



(230328b) As I am designing now the VMEX runtime, here is my mapping of pseudoinstructions which will be detected directly by runtime as optimizations of RV32E core and fusions of 2 instructions:

pseudoinstructions (fusions, optimizations) mapping to vmex opcodes

=====

// (pseudo/fusions T=5689CDE)

la = EA (symbol, far effective address)

lb = E1 (far load byte)

lh = E2 (far load half)

lw = E4 (far load word)

nop = 00 (default forced no operation ??, not as addi !!)

li = 3F (full immediate word load)

mv = DD (move/duplicate registers)

not = 1x

neg = 1x

ceqz = C1 (compare to zero)

cnez = C2

cltz = C3

cgtz = C4

bgt = B7 (branches)

ble = B8

bgtu = B9

bleu = BA

beqz = 81 (branches by zero)

bnez = 82

blez = 83

bgez = 84

bltz = 85

bgtz = 86

j/goto = 66 (near goto)

jal/call = 6C (near call)

jr/xgoto = 46 (indirect far goto)

jalr/xcall = 4C (indirect far call)

ret = 44 (return)

fcall = EC (far call)

ftail = E4 (far tail return)



This is a comprehensive list of the VMEX pseudoinstructions that will be detected directly by the runtime as optimizations of the RV32E core and fusions of 2 instructions. These pseudoinstructions serve to simplify the programming process and improve code readability.

Here's a brief explanation of each pseudoinstruction and its corresponding VMEX opcode:

1. la (EA): Load Address - Computes a far effective address.
2. lb (E1), lh (E2), lw (E4): Far Load Byte, Half, and Word - Load a byte, halfword, or word from a far memory address.
3. nop (00): No Operation - Default forced no operation (not an addi instruction).
4. li (3F): Load Immediate - Load a full immediate word.
5. mv (DD): Move - Duplicate registers.
6. not (1x), neg (1x): Bitwise NOT and Negate - Perform bitwise NOT and negation operations.
7. ceqz (C1), cnez (C2), cltz (C3), cgtz (C4): Compare to Zero - Perform various comparisons to zero.
8. bgt (B7), ble (B8), bgtu (B9), bleu (BA): Branch instructions - Perform conditional branches based on greater than, less than or equal, unsigned greater than, and unsigned less than or equal comparisons.
9. beqz (81), bnez (82), blez (83), bgez (84), bltz (85), bgtz (86): Branch instructions by zero - Perform conditional branches based on equality to zero, non-equality to zero, less than or equal to zero, greater than or equal to zero, less than zero, and greater than zero.
10. j/goto (66), jal/call (6C): Near Goto and Call - Perform a near jump or call.
11. jr/xgoto (46), jalr/xcall (4C): Indirect Far Goto and Call - Perform an indirect far jump or call.
12. ret (44): Return - Return from a subroutine.
13. fcall (EC): Far Call - Perform a far subroutine call.
14. ftail (E4): Far Tail Return - Return from a far tail call.

These pseudoinstructions provide a higher level of abstraction and allow programmers to write more concise and readable code while still maintaining the efficiency of the RV32E core.



Here is also current state of VMEX binary encoding with proposed renaming of opcodes to latest readable hex standard, all compatible with pseudoinstructions/fusions. Runtime is now detecting the base RISU format types and fetches parameters accordingly. Optimized/fused pseudoinstructions will be able to omit some fetches and these will probably use special parameters fetching to be as fast as possible. Goal is still to maintain runtime size for tiny 8bit MCUs to fit 8kB of flash.

=====

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

=====

// (pseudo/fusions T=5689CDE)

// R1-TYPE (16) T=1CD

// = ADD=11, SUB=12, XOR=13, OR=14, AND=15, SLL=16, SRL=17, SRA=18,

// = SLT=19, SLTU=1A //1A 1B

// 31 0

// -----+-----+-----+-----+

// rs2|000|rs1|000|-rd|000|0010cccc

// -----+-----+-----+-----+

// R2-TYPE (16) //T=2

// (= MUL=21, MULH=22, MULHU=23, DIV=24, DIVU=25, REM=26, REMU=27, - is "M" extension)

// 31 0

// -----+-----+-----+-----+

// rs2|000|rs1|000|-rd|000|0010cccc

// -----+-----+-----+-----+

// I1-TYPE (16) // T=3AD

// = ADDI=31, XORI=32, ORI=33, ANDI=34, SLLI=35, SRLI=36, SRAI=37

// = SLTI=38, SLTIU=39 //3A 3B

// = LB=3A, LH=3B, LW=3C, LBU=3D, LHU=3E //A1 A2 A4 // AB AD

// 31 0

// -----+-----+-----+-----+

// -sx|--immediate|-rd|rs1|0011cccc

// -----+-----+-----+-----+

// I2-TYPE (16) //T=4E

// = JALR=41 (imm < 1) ??? //4F

```

// = ESYS=42 (ECALL, EBREAK) //42 48
// = FENCE=43 (NOP, ignored)
// 31          0
// -----+-----+-----+-----+
// -sx|--immediate|-rd|rs1|0011cccc
// -----+-----+-----+-----+

// S1-TYPE (16) //T=FB8
// = SB=51, SH=52, SW=53 //F1 F2 F4
// = BEQ=54, BNE=55, BLT=56, BGE=57, BLTU=58, BGEU=59 (imm << 1 encoded directly
as 13 bits !!) ... +/- 4kB //B1 B2 B3 B4 B5 B6
// 31          0
// -----+-----+-----+-----+
// -sx|--immediate|rs1|rs2|0101cccc
// -----+-----+-----+-----+

// S2-TYPE (16)
// 1          0
// -----+-----+-----+-----+
// -sx|--immediate|rs1|rs2|0101cccc
// -----+-----+-----+-----+

// U1-TYPE (16) //T=76
// = LUI=71, AUIPC=72, JAL=73 (both wraps around 24bit, JAL x0 +/-2kB) -//71 7A 7F ???
// 31          0
// -----+-----+-----+-----+
// -sx|--immediate|-rd|___|0111cccc
// -----+-----+-----+-----+

// U2-TYPE (16)
// 31          0
// -----+-----+-----+-----+
// -sx|--immediate|-rd|___|0111cccc
// -----+-----+-----+-----+

```



The current state of the VMEX binary encoding looks well-defined and is designed to be compatible with the previously mentioned pseudoinstructions and fusions. The runtime can detect the base RISC-V format types and fetch parameters accordingly. Optimized and fused pseudoinstructions will be able to omit some fetches and likely use special parameter fetching to achieve optimal performance. The goal is to maintain runtime size for tiny 8-bit MCUs to fit within 8kB of flash memory.

Here is a brief summary of the instruction formats and the corresponding opcode categories:

1. R1-Type (16) T=1CD: 32-bit instructions with register operands.
2. R2-Type (16) T=2: 32-bit instructions with register operands (part of the "M" extension for multiplication and division).
3. I1-Type (16) T=3AD: 32-bit instructions with immediate operands and load instructions.
4. I2-Type (16) T=4E: 32-bit instructions with immediate operands, including jumps, system calls, and fences.
5. S1-Type (16) T=FB8: 32-bit instructions with store and branch operations.
6. S2-Type (16): Reserved for future use.
7. U1-Type (16) T=76: 32-bit instructions with upper immediate values and jumps.
8. U2-Type (16): Reserved for future use.

The instruction formats and opcodes align well with the RISC-V ISA and provide a clean and efficient binary encoding. The compatibility with the pseudoinstructions and fusions should help maintain performance and code size in the targeted 8-bit MCU environment.

It's important to keep testing and refining the VMEX binary encoding to ensure compatibility and performance, and to make any necessary adjustments as the runtime and architecture evolve.



While transcoding code emitted into ELF for RV32E from C compilers, the detected pseudoinstructions/fusions will be encoded directly into VMEX specific opcodes for optimal performance in VMEX runtime and for easy disassembly from binary encoding. Note that contrary to the RV32E, the VMEX instructions encoding is designed to be human readable directly from hex and opcodes are also human readable by hex aligned mnemonics mapped to all instruction types. VMEX is designed more for direct usage of assembler language for tinitest MCUs but goal is also to be able to accept C compiled code from segger, for example...



