

# Computer Organization

## 5-stage RISC-V32I Processor

### Assignment 2: Power of Abstraction

#### Spring 2023

**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 assignment, 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 assignment:

**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 assignment.
- **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 assignment.

- **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 assignment.
- **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 assignment, 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 assignment 1, you made a simple alteration of an input assembly project. In this assignment, 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:

- See canvas

#### Instructions:

- You will use your RISC-V project from assignment 1, ia\_riscv32i.
  - If needed, you can re-import ia\_riscv32i through the following git clone command using one of the terminal windows in Cudasip Studio in the Cloud
  - `git clone https://github.com/CompOrg-RISCV/ia_riscv32i.git`
- Import the **assembly\_abstraction** project using the following git clone command
  - `git clone https://github.com/CompOrg-RISCV/assembly_abstraction.git`
- Import the **c\_abstraction** project using the following git clone command
  - `git clone https://github.com/CompOrg-RISCV/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 RISC-V 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

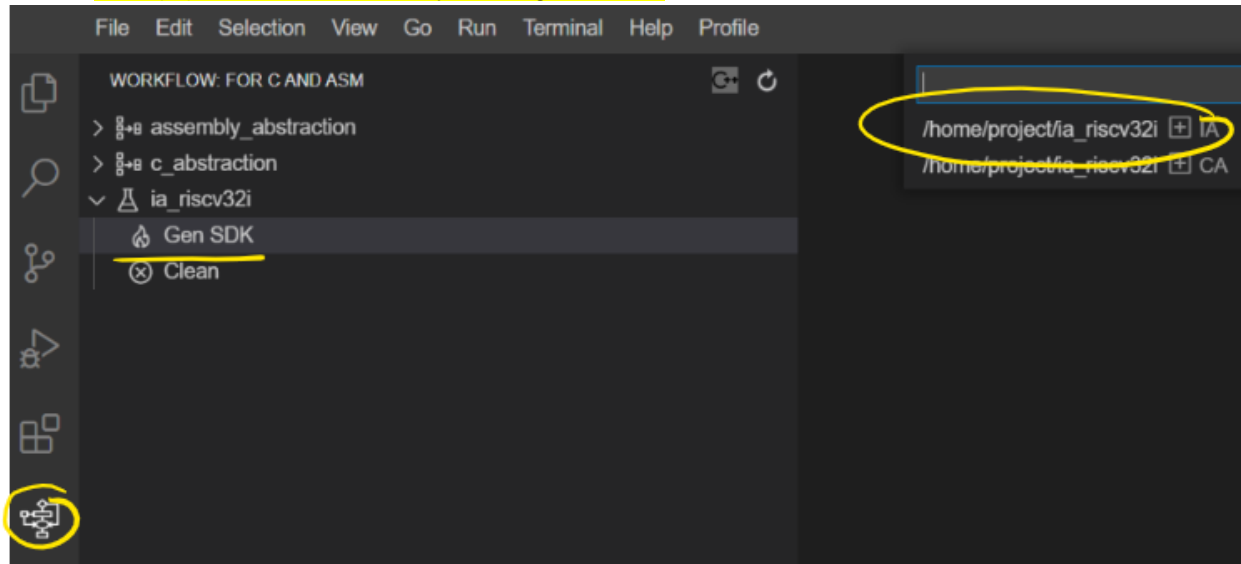
## Checkpoint 1: Writing the routine in c

- 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_abstraction` project that you imported. It will be set to optimization level `-O0` or no optimization
- Function to program
  - `unsigned int letter = 'your last initial in Uppercase'` // Keith Graham = 'G'
  - `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 Cudasip 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);`

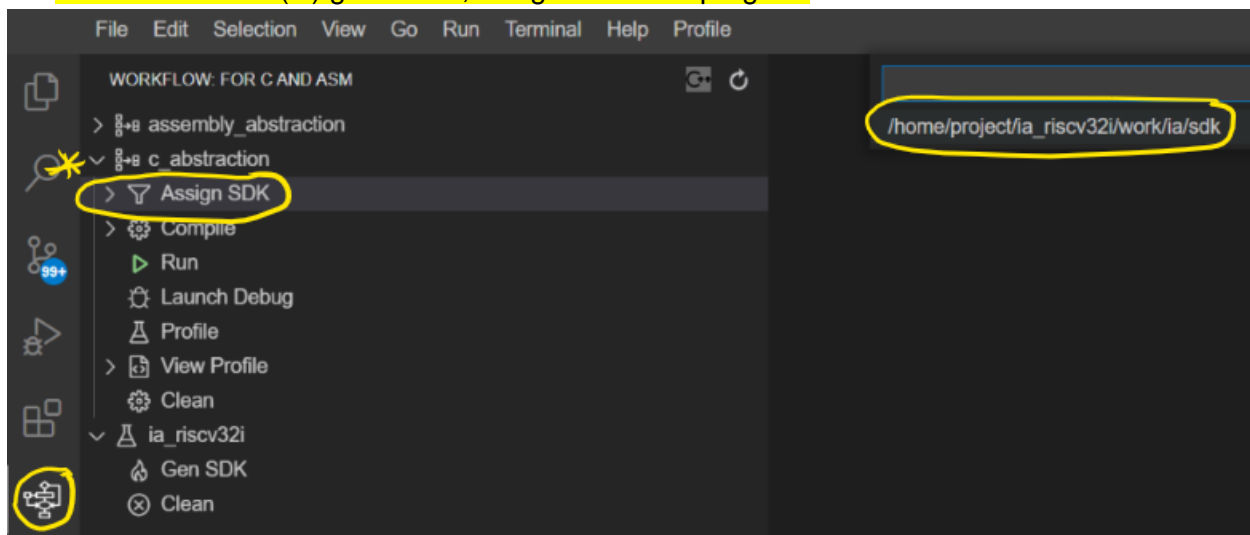
- Note: Do not copy and paste the printf command. From experience, there appears to be hidden special characters that copy over which are not visible and will result in compile errors. Type the command as shown above.


