



(230525d) Considering to write RV32E interpreter based on switch by instruction format type to fetch instruction parameters as Rd Rs1 Rs2 Imm and then in each path switch on instruction ccccc code to implement each one and then perform PC increment and go over in loop. Can you provide some template in ANSI C?



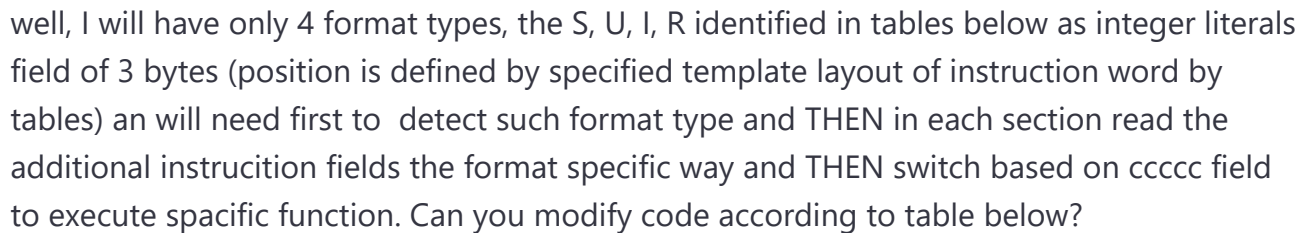
I understand that you want to write an RV32E interpreter using ANSI C, and you plan to use a switch statement to fetch instruction parameters and implement each instruction. I'll provide a basic template to help you get started.

c

Copy code

```
#include <stdint.h> #include <stdio.h> // Define RV32E instruction format types
typedef enum { R_TYPE, I_TYPE, S_TYPE, B_TYPE, U_TYPE, J_TYPE, UNKNOWN_TYPE }
InstructionFormatType; // Function prototypes InstructionFormatType
decode_instruction_format(uint32_t instruction); void
execute_instruction(uint32_t instruction, InstructionFormatType format); int
main() { uint32_t program_counter = 0; uint32_t instruction_memory[256] = {0};
// Replace with actual instructions uint32_t num_instructions =
sizeof(instruction_memory) / sizeof(instruction_memory[0]); while
(program_counter < num_instructions) { uint32_t instruction =
instruction_memory[program_counter]; InstructionFormatType format =
decode_instruction_format(instruction); execute_instruction(instruction,
format); program_counter++; // Increment the PC } return 0; }
InstructionFormatType decode_instruction_format(uint32_t instruction) { //
Decode the instruction format based on opcode or other fields // Return the
appropriate format type } void execute_instruction(uint32_t instruction,
InstructionFormatType format) { uint32_t rd, rs1, rs2, imm; switch (format) {
case R_TYPE: // Extract rd, rs1, rs2, and imm (if needed) // Perform the R-type
instruction based on the function code break; case I_TYPE: // Extract rd, rs1,
and imm // Perform the I-type instruction based on the opcode break; case
S_TYPE: // Extract rs1, rs2, and imm // Perform the S-type instruction based on
the opcode break; case B_TYPE: // Extract rs1, rs2, and imm // Perform the B-
type instruction based on the opcode break; case U_TYPE: // Extract rd and imm
// Perform the U-type instruction based on the opcode break; case J_TYPE: //
Extract rd and imm // Perform the J-type instruction based on the opcode break;
case UNKNOWN_TYPE: default: printf("Unknown instruction format\n"); break; } }
```

This template provides a basic structure for an RV32E interpreter. You'll need to implement the `decode_instruction_format()` function to identify the instruction format type and fill out



RV32E/VMEX - YES we will implement RV32E 32bit instructions

-----+

-----+-----+-----+

```
-sx|--immediate|rs1|-rd|010cccc
-----+-----+-----+-----+
```

R-TYPE (32)

= ADD,SUB,AND,OR,XOR,SLT,SLTU,SRA,SRL,SLL,MUL

```
31                0
-----+-----+-----+-----+
000|rs2|000|rs1|000|-rd|011cccc
-----+-----+-----+-----+
```