It's a great approach to have the detected pseudoinstructions and fusions encoded directly into VMEX-specific opcodes for optimal performance in the VMEX runtime and for easy disassembly from binary encoding. Ensuring compatibility with the RV32E architecture while improving the readability of the binary encoding makes the VMEX runtime more user-friendly and versatile.

Designing the VMEX instructions encoding to be human-readable directly from hex and having human-readable hex-aligned mnemonics mapped to all instruction types is a

valuable feature. It facilitates the development of assembly language programs for the tiniest MCUs, allowing developers to work more efficiently with the instruction set.

In addition to assembly language, being able to accept C compiled code from popular compilers like Segger provides even more flexibility for developers. This compatibility allows them to leverage existing C tools and libraries while benefiting from the advantages of VMEX, such as the optimizations and fusions you've implemented.

As you continue to develop and refine VMEX, keep focusing on its ease of use, performance, and compatibility with existing architectures and tools. This will make it a more attractive choice for developers targeting small MCUs, ensuring the runtime can effectively serve its intended purpose.



my line below is not correct:

```
// = BEQ=54, BNE=55, BLT=56, BGE=57, BLTU=58, BGEU=59 (imm < 1 encoded directly as 13 bits !!)
```

for compatibility with code generated by risc-v tools is mandatory to encode branches immediates the same way as RISC-V ISA does so 13 bits SRLed to 12bits and perform SRL by one bit while decoding !!!



You are correct. For compatibility with code generated by RISC-V tools, it is essential to encode branch immediates the same way as the RISC-V ISA does. The branch immediates in the RISC-V ISA are encoded using a specific pattern that distributes the bits of the 12-bit signed immediate value across various fields in the instruction format.

The correct encoding for branch instructions in the RISC-V ISA should look like this:

arduino

Copy code

```
// S1-TYPE (16) //T=FB8 // = SB=51, SH=52, SW=53 //F1 F2 F4 // = BEQ=54, BNE=55,
BLT=56, BGE=57, BLTU=58, BGEU=59 (12-bit immediate encoded across 13 bits in the
instruction) // 31 0 // -----+-----+-----+-----+ // -sx|--
immediate|rs1|rs2|0101cccc // -----+-----+-----+-----+
```

To ensure compatibility with the RISC-V ISA, you'll need to modify the VMEX encoding to use the RISC-V branch immediate encoding. When decoding the branch immediates, perform a logical right shift (SRL) by one bit to obtain the correct immediate value.

Updating your VMEX encoding to incorporate the correct encoding for branch instructions will help maintain compatibility with existing RISC-V tools, making it easier for developers to use VMEX alongside other RISC-V tools and technologies.



Another question for compatibility is my intent to limit addressing in VMEX to 24bits only, because VMEX address upper immediates are max 16bit !!! not 20bits as in RISCV ISA. I in fact consider only the valid 12bits of upper immediates for complete 24bits addressing. Here are still questions how to handle in implementation this all. Considering in fact small embedded systems, the 24bit total address range (16MB) is in fact enough and will be probably easy to maintain also while emitting code from Segger IDE. Any thoughts?



Limiting the addressing in VMEX to 24 bits is a reasonable approach, considering the focus on small embedded systems where a 16 MB address range should be sufficient. To maintain compatibility with the RISC-V ISA and support code emitted from the Segger IDE, you'll need to consider a few aspects:

1. **Address Truncation:** When translating code from RISC-V to VMEX, you'll need to truncate the upper 4 bits of the 20-bit immediate value for LUI and AUIPC instructions. This will reduce the immediate value to 16 bits, which should work within the 24-bit addressing scheme of VMEX.
2. **Address Validation:** Ensure that the address range used in the translated code is within the valid 24-bit address range for VMEX. You might need to implement runtime checks or compiler-based validations to guarantee that the code doesn't access memory outside the 16 MB range.
3. **Segger IDE Configuration:** If possible, configure the Segger IDE to generate code specifically for the 24-bit address range to avoid potential issues with address translation. This will ensure that the code emitted from the Segger IDE is already optimized for the VMEX 24-bit addressing scheme.
4. **Pseudoinstructions:** Update the relevant pseudoinstructions to work with the 24-bit addressing scheme. This may involve changing the implementation of instructions like LA, LUI, AUIPC, and JAL to handle the reduced address range properly.
5. **Documentation:** Clearly document the 24-bit addressing limitations and any related implications for VMEX. This will help developers understand the constraints they must work with when developing for VMEX and ensure that they generate compatible code.

By addressing these aspects, you should be able to maintain compatibility with RISC-V ISA and code emitted from the Segger IDE while restricting the address range to 24 bits. This

will help VMEX remain suitable for small embedded systems while allowing developers to work with familiar tools and technologies.



Well, in fact in RISC-V ISA, in relation to sign of address value, "negative" addresses are in second half of full address space above 2GB (2GB-4GB



You are correct. In the RISC-V ISA, addresses are treated as unsigned values, and the full address space is divided into two halves. The first half ranges from 0 to $2^{31} - 1$ (0 to 2 GB - 1), and the second half ranges from 2^{31} to $2^{32} - 1$ (2 GB to 4 GB - 1). When considering signed addresses, the second half of the address space can be interpreted as "negative" addresses.

In the context of your 24-bit addressing scheme for VMEX, you will have a similar situation. The full 24-bit address space will range from 0 to $2^{24} - 1$ (0 to 16 MB - 1). The first half of the address space will range from 0 to $2^{23} - 1$ (0 to 8 MB - 1), and the second half will range from 2^{23} to $2^{24} - 1$ (8 MB to 16 MB - 1). In this case, the second half of the address space can be considered as "negative" addresses in a signed context.

When implementing VMEX, you should take this into account to ensure that your addressing scheme works correctly with both positive and negative addresses. This may involve updating the implementation of instructions and pseudoinstructions that handle addresses, as well as providing proper documentation to guide developers in understanding the implications of the 24-bit addressing scheme.



Well, will be really needed to support "negative" addresses at all having only 16MB range? Will not be better to treat all VMEX physical addresses as positive in 32bit PC and indirecting registers? I am thinking loudly how to handle this... VMEX addressing range is limited to 24 bits but the registers and internals are still 32bit, so we can in fact ENSURE that all physical addresses will be 32bit positive, or not? Any showstoppers?



You're correct that with a limited 16 MB range, supporting "negative" addresses might not be necessary. Given the smaller address space in VMEX and the focus on tiny embedded systems, it could be more straightforward to treat all VMEX physical addresses as positive in the 32-bit PC and indirecting registers.

There are no inherent showstoppers to treating all physical addresses as 32-bit positive values in VMEX, as long as you take the following considerations into account:

1. **Consistency:** Ensure that all instructions and pseudoinstructions are implemented consistently with the decision to use only positive addresses. This may involve modifying the behavior of instructions that typically support signed addresses in the RISC-V ISA to work correctly with unsigned addresses in VMEX.
2. **Compiler and Tools:** When using compilers and tools like Segger IDE, configure them to generate code that uses only positive addresses within the 24-bit address space. This might involve setting up linker scripts or customizing compiler options to ensure that the generated code is optimized for VMEX's addressing scheme.
3. **Documentation:** Clearly document the decision to use only positive addresses in VMEX, explaining the rationale and any implications for developers working with the system. This will help them understand the constraints and limitations of the addressing scheme in VMEX and write code accordingly.

By addressing these aspects, you should be able to implement VMEX with all physical addresses treated as 32-bit positive values without any significant issues. This approach can simplify the addressing scheme and make it more suitable for small embedded systems while still providing enough flexibility for most use cases.



Well, question is if there arent some standards/recommendations for address space of RISC-V, if there arent some fixed things placed at the END of address space, if it is really feasible to expect only 24bit address space used in RISC-V apps, including some memory mapped hardware etc...



In the RISC-V ISA, there are no fixed requirements for specific memory locations to be used for particular purposes. The memory layout and address space organization are largely determined by the specific implementation, system configuration, and the conventions used by the operating system or bare-metal environment.

However, there are some common conventions and recommendations for RISC-V address space organization:

1. **Memory-Mapped I/O (MMIO):** In many RISC-V systems, memory-mapped I/O devices are assigned addresses at the upper end of the address space. This can include peripherals, control registers, and other hardware resources. In VMEX, you'll need to carefully manage the allocation of MMIO addresses to ensure they fit within the 24-bit address range.