- 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 the SDK(ia) if it has not already been generated

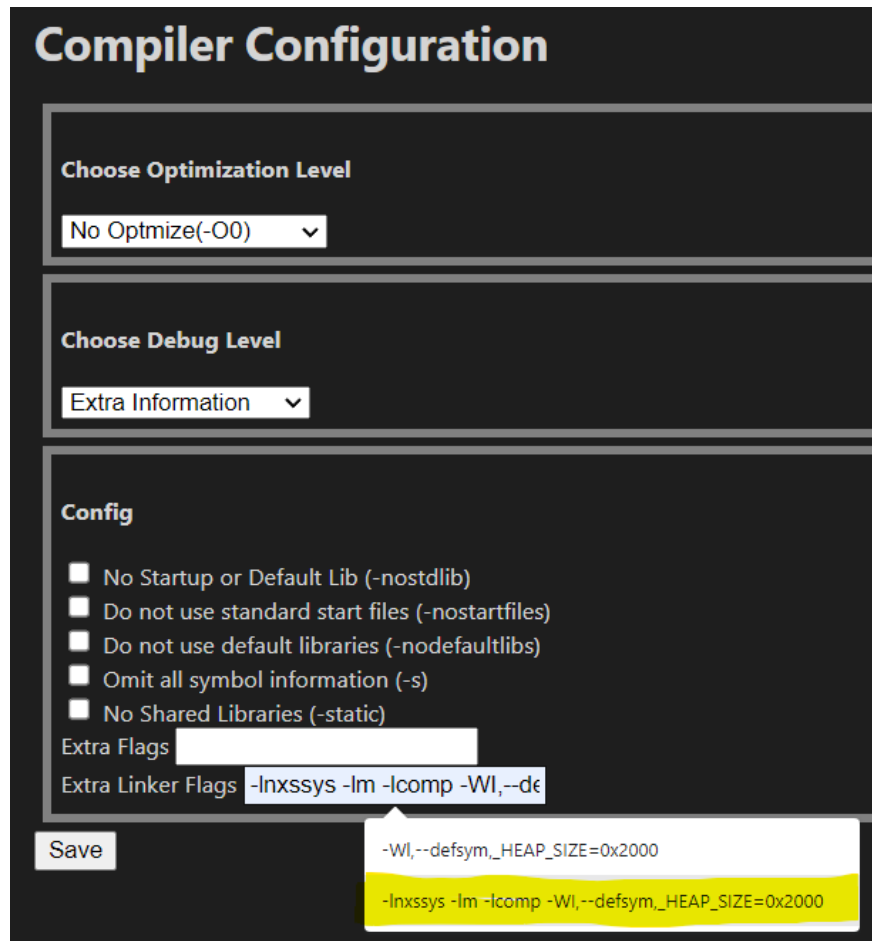


- With the SDK(ia) generated, assign it to the c-program

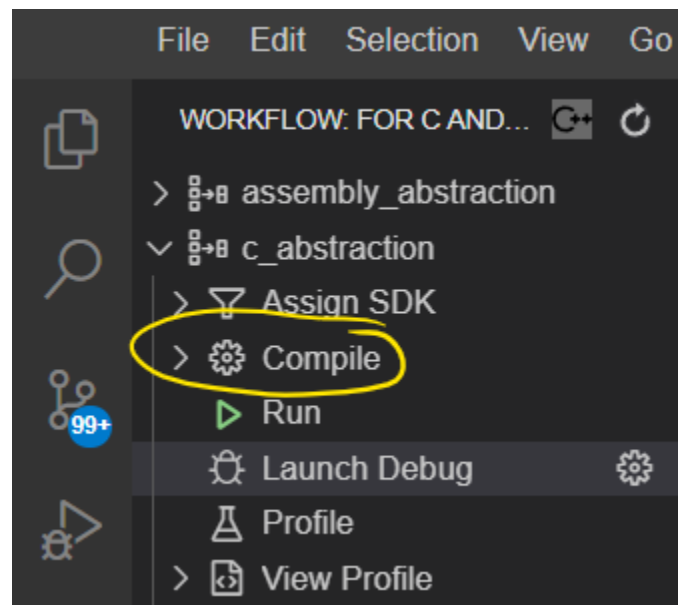


- Before you build your project, you must inform the linker to include the libraries that will enable the printf function. Click on the cog to the right of the c\_abstraction Compile command to set the **Compiler Configuration**, , to the following:

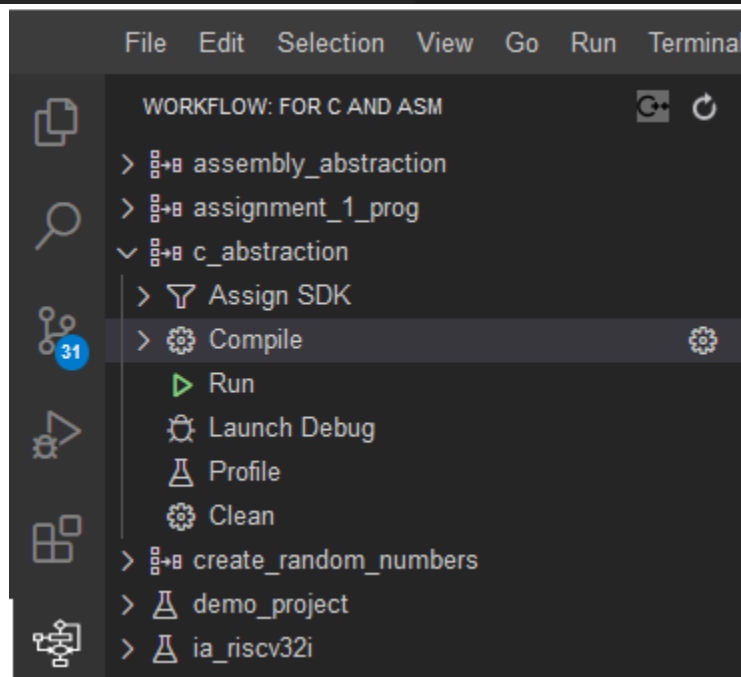
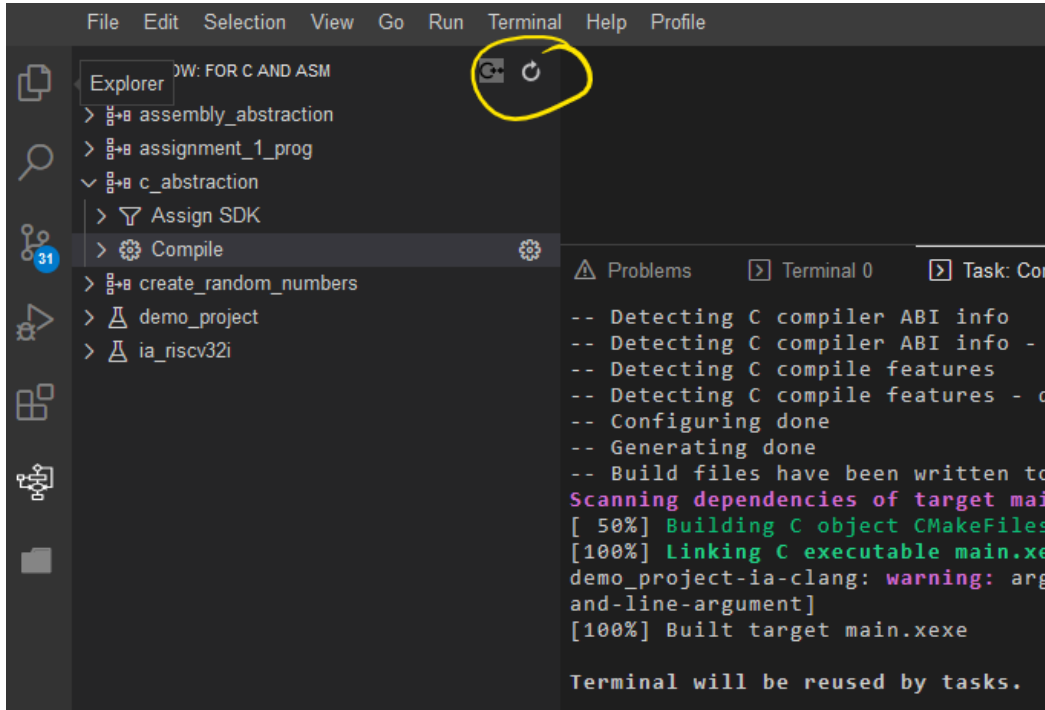
- You will inform the linker where to locate the libraries for “stdio.h” and “string.h” by specifying **-lnxssys -lm -lcomp** linker flags
- The printf function requires persistent or **Heap memory** to properly execute. You will need to notify the linker during the program compile to inform how much heap memory to allocate
  - **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.
- You will reserve enough stack memory to support the printf function by specifying **-WI,--defsym,\_HEAP\_SIZE=0x2000**
- Combine the two linker flag specifiers by setting Extra Linker Flags to **-lnxssys -lm -lcomp -WI,--defsym,\_HEAP\_SIZE=0x2000**
- In later revisions of the c\_abstraction project, the linker information will be imported with your git clone. You should verify that the linker information is located in the Extra Liner Flag dialogue box
  - **-lnxssys -lm -lcomp -WI,--defsym,\_HEAP\_SIZE=0x2000**
- **Note: I recommend copying and pasting the above bold information into the “Extra Linker Flags” field. It field is very specific regarding spaces and commas**



