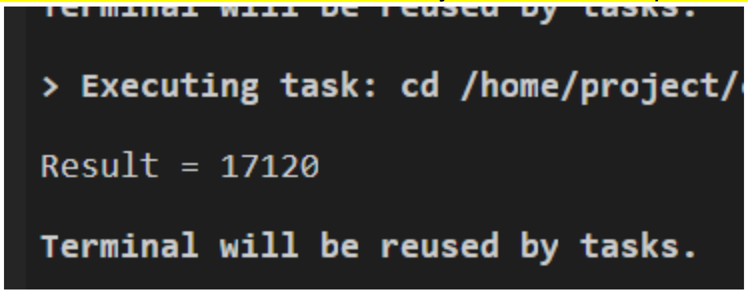


```
*running,thread-id="1"
(gdb)
stdout: @"Result = 17120\r\n"
(gdb)
Result = 17120
stdout: *stopped,reason="breakpoint-hit",disp="
[],file="c_abstraction.c",fullname="/home/proje
(gdb)
```

- Once you have a working program, you can also see your program's printf statement by executing the program by clicking on Run.



- Through the Task:Run window in the bottom, you can see the printf Result



```
Terminal will be reused by tasks.

> Executing task: cd /home/project/

Result = 17120

Terminal will be reused by tasks.
```

- Record the time to implement the c-program and roughly how hard it was to debug (very difficult, difficult, moderate, or easy)

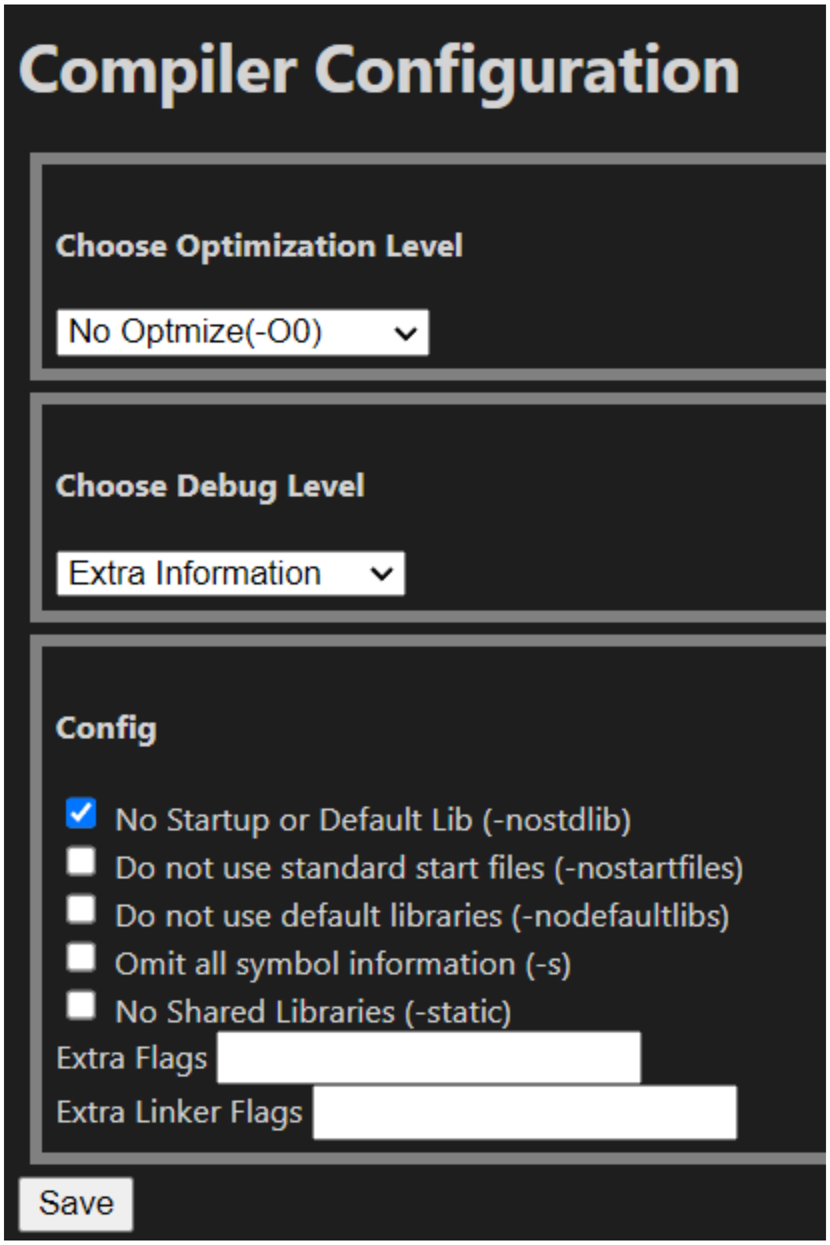
- Assembly programming
    - Assembly programming is extremely detailed in that the programmer is responsible to know which variable is in which register file location and where these variables reside in memory
    - RISCV is a load-store computer architecture
        - The load-store computer architecture separates instructions into memory access operations (loads and stores) and into operations that operate on the data in the register file (register to register or register to immediate). By not combining memory accesses with data manipulation operations, the processor's complexity is reduced which enables **making the common case fast,** one of the eight great ideas in computer architecture.
    - RISCV has six base instruction formats
        - I-type: short immediates and loads
        - R-type: register to register operations
        - S-type: store instructions
        - B-type: conditional branches
        - U-type: long immediates
        - J-type: unconditional jumps
    - Planning is very important in programming and hardware engineering. For this phase, here are some hints and planning suggestions:
        - Register locations are defined in this assembly by the letter x and then the number address such as x15 equates to register 15 in the Register File
        - A common constant used in programming is 0. To assist the compiler and programmer, the RISCV architecture has defined one of the registers in the Register File to be this constant, x0. Register x0 is 0 and is read-only.
            - This constant is another example of **making the common case fast**
        - Assign a unique register file location for each specified variable and non-specified variables
            - Specified variables are i, letter, and result
            - Non-specified variable is the base memory location for the variables
        - How can you assign a 14-bit value, 0x2000, to a register location if the RISC-V immediate is 12-bits?
            - Use an addi immediate function where rs1 = x0 (x0 = 0) with an immediate less than 12-bits
            - Shift logical left the appropriate number of bit locations to move the addi immediate result into the proper location to make the 14-bit value
        - How to calculate the branch address to enable the loop?
            - Allow the assembler and linker to calculate this address by using a label