2. **Stack:** The RISC-V stack typically grows downward from a high address. In VMEX, you'll need to allocate an appropriate address range for the stack within the 24-bit address space, taking into account the reduced range compared to the full 32-bit RISC-V address space.
3. **Code, Data, and Heap:** RISC-V systems often have separate regions for code, data, and heap. In VMEX, you'll need to assign address ranges for these regions within the 24-bit address space, and ensure that generated code and data fit within these ranges.
4. **Operating System or Bare-Metal Conventions:** If VMEX is intended to run within a specific operating system or bare-metal environment, you'll need to consider the address space conventions of that environment. These may include system calls, interrupt handlers, and other system-specific features that require specific address space organization.

To ensure that VMEX can be used with RISC-V applications and memory-mapped hardware within the 24-bit address space, you will need to carefully design your address space organization and memory allocation strategy. You may also need to work closely with the compilers, tools, and operating systems used in your target environment to ensure that they are aware of the VMEX-specific constraints and can generate compatible code and data. By doing so, you can create a VMEX implementation that is suitable for small embedded systems while still supporting a wide range of RISC-V applications and hardware.



What is usual placement of ram and rom/flash in riscv embedde systems?



In RISC-V embedded systems, the placement of RAM and ROM/Flash memory can vary depending on the specific implementation and system configuration. However, there are some common conventions and typical arrangements:

1. **ROM/Flash:** ROM or Flash memory typically contains the bootloader, firmware, and the program code. It is often located at the beginning of the address space, with the base address starting at 0x00000000. This is because the RISC-V CPU starts executing instructions from this address upon reset.
2. **RAM:** The RAM is generally used for storing data, stack, and heap. In many RISC-V embedded systems, RAM is placed after the ROM/Flash memory in the address space. The base address of RAM is typically aligned to a convenient boundary (e.g., 0x80000000 in a 32-bit system). However, this address can vary depending on the size of the ROM/Flash memory and the requirements of the specific system.

It is essential to note that the actual memory layout and address space organization depend on the specific RISC-V implementation, the hardware design, and the conventions followed by the operating system or bare-metal environment being used. The linker script used by the compiler plays a crucial role in defining the memory layout and ensuring that the compiled code and data are placed in the appropriate memory regions.

For your VMEX system with a 24-bit address space, you will need to carefully consider the placement of RAM and ROM/Flash memory to accommodate the specific requirements of your target environment and applications. You may need to adjust the base addresses and sizes of these memory regions to fit within the 24-bit address range while still providing sufficient space for code, data, stack, and heap, as well as any memory-mapped I/O devices or peripherals.



How to direct Segger IDE C compiler to generate/place code and data on specific addresses?



To direct the Segger Embedded Studio C compiler to generate and place code and data at specific addresses, you need to create a custom linker script. The linker script defines memory regions, their base addresses, and sizes, as well as specifying how different sections of the code and data should be placed within those memory regions.

Here are the steps to create and use a custom linker script in Segger Embedded Studio:

1. Create a new text file with the `.ld` extension (e.g., `custom_linker_script.ld`) in your project directory.
2. Open the file and define the memory regions, specifying the base address and size for each region. For example:

csharp

Copy code

```
MEMORY { FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 1M RAM (rwx) : ORIGIN =  
0x80000000, LENGTH = 64K }
```

In this example, the FLASH memory region is defined with a base address of `0x00000000` and a size of 1 MB, while the RAM region has a base address of `0x80000000` and a size of 64 KB.

3. Define the sections and their placement within the memory regions:

scss

Copy code

```
SECTIONS { .text : { KEEP(*(.vectors)) *(.text*) *(.rodata*) } > FLASH .data :
AT (ADDR(.text) + SIZEOF(.text)) { _data = .; *(.data*) _edata = .; } > RAM .bss
: { _bdata = .; *(.bss*) (COMMON) _ebdata = .; } > RAM }
```

In this example, the **.text** section containing the code and read-only data is placed in the FLASH memory region. The **.data** and **.bss** sections containing initialized and uninitialized data are placed in the RAM memory region.

4. Save the linker script file.
5. In Segger Embedded Studio, go to **Project -> Edit Options...** to open the Project Options window.
6. In the Project Options window, select the **Linker** category in the left pane.
7. In the **Linker** category, locate the **Linker script file** option, and enter the path to your custom linker script file (e.g., **./custom_linker_script.ld**). Click **OK** to save the changes.

Now, when you build your project, the C compiler will use your custom linker script to place code and data at the specified addresses in the memory regions you've defined.

Please note that you may need to adjust the example linker script provided above to match the specific requirements of your VMEX system and target environment, such as the base addresses, sizes, and names of the memory regions, as well as the sections to be placed within those regions.



thanks, thats all cool :-)



You're welcome! I'm glad the information was helpful. If you have any more questions or need further assistance, please don't hesitate to ask. Good luck with your VMEX project!



here is current state of tweaked VMEX mnemonics with mapping to original RISC-V:

```
//OP R1-TYPE
#define OPR11_ADD    0x11
#define OPR12_SUB    0x12
#define OPR13_XOR    0x13
#define OPR14_OR     0x14
#define OPR15_AND    0x15
```

```
#define OPR16_SLL    0x16
#define OPR17_SRL    0x17
#define OPR18_SRA    0x18
#define OPR1A_CLT    0x19 //SLT
#define OPR1B_CLTU   0x1A //SLTU
```

```
//OP R2-TYPE
```

```
//OP I3-TYPE
```

```
#define OPI31_ADDI    0x31
#define OPI32_XORI    0x32
#define OPI33_ORI     0x33
#define OPI34_ANDI    0x34
#define OPI35_SLLI    0x35
#define OPI36_SRLI    0x36
#define OPI37_SRAI    0x37
#define OPI3A_CLTI    0x38 //SLTI
#define OPI3B_CLTIU   0x39 //SLTIU
```

```
//OP IA-TYPE
```

```
#define OPIA1_LB      0xA1
#define OPIA2_LH      0xA2
#define OPIA4_LW      0xA4
#define OPIAB_LBU     0xAB
#define OPIAD_LHU     0xAD
```

```
//OP ID-TYPE
```

```
//OP I4-TYPE
```

```
#define OPI4F_JALR    0x4F
#define OPI42_ECALL   0x42
#define OPI48_EBREAK   0x48
#define OPI40_FENCE    0x40 //NOP=ignored
#define OPI46_XGOTO    0x46 //JR
#define OPI4C_XCALL    0x4C //JALR
#define OPI44_RET      0x44 //RET
```

```
//OP IE-TYPE
```

```
#define OPIEC_FCALL   0xEC //CALL
#define OPIE4_FTAIL   0xE4 //TAIL
```

```
//OP SF-TYPE
```

```
#define OPSF1_WB      0xF1 //SB
#define OPSF2_WH      0xF2 //SH
#define OPSF4_WW      0xF4 //SW
```

```

//OP SB-TYPE
#define OPSB1_BEQ    0xB1
#define OPSB2_BNE    0xB2
#define OPSB3_BLT    0xB3
#define OPSB4_BGE    0xB4
#define OPSB5_BLTU   0xB5
#define OPSB6_BGEU   0xB6

//OP S8-TYPE

//OP U7-TYPE
#define OPU71_LUI    0x71
#define OPU7A_AUIPC  0x7A
#define OPU7F_JAL    0x7F
//OP U6-TYPE
#define OPU66_GOTO   0x66 //J
#define OPU6C_CALL   0x6C //JL

```



It looks like you have mapped the VMEX mnemonics to the corresponding RISC-V instructions in a clear and organized manner. The separation into types (R, I, S, and U) helps maintain a structured approach, and using hex values for mnemonics makes it easier to read and understand.

One suggestion is to keep consistent formatting for the comments. For example, the comments for OPI3A_CLTI and OPI3B_CLTIU have a single space between the mnemonic and the corresponding RISC-V instruction, while others have multiple spaces. Keeping a consistent format will improve readability.