- With the SDK(ia) assigned and the compiler settings updated, you can now compile your project.

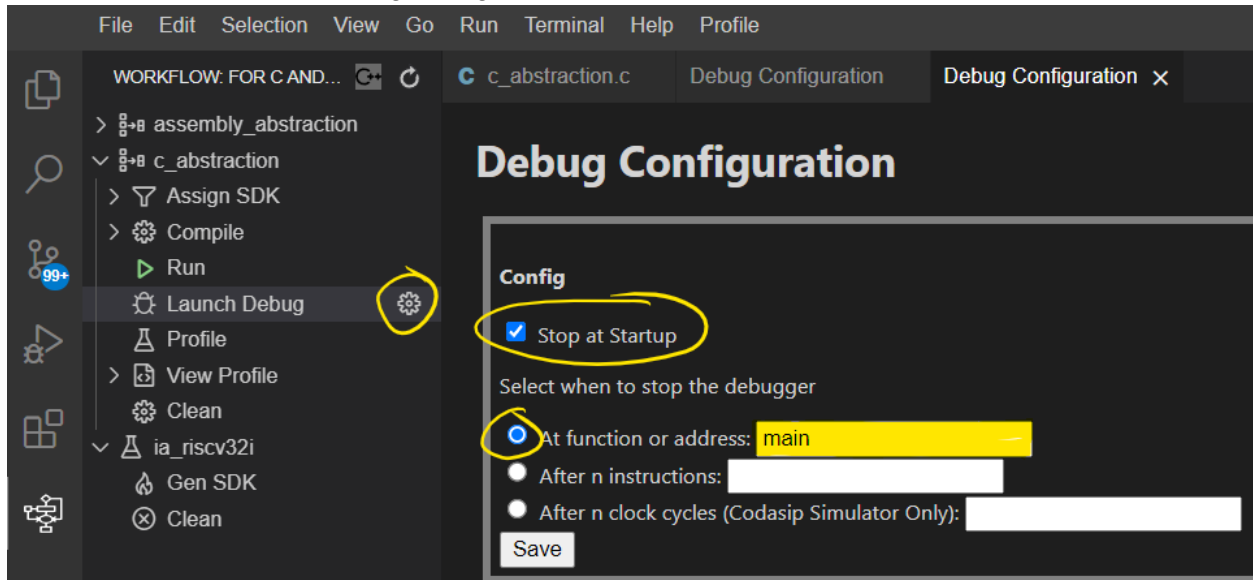


- In the most recent release of Cudasip Studio in the Cloud, you will not see the Run, Launch Debug, Profile, and View Profile commands under the Compile command until a successful compile has occurred. The tool is forcing the correct workflow.
- If the Launch Debug command does not automatically show up after a successful compile, you can manually refresh the screen through click on the refresh icon.

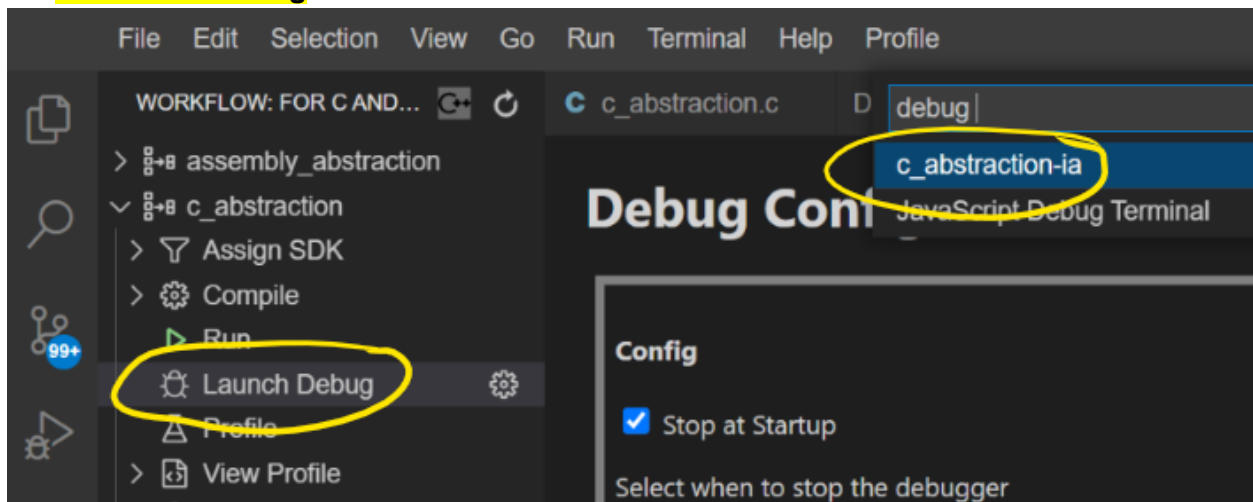



- Before you debug your c-program, it is best practice to specify where the debugger should stop upon startup.
  - Open the Debug Configuration and specify:

- Stop at Startup
- At function or address: `main`
- Save the debug configuration



- Launch Debug

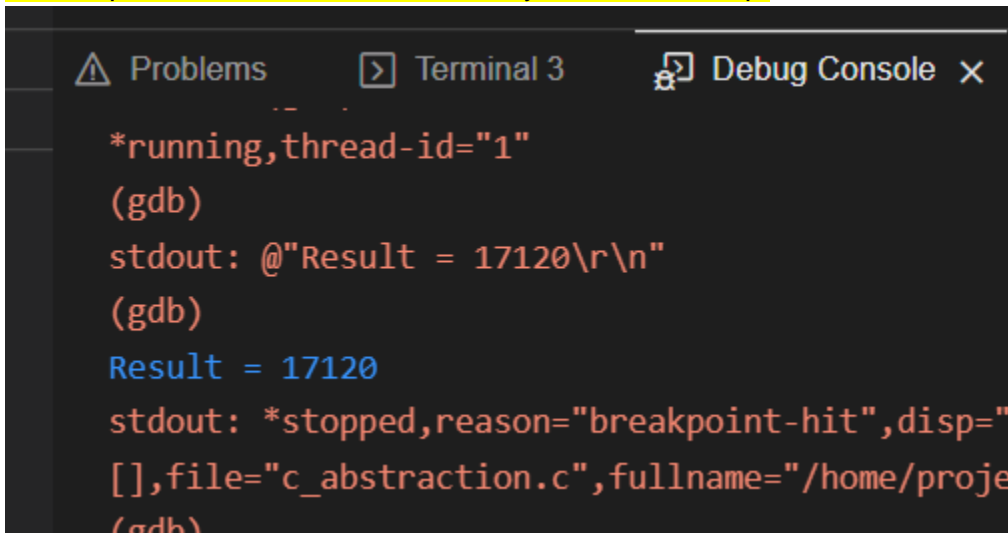


- While in the c-program, you can single step your program as you did in assignment 1's assembly routine. The single step now will be a single line of c-code instead of a single assembly instruction
- To determine the result, step through your code to the printf statement or place a breakpoint at the printf statement and click on resume, . Place your cursor over the variable result in the printf statement. The value of this variable should appear.

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



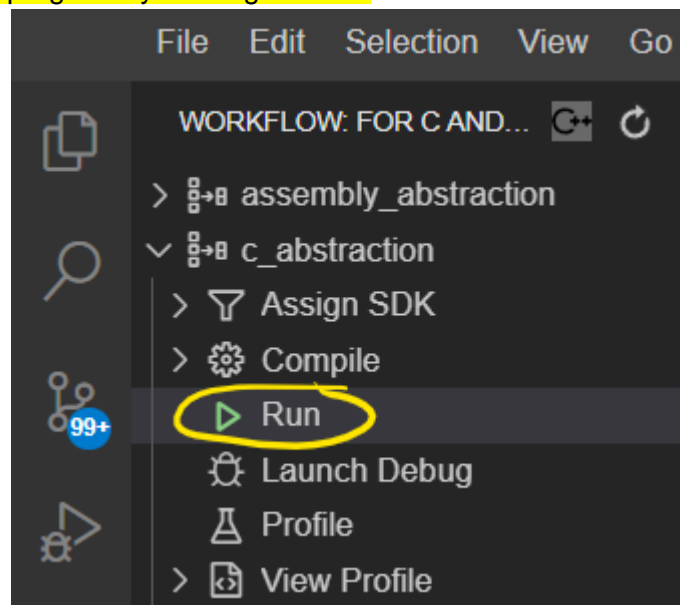
- Once you have a working program, in the debug console, look in the Debug Console to see the printf statement in blue. You may need to scroll up.



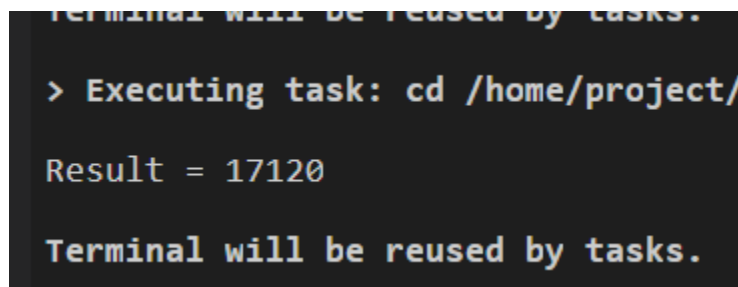
The screenshot shows the 'Debug Console' tab in VS Code. The output is as follows:

```
*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



The screenshot shows the terminal output of the program execution:

```
> Executing task: cd /home/project/
Result = 17120
Terminal will be reused by tasks.
```

- Record your 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
  - RISC-V 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.
  - RISC-V 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 assignment, 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 RISC-V 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 the long upper immediate, `lui`, function that allows you to store the 14-bits as an unsigned 20-bit immediate in the upper 20-bits of the desired register

- Shift logical right immediate, [srl](#), 12-bits to the right to place the 14-bit value in its proper position, the lower 14-bits
- How to calculate the branch address to enable the loop?
    - Allow the assembler and linker to calculate this offset (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:
      - in binary: 0b00000000111101010010011110100011
      - in hex: 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 RISC-V32i assembly language

- Appendix A contains the RISC-V32i assembly instructions in your [ia\\_riscv32i](#) project
- For additional information on the RISC-V32i instructions and their immediate values, reference the RISC-V Instruction Set Manual (Volume 1: 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 assignment 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 last initial in Uppercase'** // e.g. = '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
    - In addition to store the read result, at the end of the assembly routine, save the values of i, result and letter into memory in the memory addresses shown below:
      - letter = 0x2000
      - result = 0x2004
      - i = 0x2008
    - The store operation should be made immediately before the halt instruction
  - 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 RISC-V add immediate instruction, `addi`, to initialize a register
- Memory locations:
  - letter = 0x2000
  - result = 0x2004
  - i = 0x2008
- The load and store immediates are only 12-bit offsets to a register value. With the memory locations greater than 12-bits, you will need to store a value into a register before accessing the memory locations. I suggest assigning a register with the value of 0x2000 which can be used by load and stores.
  - example: `x10 = 0x2000`
  - `lw x11, 0x04(x10)` // loads memory location 0x2004 (result) into reg x11
- Please review the assembly hints and planning suggestions before this **Checkpoint**.
- For storing the three values into memory, you will implement it with the following instructions:
  - Example to store result into memory at location 0x2004
    - To get the value of 0x2000 into a register, you can perform the following instructions

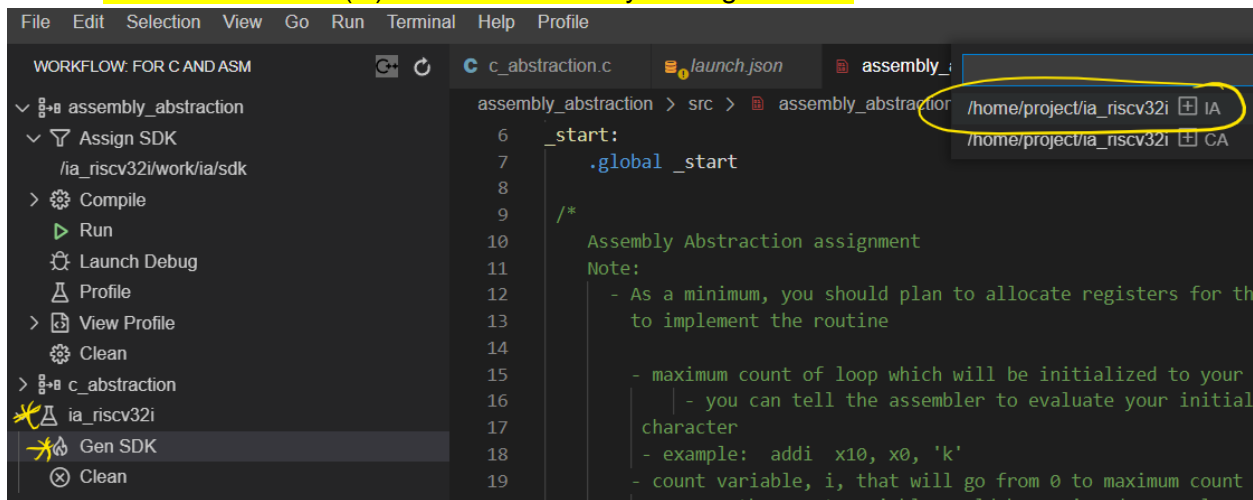
- **lui x10, 0x2**

- load 0x2 into x10's upper 20-bits which will also set the lower 12-bits to 0s
- $x10 = 0x2000$

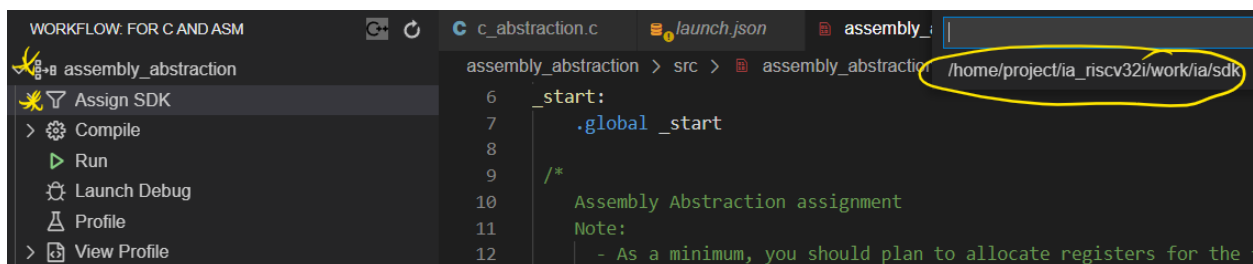
- **sw** x-register that contains the result value, 4(x10)


- What will be the **sw** instructions to store the variables **i** and **letter** into memory without changing the value of x10?
- **halt** instruction

- With the assembly program written, switch to the Workflow perspective, , and Generate the SDK(ia) if it has not already been generated



- With the SDK(ia) generate, assign it to the assembly program



- In the Compiler Configuration, , for the assembly program, you must select "No Startup or Default Lib"

## Compiler Configuration

Choose Optimization Level

No Optimize(-O0) ▼

Choose Debug Level

Extra Information ▼

Config

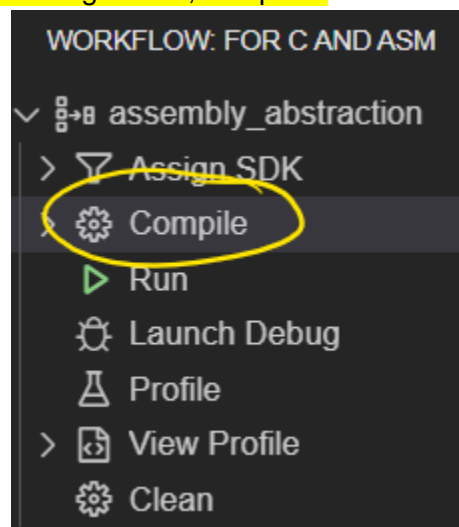
- ☒ No Startup or Default Lib (-nostdlib)
- ☐ Do not use standard start files (-nostartfiles)
- ☐ Do not use default libraries (-nodefaultlibs)
- ☐ Omit all symbol information (-s)
- ☐ No Shared Libraries (-static)

Extra Flags

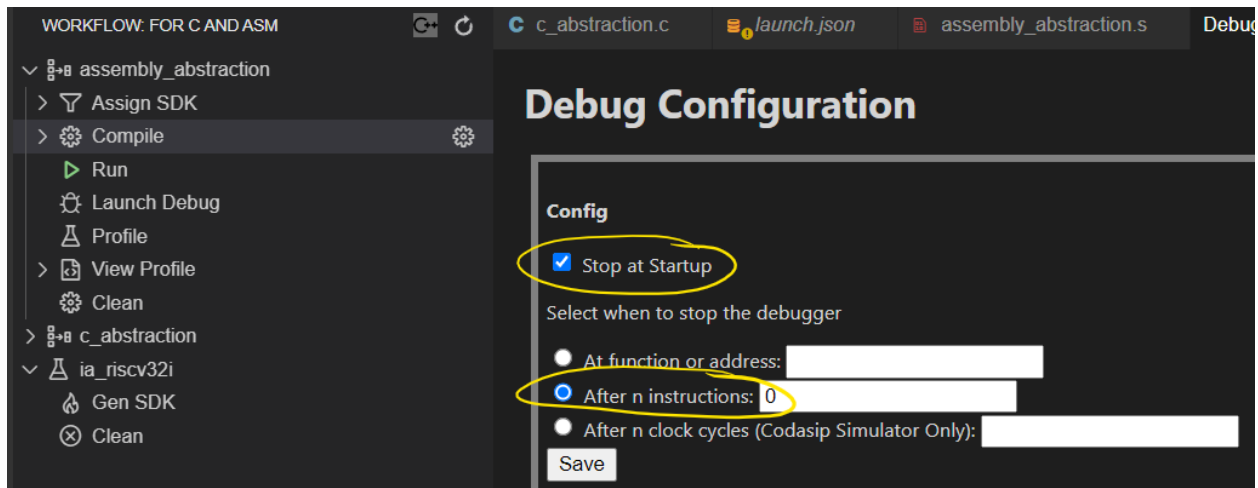
Extra Linker Flags

Save

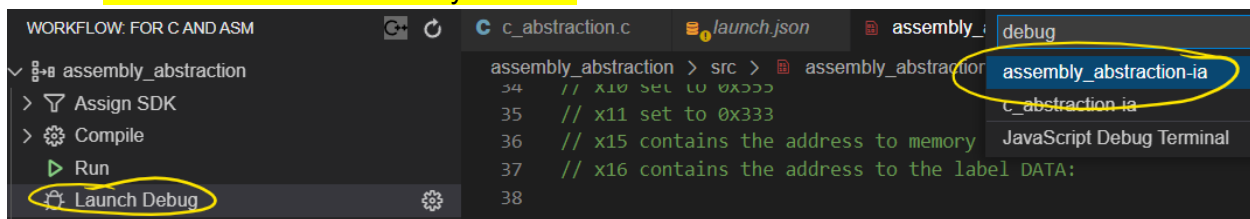
- After setting Compiler Configuration, compile it



- Once the assembly routine compiles without errors, go to the Debugger Configuration and set it to "Stop at Startup" "After n instructions: 0"