        - example:
            - LOOP:
                - addi x2, x3, x4
                - …
                - blt x5, x5, LOOP
- Assembly language is a higher level of **abstraction** compared to machine code
    - In reviewing the definition of **abstraction**, it is the concept that a higher level hides details of a lower level. By hiding these details, productivity increases while design errors decrease.
    - Let's take a look at an example:
        - assembly programming
            - sw x15, 0x20(x10)
        - machine code equivalent:
            - 0b00000000111101010010011110100011
            - or, 0x00f527b3
    - What details is the assembly code hiding?
        - Replaces bit patterns with easy to remember mnemonics such as using letters to replace opcodes and register file designators such as x10 to replace 0b01010
        - Hides where each bit is placed in the machine code
            - For example, the destination register location is placed in the instruction word bits 7 through 11
    - As C is a higher abstraction to assembly, assembly is a higher level of abstraction to machine code. As a program gets larger and larger, hiding these details becomes more significant providing increased productivity. Relying on the assembler and linker to manage the opcodes, bit placement, and branch offsets, automation, enables the programmer to focus on the assembly program development and not programming the machine code bits

Checkpoint 2: Writing the routine in RISCV32i assembly language

- Appendix A contains the RISCV32i assembly instructions in your ia_riscv32i project
- For additional information on the RISCV32I instructions and their immediate values, reference the RISC-V refrence card from the textbook or Canvas or the Instruction Set Manual (Unprivileged ISA)
    - https://riscv.org/technical/specifications/
    - Reference chapter 2: RV32I Base Integer Instruction Set
- Using the assembly_abstraction project imported, write the assembly program for the phase function. It is the same function as for **Checkpoint 1** without the printf and a modified return statement.
- Function to program
    - unsigned int letter = 'your first initial in lowercase'   // Keith would be 'k'
    - int result = 0   // declare and initialize result