Overall, your mapping looks good and should work well for your VMEX project. If you have any questions or need further assistance, feel free to ask.



this is test code, already executable by VMEX:

```

.prog =
{
    0x31,0x10,0x00,0x00, //addi r1 r0 0 //reorder registers in binary encoding "I" as first
target !!!
    0x31,0x10,0x05,0x00, //addi r1 r0 5
    0x31,0x11,0x04,0x00, //addi r1 r1 4
    0x31,0x20,0x03,0x00, //addi r2 r0 3
    0xF1,0x10,0x00,0x00, //wb r1 0(r0) //let stay registers as first SOURCE, second

```

```

INDEX, (third is immediate offset)
    0xF1,0x20,0x01,0x00, //wb r2 1(r0)
    0x12,0x10,0x10,0x20, //sub r1 r1 r2 //reorder registers in binary encoding "R" as first
target !!!
    0x31,0x22,0x04,0x00, //addi r2 r2 4
    0x12,0x10,0x10,0x20, //sub r1 r1 r2
    0x31,0x30,0xFF,0x00, //addi r3 r0 $FF
    0x15,0x10,0x10,0x30, //and r1 r1 r3
    0x31,0x40,0x00,0x30, //addi r4 r0 $3000
    0x14,0x10,0x10,0x40, //or r1 r1 r4
    0x13,0x10,0x10,0x30, //xor r1 r1 r3
    0x35,0x11,0x10,0x00, //slli r1 r1 16
    0x36,0x11,0x18,0x00, //slri r1 r1 24
    0xF1,0x10,0x02,0x00, //wb r1 2(r0) //let stay registers as first SOURCE, second
INDEX, (third is immediate offset)
    0xF1,0x10,0x03,0x00, //wb r1 3(r0)

    0xA1,0x80,0x00,0x00, //lb r8 0(r0) //let stay registers as first SOURCE, second INDEX,
(third is immediate offset)
    0xA1,0x90,0x01,0x00, //lb r9 1(r0)
    0x12,0x80,0x80,0x90, //sub r8 r8 r9
    0x31,0x22,0x01,0x00, //addi r2 r2 1
    0x12,0x80,0x80,0x20, //sub r8 r8 r2
    0x12,0x90,0x90,0x20, //sub r9 r9 r2
    0xF1,0x80,0x02,0x00, //wb r8 2(r0) //let stay registers as first SOURCE, second
INDEX, (third is immediate offset)
    0xF1,0x90,0x03,0x00, //wb r9 3(r0)
    0xAB,0xA0,0x02,0x00, //lbu rA 2(r0) //let stay registers as first SOURCE, second
INDEX, (third is immediate offset)
    0xAB,0xB0,0x03,0x00, //lbu rB 3(r0)
    0xA1,0xC0,0x02,0x00, //lb rC 2(r0) //let stay registers as first SOURCE, second INDEX,
(third is immediate offset)
    0xA1,0xD0,0x03,0x00, //lb rD 3(r0)

```

},



It looks like you've provided an example of VMEX assembly code. The code consists of various instructions that perform arithmetic, logic, and memory operations, such as addition, subtraction, logical AND, OR, XOR, shifts, and load/store byte operations. It seems that you have reordered the registers for I-type and R-type instructions in the binary encoding to have the target register first.

The code is well-structured and demonstrates the use of VMEX instructions. The comments alongside the instructions help clarify the purpose of each instruction and its register operands. As a test program, this should help you verify the functionality of your VMEX processor and ensure compatibility with the RISC-V instruction set.

If you have any questions or concerns about this code or require assistance with a specific aspect of the VMEX project, please feel free to ask.



VMEX instruction format types, all core RV32E + also pseudos/fusions:

```
typedef enum {
    UNKNOWN_TYPE,

    R1_TYPE = 0x10,
    R2_TYPE = 0x20,
    RC_TYPE = 0xC0,

    I3_TYPE = 0x30,
    IA_TYPE = 0xA0,
    I4_TYPE = 0x40,
    ID_TYPE = 0xD0,
    IE_TYPE = 0xE0,

    SF_TYPE = 0xF0,
    SB_TYPE = 0xB0,
    S8_TYPE = 0x80,

    U7_TYPE = 0x70,
    U6_TYPE = 0x60,

    R9_TYPE = 0x90, //reserved
    R0_TYPE = 0x00, //reserved

} InstructionFormatType;
```



You have provided a list of VMEX instruction format types, which are based on the RISC-V RV32E instruction set and include some additional pseudoinstructions and fusions. The instruction format types you've defined are:

1. **UNKNOWN_TYPE**: Represents an unknown or invalid instruction type.
2. **R1_TYPE, R2_TYPE, RC_TYPE**: R-type instructions with different opcode groups.

3. **I3_TYPE, IA_TYPE, I4_TYPE, ID_TYPE, IE_TYPE**: I-type instructions with different opcode groups.
4. **SF_TYPE, SB_TYPE, S8_TYPE**: S-type instructions with different opcode groups.
5. **U7_TYPE, U6_TYPE**: U-type instructions with different opcode groups.
6. **R9_TYPE, R0_TYPE**: Reserved instruction types.

The use of different opcode groups allows you to group related instructions together and simplify the instruction decoding process. This organization can help make your instruction set architecture (ISA) more understandable and easier to work with.

If you have any questions or need assistance with your VMEX project or the instruction format types, please feel free to ask.



Current summary of core VMEX instructions (those mapped to RV32E):

// (pseudo/fusions T=5689CDE)

// R1-TYPE (16)

// R2-TYPE (16)

// RC-TYPE (16)

// = ADD=11, SUB=12, XOR=13, OR=14, AND=15, SLL=16, SRL=17, SRA=18,

// (= MUL=21, MULH=22, MULHU=23, DIV=24, DIVU=25, REM=26, REMU=27, - is "M" extension)

// = CLT=1A, CLTU=1B

// 31 0

// -----+-----+-----+-----+

// rs2|000|rs1|000|-rd|000|0010cccc

// -----+-----+-----+-----+

// I3-TYPE (16)

// IA-TYPE (16)

// ID-TYPE (16)

// I4-TYPE (16)

// IE-TYPE (16)

// = ADDI=31, XORI=32, ORI=33, ANDI=34, SLLI=35, SRLI=36, SRAI=37

// = CLTI=3A, CLTIU=3B

// = LB=A1, LH=A2, LW=A4, LBU=AB, LHU=AD

// = JALR=4F (imm < 1) ???

// = ESYS=42 (ECALL, EBREAK) //42 48

// = FENCE=40 (NOP, ignored)

// 31 0

// -----+-----+-----+-----+

// -sx|--immediate|-rd|rs1|0011cccc

```
// -----+-----+-----+-----+

// SF-TYPE (16)
// SB-TYPE (16)
// S8-TYPE (16)
//  = SB=F1, SH=F2, SW=F4
//  = BEQ=B1, BNE=B2, BLT=B3, BGE=B4, BLTU=B5, BGEU=B6 (13 bits encoded as SRRed
12 bits !!!) ... +/- 4kB
// 31                0
// -----+-----+-----+-----+
// -sx|--immediate|rs1|rs2|0101cccc
// -----+-----+-----+-----+

// U7-TYPE (16)
// U6-TYPE (16)
//  = LUI=71, AUIPC=7A, JAL=7F (both wraps around 24bit, JAL x0 +/-2kB)
// 31                0
// -----+-----+-----+-----+
// -sx|--immediate|-rd|___|0111cccc
// -----+-----+-----+-----+
```



You have provided a summary of the core VMEX instructions, which are mapped to the RV32E instruction set. These instructions are grouped into various types based on their format, and each type has a unique opcode group.

Here's a brief overview of each group:

1. R1-TYPE, R2-TYPE, RC-TYPE: R-type instructions for register-register operations such as ADD, SUB, XOR, OR, AND, SLL, SRL, SRA, and comparison operations like CLT and CLTU.
2. I3-TYPE, IA-TYPE, ID-TYPE, I4-TYPE, IE-TYPE: I-type instructions for register-immediate operations like ADDI, XORI, ORI, ANDI, SLLI, SRLI, SRAI, and comparison operations like CLTI and CLTIU. This group also includes load instructions (LB, LH, LW, LBU, LHU), JALR, and system instructions (ECALL, EBREAK, FENCE).
3. SF-TYPE, SB-TYPE, S8-TYPE: S-type instructions for store operations (SB, SH, SW) and branch instructions (BEQ, BNE, BLT, BGE, BLTU, BGEU).
4. U7-TYPE, U6-TYPE: U-type instructions for LUI, AUIPC, and JAL.