I understand your requirements better now. Based on the provided table, I've modified the template to detect the format type using the 3-byte integer literal fields and then extract and execute the instructions accordingly.

c

Copy code

```
#include <stdint.h> #include <stdio.h> // Define RV32E instruction format types
typedef enum { S_TYPE, U_TYPE, I_TYPE, R_TYPE, UNKNOWN_TYPE }
InstructionFormatType; // Function prototypes InstructionFormatType
decode_instruction_format(uint32_t instruction); void
execute_instruction(uint32_t instruction, InstructionFormatType format); int
main() { uint32_t program_counter = 0; uint32_t instruction_memory[256] = {0};
// Replace with actual instructions uint32_t num_instructions =
sizeof(instruction_memory) / sizeof(instruction_memory[0]); while
(program_counter < num_instructions) { uint32_t instruction =
instruction_memory[program_counter]; InstructionFormatType format =
decode_instruction_format(instruction); execute_instruction(instruction,
format); program_counter++; // Increment the PC } return 0; }
InstructionFormatType decode_instruction_format(uint32_t instruction) { uint32_t
format_code = instruction & 0x7; // Extract the 3 least significant bits switch
(format_code) { case 0: return S_TYPE; case 1: return U_TYPE; case 2: return
I_TYPE; case 3: return R_TYPE; default: return UNKNOWN_TYPE; } } void
execute_instruction(uint32_t instruction, InstructionFormatType format) {
uint32_t rd, rs1, rs2, imm, opcode; switch (format) { case S_TYPE: rs1 =
(instruction >> 15) & 0x1F; rs2 = (instruction >> 20) & 0x1F; imm =
((instruction >> 25) << 5) | ((instruction >> 7) & 0x1F); opcode = (instruction
>> 2) & 0x7; // Perform the S-type instruction based on the opcode break; case
U_TYPE: rd = (instruction >> 7) & 0x1F; imm = (instruction >> 12) & 0xFFFF;
opcode = (instruction >> 2) & 0x7; // Perform the U-type instruction based on
the opcode break; case I_TYPE: rd = (instruction >> 7) & 0x1F; rs1 =
```

```
(instruction >> 15) & 0x1F; imm = (instruction >> 20) & 0xFFF; opcode =
(instruction >> 2) & 0x7; // Perform the I-type instruction based on the opcode
break; case R_TYPE: rd = (instruction >> 7) & 0x1F; rs1 = (instruction >> 15) &
0x1F; rs2 = (instruction >> 20) & 0x1F; opcode = (instruction >> 2) & 0x7; //
Perform the R-type instruction based on the opcode break; case UNKNOWN_TYPE:
default: printf("Unknown instruction format\n"); break; } }
```

This modified template extracts the format type based on the 3-byte integer literal fields and then, in each case, extracts the additional instruction fields specific to that format. You'll need to implement the actual execution of instructions based on the opcode inside each switch case.



format type is identified by literal 3bits field at instruction[7-5], opcode is defined as ccccc at instruction[4-0], other parameters fields placement is format specific, can you please provide code also with template switch by ccccc in each path too?



I apologize for the confusion earlier. Based on your requirements, I've updated the template to identify the format type using the 3-bit field at instruction[7-5] and added switch cases for the opcode (cccc) at instruction[4-0]. I also made sure the instruction fields are extracted according to the specified format.