- Time to **Launch Debug** 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 your time to implement the assembly program and roughly how hard it was to debug compared to writing in c. (very difficult, difficult, moderate, or easy)

#### Notes for accessing memory:

- For this assignment 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 a future Phase.

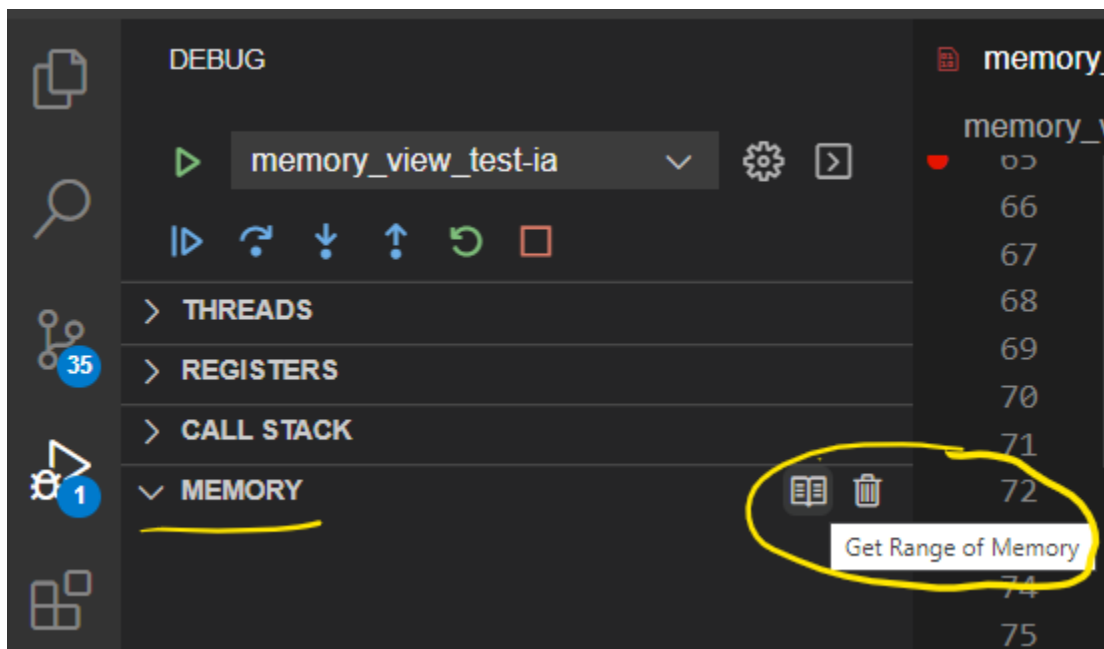
**FAQ: Addressing Memory and Memory View:** [Stores and Loads instructions in the RISCv assembly language access the memory address defined by the rs1 register value plus a 12-bit signed immediate. This video will demonstrate how to load a 32-bit immediate into the Register File that can be used by load and store operations. If you use an ITYPE immediate operation to bring in the value to the register file instead of Load Unsigned Immediate \(lui\), you need to pay attention to whether the 12-bit immediate is a signed value. If you have a smart linker, the linker can take this signed value into account, and provide immediates that will be correctly combined to generate the desired 32-bit immediate.](#)



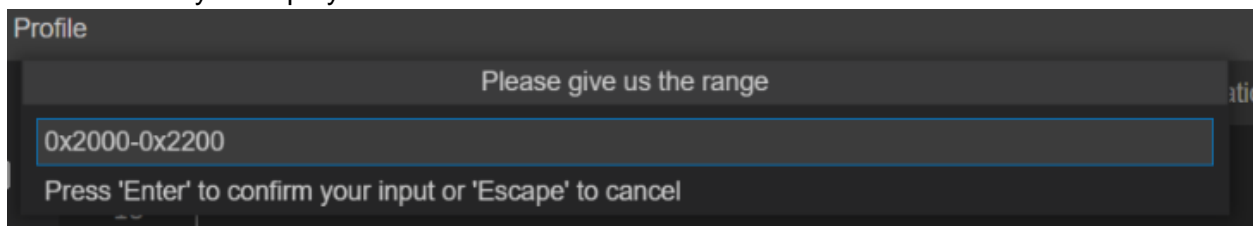
[This video also demonstrates how you can use the linker to load into a register the 32-bit address of a symbol in your assembly program.](#)

To validate that stores occurred properly by your program and processor model, you can view memory locations directly by using the Memory View in Cudasip Studio's debugger perspective. You can select a specific memory location such as 0x1f70 or a range of memory locations, 0x1060-0x1090. The default memory view for the curriculum is Little Endian Word rendering. This format will align the bytes in the same order as the bytes in the Register File.

- 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"

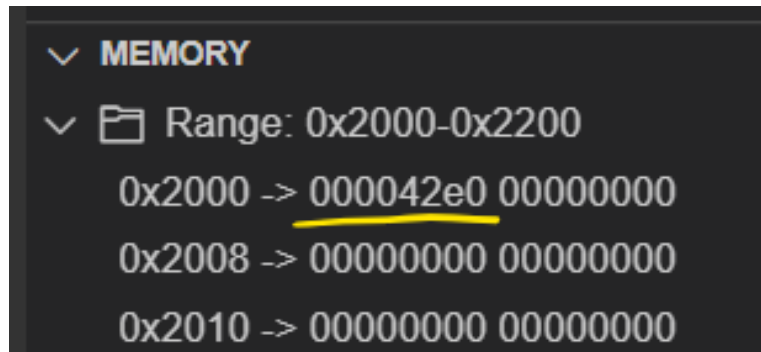


- In the dialogue box in the upper middle of the screen, type in the desired range of memory to display such as 0x2000-0x2200



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





- 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 of a line of displayed data
  - Hexadecimal 0x000042e0 equates to decimal 17120
- Your value will be different if your initial is not 'k'



**FAQ: Extended Register and Memory Views:** [While using Cudasip Studio's debugger to debug C, Assembly, and your Processor projects, it is helpful to see the full processor register set and a range of memory simultaneously. The extended Register and Memory Views expand the register and memory sections in the left hand column while in the debugger perspective by opening tabs for each. These tabs have scrollbars to enable easy navigation to see all the registers as well as the full range of memory.](#)

- 
- 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
    - RISC-V'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.

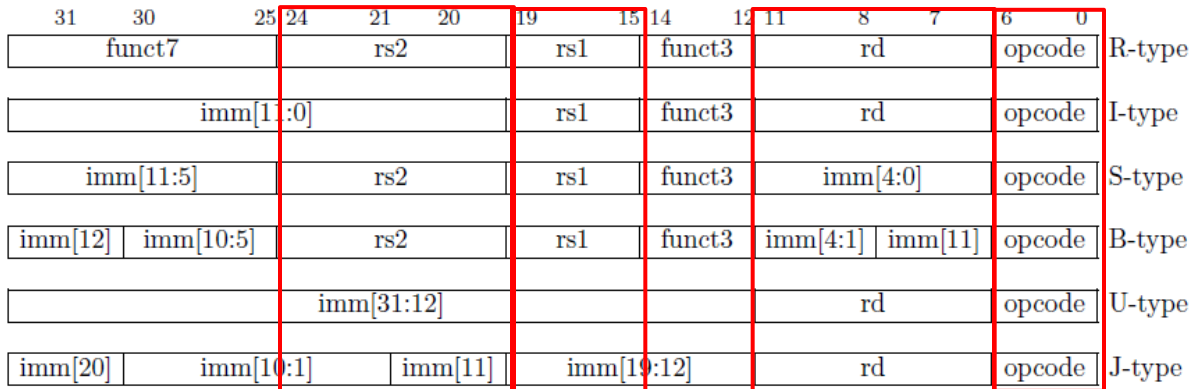


Figure 2.3: RISC-V base instruction formats showing immediate variants.

- 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 RISC-V32i machine code

- Appendix B & Appendix C contains information to help write the machine code
- For additional information on the RISC-V32I 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
- SHORT CUT:** One option is to manually convert each instruction into machine code. Alternatively, you can use `codasip` to do it. Hint: You can look in memory (look at `r_pc` architectural register to see where the instructions are located in memory), or you can look at the `main_r_instruction_buffer` microarchitecture. It's best to convert 1-2 by hand to be sure you're looking at the correct instructions.
- Record how long it took to convert each assembly instruction into machine code, or if you used the short cut, how did you do it and how long did it take. 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 compared to the c and assembly programs. (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. `ekeller2.c`)
  - Download your `assembly_abstraction.s` file within your `assembly_abstraction/src` folder
    - Rename it use it the standardname (i.e. `ekeller2.s`)