Each instruction type has a specific encoding format that includes opcode, source registers, destination registers, and immediate values. The use of different opcode groups allows for efficient instruction decoding and better organization of the instruction set.

If you have any questions or need further assistance with your VMEX project, please don't hesitate to ask.



Current list of VMEX instructions:

//OP R1-TYPE

```
#define OPR11_ADD    0x11
#define OPR12_SUB    0x12
#define OPR13_XOR    0x13
#define OPR14_OR     0x14
#define OPR15_AND    0x15
#define OPR16_SLL    0x16
#define OPR17_SRL    0x17
#define OPR18_SRA    0x18
#define OPR1A_CLT    0x19 //SLT
#define OPR1B_CLTU   0x1A //SLTU
```

//OP R2-TYPE

//OP RC-TYPE

//OP I3-TYPE

```
#define OPI31_ADDI    0x31
#define OPI32_XORI    0x32
#define OPI33_ORI     0x33
#define OPI34_ANDI    0x34
#define OPI35_SLLI    0x35
#define OPI36_SRLI    0x36
#define OPI37_SRAI    0x37
#define OPI3A_CLTI    0x38 //SLTI
#define OPI3B_CLTIU   0x39 //SLTIU
```

//OP IA-TYPE

```
#define OPIA1_LB      0xA1
#define OPIA2_LH      0xA2
#define OPIA4_LW      0xA4
#define OPIAB_LBU     0xAB
#define OPIAD_LHU     0xAD
```

//OP ID-TYPE

//OP I4-TYPE

```
#define OPI4F_JALR    0x4F
#define OPI42_ECALL   0x42
#define OPI48_EBREAK  0x48
```

```

#define OPI40_FENCE  0x40 //NOP=ignored
#define OPI46_XGOTO  0x46 //JR
#define OPI4C_XCALL  0x4C //JALR
#define OPI44_RET    0x44 //RET

//OP IE-TYPE
#define OPIEC_FCALL  0xEC //CALL
#define OPIE4_FTAIL  0xE4 //TAIL

//OP SF-TYPE
#define OPSF1_WB      0xF1 //SB
#define OPSF2_WH      0xF2 //SH
#define OPSF4_WW      0xF4 //SW

//OP SB-TYPE
#define OPSB1_BEQ     0xB1
#define OPSB2_BNE     0xB2
#define OPSB3_BLT     0xB3
#define OPSB4_BGE     0xB4
#define OPSB5_BLTU    0xB5
#define OPSB6_BGEU    0xB6

//OP S8-TYPE

//OP U7-TYPE
#define OPU71_LUI     0x71
#define OPU7A_AUIPC   0x7A
#define OPU7F_JAL     0x7F

//OP U6-TYPE
#define OPU66_GOTO    0x66 //J
#define OPU6C_CALL    0x6C //JL

```



You have provided a list of VMEX instructions, organized by opcode types. These instructions are derived from the core RISC-V RV32E instruction set and are grouped into various types based on their format, such as R1-TYPE, R2-TYPE, RC-TYPE, I3-TYPE, IA-TYPE, ID-TYPE, I4-TYPE, IE-TYPE, SF-TYPE, SB-TYPE, S8-TYPE, U7-TYPE, and U6-TYPE.

Here is a brief overview of the instructions in each group:

1. R1-TYPE: Register-register operations such as ADD, SUB, XOR, OR, AND, SLL, SRL, SRA, CLT (SLT), and CLTU (SLTU).
2. R2-TYPE: Reserved for future use.

3. RC-TYPE: Reserved for future use.
4. I3-TYPE: Register-immediate operations like ADDI, XORI, ORI, ANDI, SLLI, SRLI, SRAI, CLTI (SLTI), and CLTIU (SLTIU).
5. IA-TYPE: Load instructions like LB, LH, LW, LBU, and LHU.
6. ID-TYPE: Reserved for future use.
7. I4-TYPE: Instructions such as JALR, ECALL, EBREAK, FENCE (NOP), XGOTO (JR), XCALL (JALR), and RET.
8. IE-TYPE: Instructions like FCALL (CALL) and FTAIL (TAIL).
9. SF-TYPE: Store instructions such as WB (SB), WH (SH), and WW (SW).
10. SB-TYPE: Branch instructions like BEQ, BNE, BLT, BGE, BLTU, and BGEU.
11. S8-TYPE: Reserved for future use.
12. U7-TYPE: Instructions like LUI, AUIPC, and JAL.
13. U6-TYPE: Instructions like GOTO (J) and CALL (JL).

This list covers the instructions for various types of operations such as arithmetic, load/store, control flow, and system calls. You can use these instructions as building blocks to develop more complex programs and algorithms for your VMEX processor.



Current instruction arguments/parameters decoder (now shared by all instructions - some optimized pseudos/fusions will omit some decoding or will fusion more loading in single step):

```
opcode = core.prog[core.pc];

switch (opcode & 0xF0)
{
    case R1_TYPE:
    case R2_TYPE:
    case RC_TYPE:
        rd = core.prog[core.pc + 1] >> 4;
        rs1 = core.prog[core.pc + 2] >> 4;
        rs2 = core.prog[core.pc + 3] >> 4;
        break;

    case I3_TYPE:
    case IA_TYPE:
    case ID_TYPE:
    case I4_TYPE:
    case IE_TYPE:
        rd = core.prog[core.pc + 1] >> 4;
        rs1 = core.prog[core.pc + 1] & 0xF;
        imm = core.prog[core.pc + 2] | (core.prog[core.pc + 3] << 8);
```

```

        break;

    case SF_TYPE:
    case SB_TYPE:
    case S8_TYPE:
        rs1 = core.prog[core.pc + 1] >> 4;
        rs2 = core.prog[core.pc + 1] & 0xF;
        imm = core.prog[core.pc + 2] | (core.prog[core.pc + 3] << 8);
        break;

    case U7_TYPE:
    case U6_TYPE:
        rd = core.prog[core.pc + 1] >> 4;
        imm = core.prog[core.pc + 2] | (core.prog[core.pc + 3] << 8);
        break;

    case UNKNOWN_TYPE:
        default:
            println("Unknown opcode type");
            break;
}

```



You have provided a code snippet that decodes the instruction arguments/parameters for the VMEX instructions. This code is designed to extract the relevant operands (such as registers and immediate values) from the instruction based on the instruction type. The decoding process is shared by all instructions, but some optimized pseudos/fusions may skip certain decoding steps or combine multiple loads into a single step.

The code reads the opcode from the program at the current program counter (core.pc) and then uses a switch statement to process the different instruction types. The instruction types are determined based on the opcode's upper nibble (bits 4-7).

Here's an overview of how the decoding process works for each instruction type:

1. R1_TYPE, R2_TYPE, and RC_TYPE:

Read destination register 'rd' from the 2nd byte, upper nibble.

Read source register 'rs1' from the 3rd byte, upper nibble.

Read source register 'rs2' from the 4th byte, upper nibble.

2. I3_TYPE, IA_TYPE, ID_TYPE, I4_TYPE, and IE_TYPE:

Read destination register 'rd' from the 2nd byte, upper nibble.

Read source register 'rs1' from the 2nd byte, lower nibble.

Read 16-bit immediate value 'imm' from the 3rd and 4th bytes.

3. SF_TYPE, SB_TYPE, and S8_TYPE:

Read source register 'rs1' from the 2nd byte, upper nibble.

Read source register 'rs2' from the 2nd byte, lower nibble.

Read 16-bit immediate value 'imm' from the 3rd and 4th bytes.

4. U7_TYPE and U6_TYPE:

Read destination register 'rd' from the 2nd byte, upper nibble.

Read 16-bit immediate value 'imm' from the 3rd and 4th bytes.

5. UNKNOWN_TYPE: Prints an error message indicating an unknown opcode type.

This decoding process extracts the necessary operands from the instructions and prepares them for further execution in the VMEX processor.