    - loop until variable i incrementing by 1 from 0 to letter
        - within the loop, result = result + i + letter;
    - return result   // for c-program, it would be the return of main, for assembly, you will be storing the result value in memory location 0x2004
    - Halt
- Function Assembly Hints:
    - As a minimum, you should plan to allocate registers for the following to implement the routine
        - maximum count of loop which will be assigned your first initial lowercase
        - count variable, i, that will go from 0 to maximum count
            - or, the count variable could be assigned max value and count to 0
        - result variable which is initialized to 0
        - memory address to store result initialized to 0x2000
    - You can use the RISCV add immediate instruction, addi, to initialize a register
- Memory locations:
    - letter = 0x2000
    - result = 0x2004
    - i = 0x2008
- As shown in the reference card, the immediate values for an I and S type instructions (as the load and store are) are only 12-bits long. When a memory address requires more than 12 bits, you can first store a value into a register that does not require the 12 bits and then use the natural behavior of the load and store instructions to access the proper memory address. I suggest assigning a register with the value of 0x2000 which can be used by load and stores, as follows:
    - example: x10 = 0x2000
    - lw x11, 0x04(x10)   // loads the value read from memory location 0x2004 (result) into reg x11
- Please review the assembly hints and planning suggestions before this **Checkpoint**.
- For return result, you will implement it with the following instructions:
    - Store value of memory at location 0x2004
        - To get the value of 0x2000 into a register, you can perform the following set of instructions
            - addi x10, x0, 0x2   (load 0x2 into x10)
            - sll x10, x10, 12   (shift 0x2 12-bits to the left, 0x2000)
        - sw x-register that contains the result value, 4(x10)
    - halt instruction
- As you did in Phase 1, Gen SDK if necessary and Assign SDK ot the ia_riscv32i (ia) model
- In the Compiler Configuration for the assembly program, you must select “No Startup or Default Lib”


Compiler Configuration

Choose Optimization Level

No Optmize(-O0)

Choose Debug Level

Extra Information

Config

- [✓] No Startup or Default Lib (-nostdlib)
- [ ] Do not use standard start files (-nostartfiles)
- [ ] Do not use default libraries (-nodefaultlibs)
- [ ] Omit all symbol information (-s)
- [ ] No Shared Libraries (-static)