  - Submit both files into for Phase 2 on Canvas along with a short text document (`standardname2.txt`, e.g., `ekeller2.txt`) 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: assignment3\_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`)

### Shift-immediate instructions

- `slli rd, rs1, immediate` (`rd = rs1 << shift immediate`)
- `srl rd, rs1, immediate` (`rd = rs1 >> shift immediate, logical shift`)
- `srai rd, rs1, immediate` (`rd = rs1 >> shift 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: RISC-V32i 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.*

## Appendix C: RISC-V-32I Opcodes

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

RV32I Base Instruction Set							
imm[31:12]				rd	0110111	LUI	
imm[31:12]				rd	0010111	AUIPC	
imm[20:10:11:19:12]				rd	1101111	JAL	
imm[11:0]		rs1	000	rd	1100111	JALR	
imm[12:10:5]	rs2	rs1	000	imm[4:1:11]	1100011	BEQ	
imm[12:10:5]	rs2	rs1	001	imm[4:1:11]	1100011	BNE	
imm[12:10:5]	rs2	rs1	100	imm[4:1:11]	1100011	BLT	
imm[12:10:5]	rs2	rs1	101	imm[4:1:11]	1100011	BGE	
imm[12:10:5]	rs2	rs1	110	imm[4:1:11]	1100011	BLTU	
imm[12:10:5]	rs2	rs1	111	imm[4:1:11]	1100011	BGEU	
imm[11:0]		rs1	000	rd	0000011	LB	
imm[11:0]		rs1	001	rd	0000011	LH	
imm[11:0]		rs1	010	rd	0000011	LW	
imm[11:0]		rs1	100	rd	0000011	LBU	
imm[11:0]		rs1	101	rd	0000011	LHU	
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB	
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH	
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW	
imm[11:0]		rs1	000	rd	0010011	ADDI	
imm[11:0]		rs1	010	rd	0010011	SLTI	
imm[11:0]		rs1	011	rd	0010011	SLTIU	
imm[11:0]		rs1	100	rd	0010011	XORI	
imm[11:0]		rs1	110	rd	0010011	ORI	
imm[11:0]		rs1	111	rd	0010011	ANDI	
0000000	shamt	rs1	001	rd	0010011	SLLI	
0000000	shamt	rs1	101	rd	0010011	SRLI	
0100000	shamt	rs1	101	rd	0010011	SRAI	
0000000	rs2	rs1	000	rd	0110011	ADD	
0100000	rs2	rs1	000	rd	0110011	SUB	
0000000	rs2	rs1	001	rd	0110011	SLL	
0000000	rs2	rs1	010	rd	0110011	SLT	
0000000	rs2	rs1	011	rd	0110011	SLTU	
0000000	rs2	rs1	100	rd	0110011	XOR	
0000000	rs2	rs1	101	rd	0110011	SRL	
0100000	rs2	rs1	101	rd	0110011	SRA	
0000000	rs2	rs1	110	rd	0110011	OR	
0000000	rs2	rs1	111	rd	0110011	AND	
fm	pred	succ	rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL	
000000000001		00000	000	00000	1110011	EBREAK	

## Appendix D: YouTube videos for Assignment 2

### Assignment Videos:

- **Assignment 2: Power of Abstraction**

- Abstraction is one of the “Eight Great Ideas of Computer Architecture” discussed by David Patterson and John Hennessy in their textbook, “Computer Organization and Design: The Hardware/Software Interface.” Without Abstraction, we would not be able to take advantage of Moore’s Law which roughly states that silicon or computer resources are doubling every 18-24 months. To take advantage of these additional resources, we must become more productive. Abstraction is one of the productivity tools or techniques that we use. This video summarizes the three activities in this assignment.
- <https://www.youtube.com/watch?v=bnlkM5UZuqc&list=PLTUn6Ox9e6q2ienoql3KClFtRPMqO28uj&index=4>

- **Assignment 2: Checkpoint 1: Writing the routine in c**

- In Assignment 2, Checkpoint 1, you will be writing a simple program in C which is a highly abstracted programming language compared to assembly. This video will demonstrate how to set the Compiler Configuration settings to link in the libraries for stdio.h and string.h as well as specify the linker how much heap memory to allocate for this program. With the c-routine using a printf statement, once the c-routine is functioning properly, you can obtain the result via the printf statement by executing the program using the debugger or run command.
- <https://www.youtube.com/watch?v=lpcedaDGsbOs&list=PLTUn6Ox9e6q2ienoql3KClFtRPMqO28uj&index=4>

- **Assignment 2: Checkpoint 2: Writing the routine in RISCv32i assembly language**

- In Assignment 2, Checkpoint 2, you will be implementing the same routine as Checkpoint 1, but it will be implemented in RISCv assembly instead of c. The program flow of an assembly routine is very similar to a c-program. This video will recommend a process flow to implement your assembly program, how to initialize variables (registers), use a for loop label, manage your for loop counter, and store the result(s) into memory.