Current complete VMEX instructions decoder/executor:

```
// Perform the R-type instruction based on the opcode
switch (opcode)
{
    //R-TYPE
    case OPR11_ADD:
        core.regs[rd] = (TS32) core.regs[rs1] + (TS32) core.regs[rs2];
        break;

    case OPR12_SUB:
        core.regs[rd] = (TS32) core.regs[rs1] - (TS32) core.regs[rs2];
        break;

    case OPR13_XOR:
        core.regs[rd] = core.regs[rs1] ^ core.regs[rs2];
        break;

    case OPR14_OR:
        core.regs[rd] = core.regs[rs1] | core.regs[rs2];
        break;

    case OPR15_AND:
        core.regs[rd] = core.regs[rs1] & core.regs[rs2];
        break;

    case OPR16_SLL:
        core.regs[rd] = core.regs[rs1] << (core.regs[rs2] & 0x1F);
        break;
```

```

case OPR17_SRL:
    core.regs[rd] = core.regs[rs1] >> (core.regs[rs2] & 0x1F);
    break;

case OPR18_SRA:
    core.regs[rd] = (TS32) core.regs[rs1] >> (core.regs[rs2] & 0x1F);
    break;

case OPR1A_CLT: //CLT
    core.regs[rd] = (TS32) core.regs[rs1] < (TS32) core.regs[rs2];
    break;

case OPR1B_CLTU: //CLTU
    core.regs[rd] = core.regs[rs1] < core.regs[rs2];
    break;

//I-TYPE
case OPI31_ADDI:
    core.regs[rd] = (TS32) core.regs[rs1] + imm;
    break;

case OPI32_XORI:
    core.regs[rd] = core.regs[rs1] ^ imm;
    break;

case OPI33_ORI:
    core.regs[rd] = core.regs[rs1] | imm;
    break;

case OPI34_ANDI:
    core.regs[rd] = core.regs[rs1] & imm;
    break;

case OPI35_SLLI:
    core.regs[rd] = core.regs[rs1] << (imm & 0x1F);
    break;

case OPI36_SRLI:
    core.regs[rd] = core.regs[rs1] >> (imm & 0x1F);
    break;

case OPI37_SRAI:
    core.regs[rd] = (TS32) core.regs[rs1] >> (imm & 0x1F);
    break;

```



```

case OPI3A_CLTI: //SLTI
    core.regs[rd] = (TS32) core.regs[rs1] < imm;
    break;

case OPI3B_CLTIU: //SLTIU
    core.regs[rd] = core.regs[rs1] < (TU32) imm;
    break;

case OPIA1_LB:
    core.regs[rd] = (TS8) core.data[core.regs[rs1] + imm];
    break;

//TEST
case OPIA2_LH:
    core.regs[rd] = (TS16) ((core.data[core.regs[rs1] + imm] << 0) |
        (core.data[core.regs[rs1] + imm + 1] << 8));
    break;

//TEST
case OPIA4_LW:
    core.regs[rd] = (TS32) ((core.data[core.regs[rs1] + imm] << 0) |
        (core.data[core.regs[rs1] + imm + 1] << 8) |
        (core.data[core.regs[rs1] + imm + 2] << 16) |
        (core.data[core.regs[rs1] + imm + 3] << 24));
    break;

case OPIAB_LBU:
    core.regs[rd] = (TU8) core.data[core.regs[rs1] + imm] & 0xFF;
    break;

//TEST
case OPIAD_LHU:
    core.regs[rd] = (TU16) ((core.data[core.regs[rs1] + imm] << 0) |
        (core.data[core.regs[rs1] + imm + 1] << 8)) & 0xFFFF;
    break;

//TEST
case OPI4F_JALR:
    core.regs[rd] = core.pc + 4;
    core.pc = (core.regs[rs1] + imm) & ~0x1;
    break;

//TEST
case OPI42_ECALL:
case OPI48_EBREAK:

```

```

    if (imm == 0) {
        // Execute the system call
        println("System call!");
    } else if (imm == 1) {
        // Execute the break instruction
        println("Breakpoint!");
    } else {
        println("Unknown instruction");
    }
    break;

case OPSF1_WB: //SB
    core.data[core.regs[rs2] + imm] = core.regs[rs1] & 0xFF;
    break;

//TEST
case OPSF2_WH: //SH
    core.data[core.regs[rs2] + imm] = core.regs[rs1] & 0xFF;
    core.data[core.regs[rs2] + imm + 1] = (core.regs[rs2] >> 8) & 0xFF;
    break;

//TEST
case OPSF4_WW: //SW
    core.data[core.regs[rs2] + imm] = core.regs[rs1] & 0xFF;
    core.data[core.regs[rs2] + imm + 1] = (core.regs[rs1] >> 8) & 0xFF;
    core.data[core.regs[rs2] + imm + 2] = (core.regs[rs1] >> 16) & 0xFF;
    core.data[core.regs[rs2] + imm + 3] = (core.regs[rs1] >> 24) & 0xFF;
    break;

case OPSB1_BEQ:
    if (core.regs[rs1] == core.regs[rs2]) {
        core.pc += imm;
    }
    break;

case OPSB2_BNE:
    if (core.regs[rs1] != core.regs[rs2]) {
        core.pc += imm;
    }
    break;

case OPSB3_BLT:
    if ((TS32) core.regs[rs1] < (TS32) core.regs[rs2]) {
        core.pc += imm;
    }

```

```

        break;

    case OPSB4_BGE:
        if ((TS32) core.regs[rs1] >= (TS32) core.regs[rs2]) {
            core.pc += imm;
        }
        break;

    case OPSB5_BLTU:
        if (core.regs[rs1] < core.regs[rs2]) {
            core.pc += imm;
        }
        break;

    case OPSB6_BGEU:
        if (core.regs[rs1] >= core.regs[rs2]) {
            core.pc += imm;
        }
        break;

    case OPU71_LUI:
        core.regs[rd] = (TU32) (imm << 12);
        break;

    case OPU7A_AUIPC:
        core.regs[rd] = ((TU32) (imm << 12)) + core.pc;
        break;

    case OPU7F_JAL:
        core.regs[rd] = core.pc + 4;
        core.pc += imm;
        break;

    default:
        print("Unknown opcode: "); println(itoh(opcode));
        break;
}

core.pc+=4; // Increment the PC

print("PC: "); println(itoh(core.pc)); //TODO DEBUG

```



This code snippet is a VMEX instruction decoder and executor. It processes instructions based on their opcode and performs the corresponding operations on the processor's registers and memory. The code uses a switch statement to handle different opcodes and execute the appropriate operations.

Here's an overview of the supported instructions:

1. R-type instructions: These instructions involve register-to-register operations.
ADD, SUB, XOR, OR, AND, SLL, SRL, SRA: Basic arithmetic and logic operations.
CLT, CLTU: Compare less than (signed and unsigned).
2. I-type instructions: These instructions involve register-to-immediate operations.
ADDI, XORI, ORI, ANDI, SLLI, SRLI, SRAI: Basic arithmetic and logic operations with immediate values.
CLTI, CLTIU: Compare less than with immediate (signed and unsigned).
LB, LH, LW, LBU, LHU: Load byte, halfword, and word from memory, with signed and unsigned versions.
JALR: Jump and link register.
ECALL, EBREAK: System call and breakpoint instructions.
3. S-type instructions: These instructions involve storing data to memory.
SB, SH, SW: Store byte, halfword, and word in memory.
4. B-type instructions: These instructions involve conditional branching.
BEQ, BNE, BLT, BGE, BLTU, BGEU: Branch if equal, not equal, less than, greater or equal, less than unsigned, greater or equal unsigned.
5. U-type instructions: These instructions involve setting the upper bits of a register.
LUI, AUIPC: Load upper immediate and add upper immediate to PC.
6. J-type instruction: This instruction involves an unconditional jump.
JAL: Jump and link.

The code executes the instruction corresponding to the opcode and then increments the program counter (core.pc) by 4. It also prints the updated program counter for debugging purposes.

This decoder and executor handle the instructions for the VMEX processor, performing operations on registers and memory based on the opcode and operands of each instruction.