Extra Flags

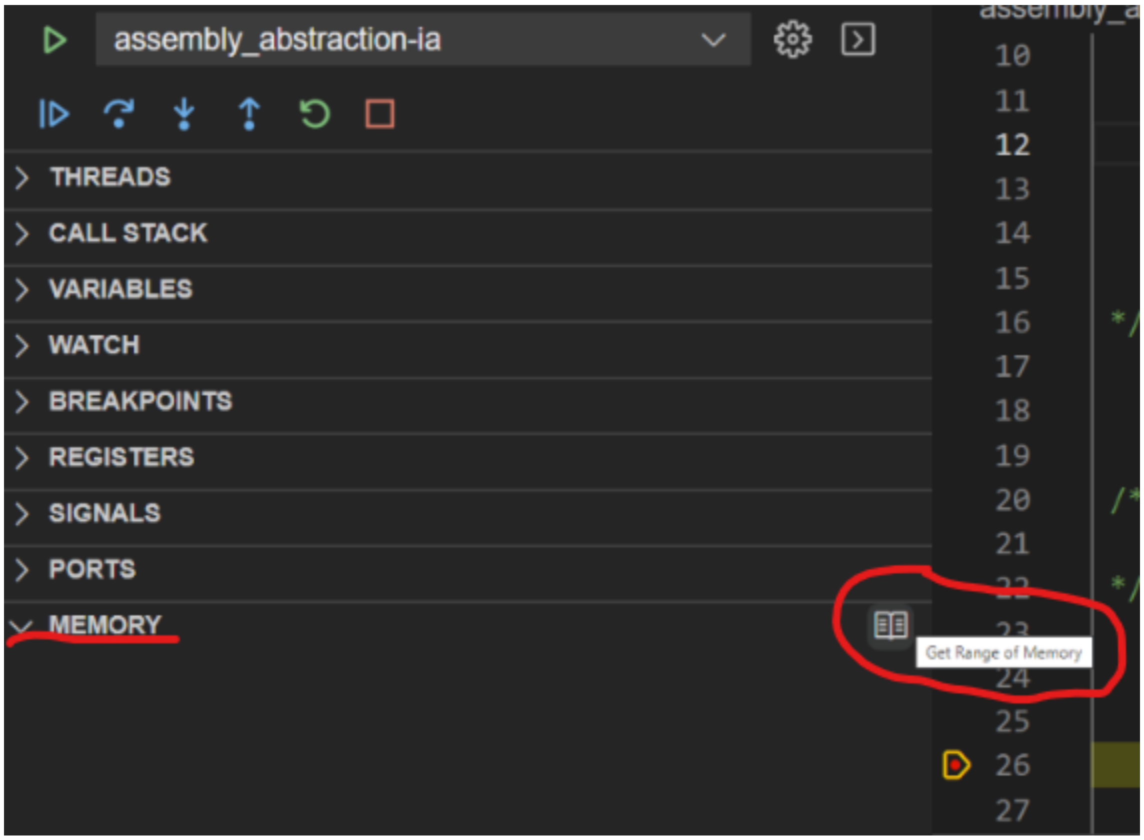
Extra Linker Flags

- After writing the function and setting Compiler Configuration, compile it and step through the routine in the debugger until you have reached the “halt” assembly function
  - You can determine the value of result by looking at the register file location that you have assigned for result
- Record the time to implement the assembly program and roughly how hard it was to debug (very difficult, difficult, moderate, or easy)

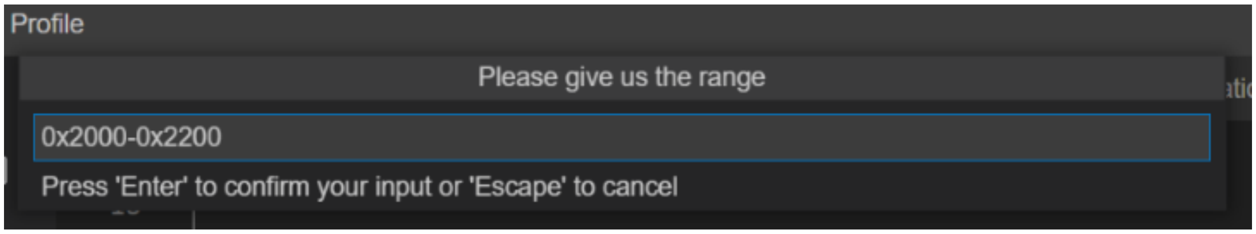
Notes for accessing memory:

For this Phase you are not required to look at the values in memory. However, for those interested, it is a simple process. This topic will also be discussed more in depth in Phase 3.

Place a breakpoint in your assembly routine at the halt statement. Run the debugger and step through or run your program to the halt statement. While in the debugger perspective, expand the Memory and click on “Get Range of Memory”



- In the dialogue box in the upper middle of the screen, type in the desired range of memory to display such as 0x2000-0x2200

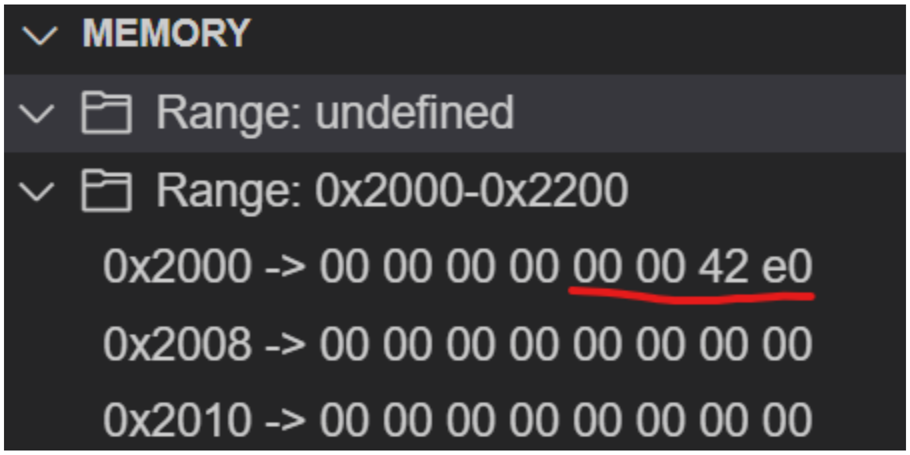


Please give us the range

0x2000-0x2200

Press 'Enter' to confirm your input or 'Escape' to cancel

- Going back to the expanded Memory in the Debugger perspective, expand the range of memory that you entered to see the actual memory locations. In my example,



```
MEMORY
Range: undefined
Range: 0x2000-0x2200
0x2000 -> 00 00 00 00 00 00 42 e0
0x2008 -> 00 00 00 00 00 00 00 00
0x2010 -> 00 00 00 00 00 00 00 00
```