c

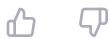
Copy code

```
#include <stdint.h> #include <stdio.h> // Define RV32E instruction format types
typedef enum { S_TYPE, U_TYPE, I_TYPE, R_TYPE, UNKNOWN_TYPE }
InstructionFormatType; // Function prototypes InstructionFormatType
decode_instruction_format(uint32_t instruction); void
execute_instruction(uint32_t instruction, InstructionFormatType format); int
main() { uint32_t program_counter = 0; uint32_t instruction_memory[256] = {0};
// Replace with actual instructions uint32_t num_instructions =
sizeof(instruction_memory) / sizeof(instruction_memory[0]); while
(program_counter < num_instructions) { uint32_t instruction =
instruction_memory[program_counter]; InstructionFormatType format =
decode_instruction_format(instruction); execute_instruction(instruction,
format); program_counter++; // Increment the PC } return 0; }
InstructionFormatType decode_instruction_format(uint32_t instruction) { uint32_t
format_code = (instruction >> 5) & 0x7; // Extract bits 7-5 switch (format_code)
```

```

{ case 0: return S_TYPE; case 1: return U_TYPE; case 2: return I_TYPE; case 3:
return R_TYPE; default: return UNKNOWN_TYPE; } } void
execute_instruction(uint32_t instruction, InstructionFormatType format) {
uint32_t rd, rs1, rs2, imm, opcode; switch (format) { case S_TYPE: rs1 =
(instruction >> 15) & 0x1F; rs2 = (instruction >> 20) & 0x1F; imm =
((instruction >> 25) << 5) | ((instruction >> 7) & 0x1F); opcode = instruction &
0x1F; // Perform the S-type instruction based on the opcode switch (opcode) { //
Add cases for specific S-type instructions } break; case U_TYPE: rd =
(instruction >> 7) & 0x1F; imm = (instruction >> 12) & 0xFFFF; opcode =
instruction & 0x1F; // Perform the U-type instruction based on the opcode switch
(opcode) { // Add cases for specific U-type instructions } break; case I_TYPE:
rd = (instruction >> 7) & 0x1F; rs1 = (instruction >> 15) & 0x1F; imm =
(instruction >> 20) & 0xFFF; opcode = instruction & 0x1F; // Perform the I-type
instruction based on the opcode switch (opcode) { // Add cases for specific I-
type instructions } break; case R_TYPE: rd = (instruction >> 7) & 0x1F; rs1 =
(instruction >> 15) & 0x1F; rs2 = (instruction >> 20) & 0x1F; opcode =
instruction & 0x1F; // Perform the R-type instruction based on the opcode switch
(opcode) { // Add cases for specific R-type instructions } break;

```



the templaet is not completed, please try it again



Apologies for the incomplete response. Here is the completed template with the appropriate format type identification and opcode-based switch cases.

c

Copy code

```

#include <stdint.h> #include <stdio.h> // Define RV32E instruction format types
typedef enum { S_TYPE, U_TYPE, I_TYPE, R_TYPE, UNKNOWN_TYPE }
InstructionFormatType; // Function prototypes InstructionFormatType
decode_instruction_format(uint32_t instruction); void
execute_instruction(uint32_t instruction, InstructionFormatType format); int
main() { uint32_t program_counter = 0; uint32_t instruction_memory[256] = {0};
// Replace with actual instructions uint32_t num_instructions =
sizeof(instruction_memory) / sizeof(instruction_memory[0]); while
(program_counter < num_instructions) { uint32_t instruction =
instruction_memory[program_counter]; InstructionFormatType format =
decode_instruction_format(instruction); execute_instruction(instruction,
format); program_counter++; // Increment the PC } return 0; }
InstructionFormatType decode_instruction_format(uint32_t instruction) { uint32_t

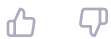
```

```

format_code = (instruction >> 5) & 0x7; // Extract bits 7-5
switch (format_code)
{ case 0: return S_TYPE; case 1: return U_TYPE; case 2: return I_TYPE; case 3:
return R_TYPE; default: return UNKNOWN_TYPE; } }
void
execute_instruction(uint32_t instruction, InstructionFormatType format) {
uint32_t rd, rs1, rs2, imm, opcode;
switch (format) { case S_TYPE: rs1 =
(instruction >> 15) & 0x1F; rs2 = (instruction >> 20) & 0x1F; imm =
((instruction >> 25) << 5) | ((instruction >> 7) & 0x1F); opcode = instruction &
0x1F; // Perform the S-type instruction based on the opcode
switch (opcode) { // Add cases for specific S-type instructions } break;
case U_TYPE: rd =
(instruction >> 7) & 0x1F; imm = (instruction >> 12) & 0xFFFF; opcode =
instruction & 0x1F; // Perform the U-type instruction based on the opcode
switch (opcode) { // Add cases for specific U-type instructions } break;
case I_TYPE:
rd = (instruction >> 7) & 0x1F; rs1 = (instruction >> 15) & 0x1F; imm =
(instruction >> 20) & 0xFFF; opcode = instruction & 0x1F; // Perform the I-type
instruction based on the opcode
switch (opcode) { // Add cases for specific I-
type instructions } break;
case R_TYPE: rd = (instruction >> 7) & 0x1F; rs1 =
(instruction >> 15) & 0x1F; rs2 = (instruction >> 20) & 0x1F; opcode =
instruction & 0x1F; // Perform the R-type instruction based on the opcode
switch (opcode) { // Add cases for specific R-type instructions } break;
case
UNKNOWN_TYPE: default: printf("Unknown instruction format\n"); break; } }