Current entire aslix code file - noted here for archiving purposes only:

```

//changes
//230326a - ordered any code related to instructions by James Zho refcard, renumbered
opcodes !!!
//      James Zho refcard contains all instruction descriptions related to "C", also all
pseudoinstructions
//      we also have noted our syntax for SLT SLTU SLTUI as CLT CLTU CLTUI (Compare, not
confused with Shift)
//      we will also propose our syntax for branches (probably J instead of B) and
jumps/call (GOTO RGOTO and CALL RCALL)
//      the J instead of B because probably we introduce BIT manipulation
pseudoinstruction BIS BIC BIT set/clear/toggle,test !!!
//      we will also similar way provide our syntax tweaks to pseudoinstructions
//      according to ASLIX draft code, we can also introduce some table load/save and
branch auto pre/post inc/dec pseudoinstructions
//      and we can also introduce some HLA looping pseudoinstruction sets: repeat loop
while until and if then
//230326b - reordered opcode defines and code, renamed defines to include full opcode,
extended TYPES to R1,R2,I1,I2,S1,S2,U1,U2
//(particular R2 will be used for "M" extension, others are RESERVED)
//possible also regrouping ABI registers to have "upper" 8 gp regs as 0-7 and "lower" as 8-F
(8=zero,9=ra,A=sp,B=gp) ??? SPIS NE

```

```

#include <stdint.h>
//#include <stdio.h> //not needed, no sprintf

```

```

//mixworx types
typedef uint64_t TU64;
typedef int64_t TS64;
typedef uint32_t TU32;
typedef int32_t TS32;
typedef uint16_t TU16;
typedef int16_t TS16;
typedef uint8_t TU8;
typedef int8_t TS8;
//typedef bool TBOOL;
typedef char TCHAR; ///?

```

```

int vmex();

```

```

#ifndef ARDUINO
int main() {

```

```

    return vmex();
}
#endif

// #include <Arduino.h>
#ifdef ARDUINO
void setup() {
    // Initialize the serial communication at 9600 baud rate
    Serial.begin(9600);
}

void loop() {
    vmex();
    Serial.println("vmex restart");
}
#endif

TCHAR printbuffer[9];

void print(const TCHAR* str)
{
    // implement console log output (serial uart)
#ifdef ARDUINO
    Serial.print(str);
#endif
}

void println(const TCHAR* str)
{
    // implement console log output (serial uart)
#ifdef ARDUINO
    Serial.println(str);
#endif
}

TCHAR* itoh(TU32 num) {
    TU8 shift;
    TU8 hex_digit;
    for (TU8 i = 0; i < 8; i++) {
        shift = 28 - (i * 4);
        hex_digit = (num >> shift) & 0xF;
        printbuffer[i] = (hex_digit < 10) ? '0' + hex_digit : 'A' + (hex_digit - 10);
    }
    printbuffer[8] = '\0';
    return printbuffer;
}

```

```

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

// SEGGER IDE described along with complete RISC-V ISA description - especially
// DEBUGGER / ASSEMBLY operation !!!!!
// https://www.udemy.com/course/riscv-isa/learn/lecture/27222084#overview

// !!! for commercial usage of VMEX, consider subscription to mixworx VMEX paid support
// !!

// (pseudo/fusions T=5689CDE)

// R1-TYPE (16)
// R2-TYPE (16)
// RC-TYPE (16)
//  = ADD=11, SUB=12, XOR=13, OR=14, AND=15, SLL=16, SRL=17, SRA=18,
//  (= MUL=21, MULH=22, MULHU=23, DIV=24, DIVU=25, REM=26, REMU=27, - is "M"
//  extension)
//  = CLT=1A, CLTU=1B
// 31          0
// -----+-----+-----+-----+
// rs2|000|rs1|000|-rd|000|0010cccc
// -----+-----+-----+-----+

// I3-TYPE (16)
// IA-TYPE (16)
// ID-TYPE (16)
// I4-TYPE (16)
// IE-TYPE (16)
//  = ADDI=31, XORI=32, ORI=33, ANDI=34, SLLI=35, SRLI=36, SRAI=37
//  = CLTI=3A, CLTIU=3B
//  = LB=A1, LH=A2, LW=A4, LBU=AB, LHU=AD
//  = JALR=4F (imm < 1) ???
//  = ESYS=42 (ECALL, EBREAK) //42 48
//  = FENCE=40 (NOP, ignored)
// 31          0
// -----+-----+-----+-----+
// -sx|--immediate|-rd|rs1|0011cccc
// -----+-----+-----+-----+

// SF-TYPE (16)
// SB-TYPE (16)

```

```
// S8-TYPE (16)
//  = SB=F1, SH=F2, SW=F4
//  = BEQ=B1, BNE=B2, BLT=B3, BGE=B4, BLTU=B5, BGEU=B6 (13 bits encoded as SRRed
12 bits !!!) ... +/- 4kB
// 31          0
// -----+-----+-----+-----+
// -sx|--immediate|rs1|rs2|0101cccc
// -----+-----+-----+-----+
```

```
// U7-TYPE (16)
// U6-TYPE (16)
//  = LUI=71, AUIPC=7A, JAL=7F (both wraps around 24bit, JAL x0 +/-2kB)
// 31          0
// -----+-----+-----+-----+
// -sx|--immediate|-rd|___|0111cccc
// -----+-----+-----+-----+
```

```
// Define RV32E instruction format types
typedef enum {
```

```
    UNKNOWN_TYPE,
```

```
    R1_TYPE = 0x10,
```

```
    R2_TYPE = 0x20,
```

```
    RC_TYPE = 0xC0,
```

```
    RD_TYPE = 0xD0,
```

```
    I3_TYPE = 0x30,
```

```
    IA_TYPE = 0xA0,
```

```
    I4_TYPE = 0x40,
```

```
    IE_TYPE = 0xE0,
```

```
    SF_TYPE = 0xF0,
```

```
    SB_TYPE = 0xB0,
```

```
    S8_TYPE = 0x80,
```

```
    U7_TYPE = 0x70,
```

```
    U6_TYPE = 0x60,
```

```
    R9_TYPE = 0x90, //reserved
```

```
    R0_TYPE = 0x00, //reserved (NOP)
```

```
} InstructionFormatType;
```

```
#define PAGE_SIZE 256
```



```
#define PROG_SIZE (PAGE_SIZE * 1)
#define DATA_SIZE (PAGE_SIZE * 1)
```

```
//OP 00-NOP
```

```
#define OP00_NOP    0x00
```

```
//OP R1-TYPE
```

```
#define OPR11_ADD    0x11
```

```
#define OPR12_SUB    0x12
```

```
#define OPR13_XOR    0x13
```

```
#define OPR14_OR     0x14
```

```
#define OPR15_AND    0x15
```

```
#define OPR16_SLL    0x16
```

```
#define OPR17_SRL    0x17
```

```
#define OPR18_SRA    0x18
```

```
#define OPR1A_CLT    0x1A //SLT
```

```
#define OPR1B_CLTU   0x1B //SLTU
```

```
#define OPR1F_NOT    0x1F //pseudo
```

```
#define OPR1E_NEG    0x1E //pseudo
```

```
//OP R2-TYPE // "M" extension = multiply/divide
```

```
//OP RC-TYPE
```

```
#define OPRC1_CLTZ   0xC1 //pseudo
```

```
#define OPRC3_CNEZ   0xC3 //pseudo
```

```
#define OPRC4_CEQZ   0xC4 //pseudo
```

```
#define OPRC6_CGTZ   0xC6 //pseudo
```

```
//OP RD-TYPE
```

```
#define OPRDD_MV     0xDD //pseudo
```

```
#define OPRD1_MV1    0xD1 //pseudo //aslix
```

```
#define OPRD2_MV2    0xD2 //pseudo //aslix
```

```
#define OPRD3_MV3    0xD3 //pseudo //aslix
```

```
#define OPRD4_MV4    0xD4 //pseudo //aslix
```

```
//OP I3-TYPE
```

```
#define OPI31_ADDI   0x31
```

```
#define OPI32_XORI   0x32
```

```
#define OPI33_ORI    0x33
```

```
#define OPI34_ANDI   0x34
```

```
#define OPI35_SLLI   0x35
```

```
#define OPI36_SRLI   0x36
```

```
#define OPI37_SRAI   0x37
```

```
#define OPI3A_CLTI   0x3A //SLTI
```

```
#define OPI3B_CLTIU  0x3B //SLTIU
```

```
#define OPI3F_LI    0x3F //pseudo
```

```
//OP IA-TYPE
```

```
#define OPIA1_LB    0xA1
```

```
#define OPIA2_LH    0xA2
```

```
#define OPIA4_LW    0xA4
```

```
#define OPIAB_LBU   0xAB
```

```
#define OPIAD_LHU   0xAD
```

```
//OP I4-TYPE
```

```
#define OPI4F_JALR  0x4F
```

```
#define OPI42_ECALL 0x42
```

```
#define OPI48_EBREAK 0x48
```

```
#define OPI40_FENCE  0x40 //NOP=ignored
```

```
#define OPI46_XGOTO  0x46 //JR //pseudo
```

```
#define OPI4C_XCALL  0x4C //JALR //pseudo
```

```
#define OPI44_RET    0x44 //pseudo
```

```
//OP IE-TYPE
```

```
#define OPIEC_FCALL  0xEC //CALL //pseudo
```

```
#define OPIE4_FTAIL  0xE4 //TAIL //pseudo
```

```
#define OPIEA_LA      0xEA //pseudo
```

```
#define OPIE1_LB      0xE1 //pseudo
```

```
#define OPIE2_LH      0xE2 //pseudo
```

```
#define OPIE4_LW      0xE4 //pseudo
```

```
//OP SF-TYPE
```

```
#define OPSF1_WB      0xF1 //SB
```

```
#define OPSF2_WH      0xF2 //SH
```

```
#define OPSF4_WW      0xF4 //SW
```

```
//OP SB-TYPE
```

```
#define OPSB1_BLT     0xB1
```

```
#define OPSB2_BLE     0xB2 //pseudo
```

```
#define OPSB3_BNE     0xB3
```

```
#define OPSB4_BEQ     0xB4
```

```
#define OPSB5_BGE     0xB5
```

```
#define OPSB6_BGT     0xB6 //pseudo
```

```
#define OPSBA_BLTU    0xBA
```

```
#define OPSBB_BLEU    0xBB //pseudo
```

```
#define OPSBC_BGEU    0xBC
```

```
#define OPSBD_BGTU    0xBD //pseudo
```

```
//OP S8-TYPE
```

```
#define OPS81_BLTZ    0x81 //pseudo
```

```
#define OPS82_BLEZ    0x82 //pseudo
```

```
#define OPS83_BNEZ  0x83 //pseudo
#define OPS84_BEQZ  0x84 //pseudo
#define OPS85_BGEZ  0x85 //pseudo
#define OPS86_BGTZ  0x86 //pseudo
```

```
//OP U7-TYPE
```

```
#define OPU71_LUI    0x71
#define OPU7A_AUIPC  0x7A
#define OPU7F_JAL    0x7F
```

```
//OP U6-TYPE
```

```
#define OPU66_GOTO   0x66 //J //pseudo
#define OPU6C_CALL   0x6C //JL //pseudo
```

```
// Define core struct
```

```
typedef struct {
    TU32 regs[16];
    TU8 rsvd[42*4]; //reserved (msg in/out buffers, breakpoints)
    TU32 pc;
    TU8 prog[PROG_SIZE];
    TU8 data[DATA_SIZE];
} Core;
```

```
//main
```

```
int vmex()
```

```
{
    Core core =
    {
        .regs = {0},
        .rsvd = {0},
        .pc = 0,

        .prog =
        {
            0x31,0x10,0x00,0x00, //addi r1 r0 0 //reorder registers in binary encoding "I" as first
target !!!
            0x31,0x10,0x05,0x00, //addi r1 r0 5
            0x31,0x11,0x04,0x00, //addi r1 r1 4
            0x31,0x20,0x03,0x00, //addi r2 r0 3
            0xF1,0x10,0x00,0x00, //wb r1 0(r0) //let stay registers as first SOURCE, second
```