- The memory is displayed as little endian words. At location 0x2000, 0x2000 word is composed of the bytes 3, 2, 1, 0 and word 0x2004 is composed of bytes 7, 6, 5, and 4. This display of hexadecimal numbers puts byte location 4 at the right most position.
  - Hexadecimal 0x000042e0 equates to decimal 17120
- Your value will be different if your initial is not ‘k’

- Machine code
  - Machine code is the lowest level of programming, it is not an **abstraction** to any other programming language
    - It does not hide any details
  - RISCV’s ISA has been developed to **make the common case fast**. The instruction formats are designed to have similar placement in each type of instruction. For example, the opcode is always in bit locations 6..0 and the destination register is in bit locations 11..7.

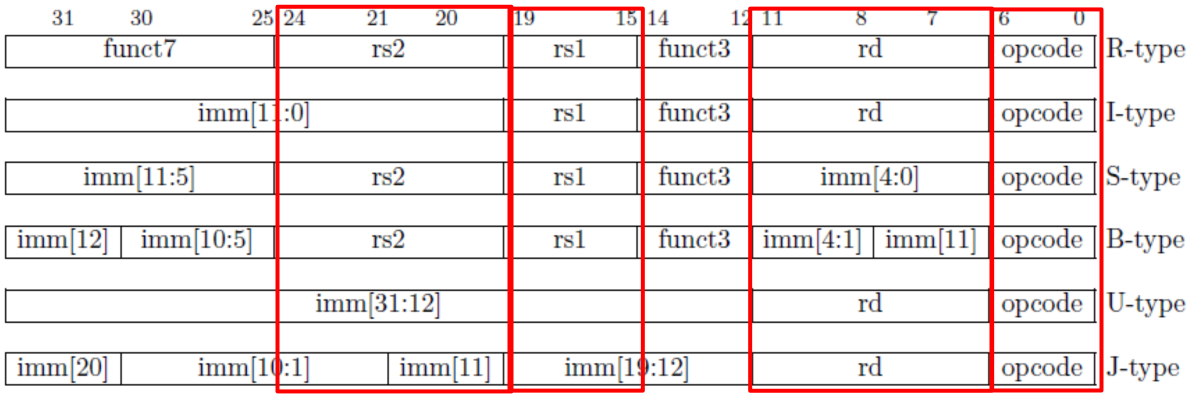

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-type

Figure 2.3: RISC-V base instruction formats showing immediate variants.

  - With the field locations fixed in the instruction formats, accessing the register file or generating the different immediates can be performed in parallel with the instruction decoding.

Checkpoint 3: Writing the routine in RISCV32i machine code

- Appendix B contains information to help write the machine code
- For additional information on the RISCV32I instructions and their immediate values, reference the RISC-V Instruction Set Manual (Unprivileged ISA)
    - https://riscv.org/technical/specifications/
    - Reference chapter 2: RV32I Base Integer Instruction Set
- Using the assembly_abstraction project that you have debugged and successfully ran in **Checkpoint 2**, convert the assembly language into machine code.
    - For each line of assembly code, provide a comment to the right of the assembly operation of the equivalent machine code using hexadecimal format, 0x12345678
    - No machine equivalent for halt instruction is required
- Record how long it took to convert each assembly instruction into machine code. Also, if we could simulate machine code, how difficult would you imagine it would be to debug and verify a machine code version of this routine (very difficult, difficult, moderate, easy)

- Complete the phase
    - Download your c_abstraction.c file within your c_abstraction/src folder
        - Rename it use it the standardname (i.e. tlehman2.c)
    - Download your assembly_abstraction.s file within your assembly_abstraction/src folder
        - Rename it use it the standardname (i.e. tlehman2.s)
    - Submit both files into for Phase 2 on Canvas along with a short text document which should contain the following:
        - Result value while running the assembly program
        - Result value while running your c-program
        - Your comments for each of the programs that you recorded that includes an estimate of the time spent on each **Checkpoint** and how difficult it was to write and debug
            - C-program
            - assembly language
            - machine code
        - Summarize your experience
            - c-program compared to assembly
            - assembly to machine code
            - c-program to machine code