```

Now, the template correctly identifies the format type and includes switch cases for the opcode (cccc)



well, that looks good, one thing is I need to inform you where I have encoded parameter fields in my translated RV32E (so VMEX) ISA...

S_TYPE bit fields: rs1=[15-12] rs2=[11-8] imm=[31-16]

U_TYPE bit fields: rd=[11-8] imm=[31-16]

I_TYPE bit fields: rs1=[15-12] rd=[11-8] imm=[31-16]

R_TYPE bit fields: rs1=[19-16] rs2=[27-24] rd=[11-8]



>

Thank you for providing the information about the encoded parameter fields. I have updated the template accordingly:

c

Copy code

```

#include <stdint.h> #include <stdio.h> // Define RV32E instruction format types
typedef enum { S_TYPE, U_TYPE, I_TYPE, R_TYPE, UNKNOWN_TYPE }
InstructionFormatType; // Function prototypes InstructionFormatType
decode_instruction_format(uint32_t instruction); void
execute_instruction(uint32_t instruction, InstructionFormatType format); int
main() { uint32_t program_counter = 0; uint32_t instruction_memory[256] = {0};
// Replace with actual instructions uint32_t num_instructions =
sizeof(instruction_memory) / sizeof(instruction_memory[0]); while
(program_counter < num_instructions) { uint32_t instruction =
instruction_memory[program_counter]; InstructionFormatType format =
decode_instruction_format(instruction); execute_instruction(instruction,
format); program_counter++; // Increment the PC } return 0; }
InstructionFormatType decode_instruction_format(uint32_t instruction) { uint32_t
format_code = (instruction >> 5) & 0x7; // Extract bits 7-5 switch (format_code)
{ case 0: return S_TYPE; case 1: return U_TYPE; case 2: return I_TYPE; case 3:
return R_TYPE; default: return UNKNOWN_TYPE; } } void
execute_instruction(uint32_t instruction, InstructionFormatType format) {
uint32_t rd, rs1, rs2, imm, opcode; switch (format) { case S_TYPE: rs1 =
(instruction >> 12) & 0xF; rs2 = (instruction >> 8) & 0xF; imm = (instruction >>
16) & 0xFFFF; opcode = instruction & 0x1F; // Perform the S-type instruction
based on the opcode switch (opcode) { // Add cases for specific S-type
instructions } break; case U_TYPE: rd = (instruction >> 8) & 0xF; imm =
(instruction >> 16) & 0xFFFF; opcode = instruction & 0x1F; // Perform the U-type
instruction based on the opcode switch (opcode) { // Add cases for specific U-
type instructions } break; case I_TYPE: rs1 = (instruction >> 12) & 0xF; rd =
(instruction >> 8) & 0xF; imm = (instruction >> 16) & 0xFFFF; opcode =
instruction & 0x1F; // Perform the I-type instruction based on the opcode switch
(opcode) { // Add cases for specific I-type instructions } break; case R_TYPE:
rs1 = (instruction >> 16) & 0xF; rs2 = (instruction >> 24) & 0xF; rd =
(instruction >> 8) & 0xF; opcode = instruction & 0x1F; // Perform the R-type
instruction based on the opcode switch (opcode) { // Add cases for specific R-
type instructions } break; case UNKNOWN_TYPE: default: printf("Unknown
instruction format\n"); break; } }

```