- <https://www.youtube.com/watch?v=9GdZTC6AW1g&list=PLTUn6Ox9e6q2ienoql3KClFtRPMqO28uj&index=5>

- **Assignment 2: Checkpoint 3: Writing the routine in RISCv32i machine code**

- In Assignment 2, Checkpoint 3, you will use the assembly routine that you wrote in Assignment 2, Checkpoint 2, Writing the routine in RISCv32i assembly, to complete this portion of the assignment.. In this checkpoint, for each assembly instruction, you will write its equivalent machine code in a comment to the write of the assembly instruction.

- This video will show you through an example of how to convert an assembly instruction into its equivalent machine code. Machine code is the binary representation of an assembly instruction that the processor will use to decode and perform the operation. At the end of this checkpoint, you will record the level of difficulty in writing the machine code for each assembly instruction and how long it took to complete the checkpoint.
- <https://www.youtube.com/watch?v=J6ZbyCEat50&list=PLTUn6Ox9e6q2ienoql3KCIEtRPMqO28uj&index=6>

## Frequently Asked Questions (FAQs) Videos

- It is intended for students to provide them **real-time support** who have been assigned a **project-based learning assignment, based on the Cudasip Curriculum.**
- **FAQ: Helpful c program debug hints**
  - While debugging a c-program, it is important (or at least easier) to configure the compiler settings with no-optimization or optimization disabled. This video will show you why the c-compiler optimization should be disabled and two ways that you can view a variable's value.
  - [https://www.youtube.com/watch?v=T5peFJndzeo&list=PLTUn6Ox9e6q1ii0fp-N\\_GDPZjAtkDZmqe&index=12&t=8s](https://www.youtube.com/watch?v=T5peFJndzeo&list=PLTUn6Ox9e6q1ii0fp-N_GDPZjAtkDZmqe&index=12&t=8s)
- **FAQ: Addressing Memory and Memory View**
  - Stores and Loads instructions in the RISC-V assembly language access the memory address defined by the rs1 register value plus a 12-bit signed immediate. This video will demonstrate how to load a 32-bit immediate into the Register File that can be used by load and store operations. If you use an ITYPE immediate operation to bring in the value to the register file instead of Load Unsigned Immediate (lui), you need to pay attention to whether the 12-bit immediate is a signed value. If you have a smart linker, the linker can take this signed value into account, and provide immediates that will be correctly combined to generate the desired 32-bit immediate. This video also demonstrates how you can use the linker to load into a register the 32-bit address of a symbol in your assembly program.

To validate that stores occurred properly by your program and processor model, you can view memory locations directly by using the Memory View in Cudasip Studio's debugger perspective. You can select a specific memory location such as 0x1f70 or a range of memory locations, 0x1060-0x1090. The default memory view for the curriculum is Little Endian Word rendering. This format will align the bytes in the same order as the bytes in the Register File.

- [https://www.youtube.com/watch?v=8BkUEZXfGSA&list=PLTUn6Ox9e6q1ii0fp-N\\_GDPZjAtkDZmqe&index=13](https://www.youtube.com/watch?v=8BkUEZXfGSA&list=PLTUn6Ox9e6q1ii0fp-N_GDPZjAtkDZmqe&index=13)
- **FAQ: Extended Register and Memory Views**



- While using Cudasip Studio's debugger to debug C, Assembly, and your Processor projects, it is helpful to see the full processor register set and a range of memory simultaneously. The extended Register and Memory Views expand the register and memory sections in the left hand column while in the debugger perspective by opening tabs for each. These tabs have scrollbars to enable easy navigation to see all the registers as well as the full range of memory.
- [https://www.youtube.com/watch?v=Zl-zSGdvSkE&list=PLTUn6Ox9e6q1ii0fp-N\\_GDPZjAtkDZmqe&index=18](https://www.youtube.com/watch?v=Zl-zSGdvSkE&list=PLTUn6Ox9e6q1ii0fp-N_GDPZjAtkDZmqe&index=18)