Appendix A: ia_riscv32i RISCV32i supported assembly instructions

i-type: Immediates instructions

addi rd, rs1, immediate	(rd = rs1 + immediate)
slti rd, rs1, immediate	(set rd to 1, if rs1 < immediate)
sltiu rd, rs1, immediate	(set rd to 1, if unsigned compare of rs1 < immediate)
xori rd, rs1, immediate	(rd = rs1 ^ immediate)
ori rd, rs1, immediate	(rd = rs1 | immediate)
andi rd, rs1, immediate	(rd = rs1 & immediate)
lb rd, immediate(rs1)	(rd = sign extended byte from memory at rs1 + immediate)
lh rd, immediate(rs1)	(rd = sign extended halfword from mem at rs1 + imm)
lw rd, immediate(rs1)	(rd = word from memory at rs1 + immediate)
lbu rd, immediate(rs1)	(rd = not signed byte from memory at rs1 + immediate)
lhu rd, immediate(rs1)	(rd = not signed halfword from memory at rs1 + immediate)
jalr rd, immediate(rs1)	(rd = pc + 4, pc = rs1 + immediate)

r-type: Register-to-Register instructions

add rd, rs1, rs2	(rd = rs1 + rs2)
sub rd, rs1, rs2	(rd = rs1 - rs2)
sll rd, rs1, rs2	(rd = rs1 << rs2)
slt rd, rs1, rs2	(set rd to 1, if rs1 < rs2)
sltu rd, rs1, rs2	(set rd to 1, if unsigned compare of rs1 < rs2)
xor rd, rs1, rs2	(rd = rs1 ^ rs2)
srl rd, rs1, rs2	(rd = rs1 >> rs2, logical shift of inserting 0s in upper bits)
sra rd, rs1, rs2	(rd = rs1 >> rs2, arithmetic shift insert sign bit in upper bits)
or rd, rs1, rs2	(rd = rs1 | rs2)
and rd, rs1, rs2	(rd = rs1 & rs2)
slli rd, rs1, immediate	(rd = rs1 << rtype immediate)
srli rd, rs1, immediate	(rd = rs1 >> rtype immediate, logical shift)
srai rd, rs1, immediate	(rd = rs1 >> rtype immediate, arithmetic shift)

s-type: Store instructions

sb rs2, immediate(rs1)	(byte 0 of rs2 stored at memory location rs1 + immediate)
sh rs2, immediate(rs1)	(byte 0 & 1 of rs2 stored at memory location rs1 + imm)
sw rs2, immediate(rs1)	(byte 0,1,2, and 3 stored at memory location rs1 + imm)

b-type: Branch instructions

beq rs1, rs2, immediate	(branch to pc + imm if rs1 = rs2)
bne rs1, rs2, immediate	(branch to pc + imm if rs1 != rs2)
blt rs1, rs2, immediate	(branch to pc + imm if rs1 < rs2)
bge rs1, rs2, immediate	(branch to pc + imm if rs1 >= rs2)
bltu rs1, rs2, immediate	(branch to pc + imm if rs1 < rs2 (unsigned compare))
bgeu rs1, rs2, immediate	(branch to pc + imm if rs1 >= rs2 (unsigned compare))

j-type: Jump instructions

jal rd, immediate	(rd = pc + 4, pc = pc + immediate)

u-type: upper immediate instructions

lui rd, immediate	(rd upper 20 bits = immediate, lower 12 bits = 0)
auipc rd, immediate	(rd = pc + 20-bit upper immediate)

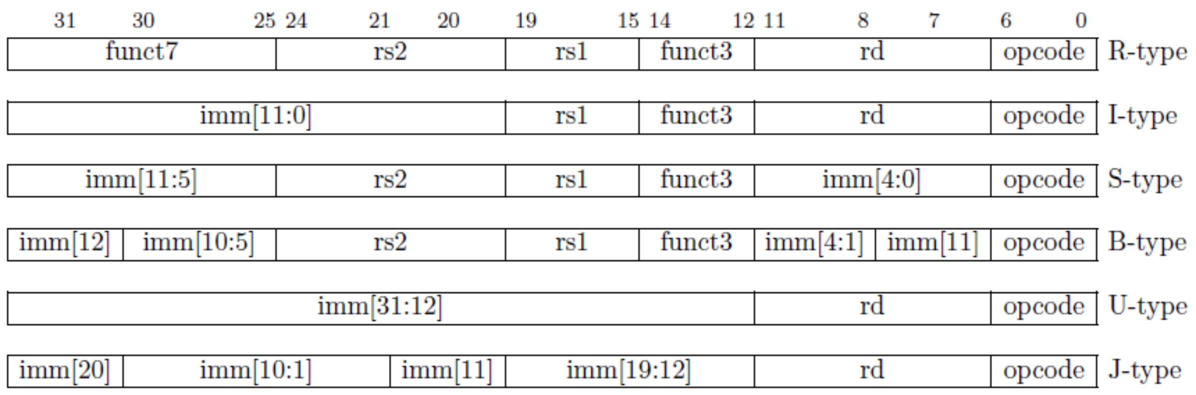
For simulation

halt	(halts simulation and exits debugger, run, or profiler)

Appendix B: RISCV32i instruction formats

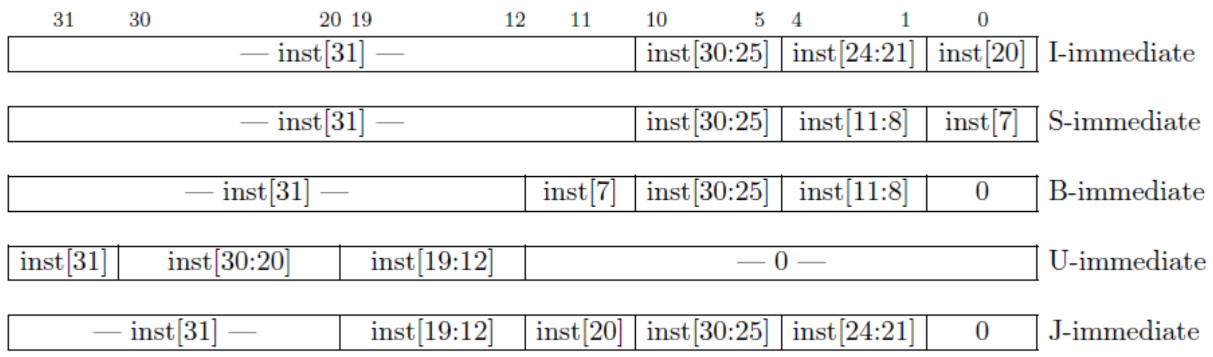
The below screenshots are from the RISC-V Instruction Set Manual (Unprivileged ISA) Vol. 1

- https://riscv.org/technical/specifications/
- Reference chapter 2: RV32I Base Integer Instruction Set



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

Figure 2.3: RISC-V base instruction formats showing immediate variants.



31	30 … 20	19 … 12	11	10 … 5	4 … 1	0	
— inst[31] —				inst[30:25]	inst[24:21]	inst[20]	I-immediate
— inst[31] —				inst[30:25]	inst[11:8]	inst[7]	S-immediate
— inst[31] —			inst[7]	inst[30:25]	inst[11:8]	0	B-immediate
inst[31]	inst[30:20]	inst[19:12]	— 0 —				U-immediate
— inst[31] —		inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0	J-immediate

Figure 2.4: Types of immediate produced by RISC-V instructions. The fields are labeled with the instruction bits used to construct their value. Sign extension always uses inst[31].

Sign-extension is one of the most critical operations on immediates (particularly for XLEN>32), and in RISC-V the sign bit for all immediates is always held in bit 31 of the instruction to allow sign-extension to proceed in parallel with instruction decoding.

Although more complex implementations might have separate adders for branch and jump calculations and so would not benefit from keeping the location of immediate bits constant across types of instruction, we wanted to reduce the hardware cost of the simplest implementations. By rotating bits in the instruction encoding of B and J immediates instead of using dynamic hardware muxes to multiply the immediate by 2, we reduce instruction signal fanout and immediate mux costs by around a factor of 2. The scrambled immediate encoding will add negligible time to static or ahead-of-time compilation. For dynamic generation of instructions, there is some small additional overhead, but the most common short forward branches have straightforward immediate encodings.