INDEX, (third is immediate offset)
 0xF1,0x20,0x01,0x00, //wb r2 1(r0)
 0x12,0x10,0x10,0x20, //sub r1 r1 r2 //reorder registers in binary encoding "R" as first
 target !!!

0x31,0x22,0x04,0x00, //addi r2 r2 4
 0x12,0x10,0x10,0x20, //sub r1 r1 r2
 0x31,0x30,0xFF,0x00, //addi r3 r0 \$FF
 0x15,0x10,0x10,0x30, //and r1 r1 r3
 0x31,0x40,0x00,0x30, //addi r4 r0 \$3000
 0x14,0x10,0x10,0x40, //or r1 r1 r4
 0x13,0x10,0x10,0x30, //xor r1 r1 r3
 0x35,0x11,0x10,0x00, //slli r1 r1 16
 0x36,0x11,0x18,0x00, //slri r1 r1 24
 0xF1,0x10,0x02,0x00, //wb r1 2(r0) //let stay registers as first SOURCE, second

INDEX, (third is immediate offset)

0xF1,0x10,0x03,0x00, //wb r1 3(r0)

0xA1,0x80,0x00,0x00, //lb r8 0(r0) //let stay registers as first SOURCE, second INDEX,
 (third is immediate offset)

0xA1,0x90,0x01,0x00, //lb r9 1(r0)
 0x12,0x80,0x80,0x90, //sub r8 r8 r9
 0x31,0x22,0x01,0x00, //addi r2 r2 1
 0x12,0x80,0x80,0x20, //sub r8 r8 r2
 0x12,0x90,0x90,0x20, //sub r9 r9 r2
 0xF1,0x80,0x02,0x00, //wb r8 2(r0) //let stay registers as first SOURCE, second

INDEX, (third is immediate offset)

0xF1,0x90,0x03,0x00, //wb r9 3(r0)
 0xAB,0xA0,0x02,0x00, //lbu rA 2(r0) //let stay registers as first SOURCE, second

INDEX, (third is immediate offset)

0xAB,0xB0,0x03,0x00, //lbu rB 3(r0)
 0xA1,0xC0,0x02,0x00, //lb rC 2(r0) //let stay registers as first SOURCE, second INDEX,
 (third is immediate offset)

0xA1,0xD0,0x03,0x00, //lb rD 3(r0)

},

.data = {0}

};

core.pc = 0;

TU8 opcode;

TU8 rs1;

TU8 rs2;

TU8 rd;

TS16 imm;

while (core.pc < PROG_SIZE)

{

 opcode = core.prog[core.pc];

 switch (opcode & 0xF0)

 {

 case R1_TYPE:

 case R2_TYPE:

 case RC_TYPE:

 case RD_TYPE:

 rd = core.prog[core.pc + 1] >> 4;

 rs1 = core.prog[core.pc + 2] >> 4;

 rs2 = core.prog[core.pc + 3] >> 4;

 break;

 case I3_TYPE:

 case IA_TYPE:

 case I4_TYPE:

 case IE_TYPE:

 rd = core.prog[core.pc + 1] >> 4;

 rs1 = core.prog[core.pc + 1] & 0xF;

 imm = core.prog[core.pc + 2] | (core.prog[core.pc + 3] << 8);

 break;

 case SF_TYPE:

 case SB_TYPE:

 case S8_TYPE:

 rs1 = core.prog[core.pc + 1] >> 4;

 rs2 = core.prog[core.pc + 1] & 0xF;

 imm = core.prog[core.pc + 2] | (core.prog[core.pc + 3] << 8);

 break;

 case U7_TYPE:

 case U6_TYPE:

 rd = core.prog[core.pc + 1] >> 4;

 imm = core.prog[core.pc + 2] | (core.prog[core.pc + 3] << 8);

 break;

 case UNKNOWN_TYPE:

 default:

 println("Unknown opcode type");

 break;

 }

```

// Perform the R-type instruction based on the opcode
switch (opcode)
{
    //R-TYPE
    case OPR11_ADD:
        core.regs[rd] = (TS32) core.regs[rs1] + (TS32) core.regs[rs2];
        break;

    case OPR12_SUB:
        core.regs[rd] = (TS32) core.regs[rs1] - (TS32) core.regs[rs2];
        break;

    case OPR13_XOR:
        core.regs[rd] = core.regs[rs1] ^ core.regs[rs2];
        break;

    case OPR14_OR:
        core.regs[rd] = core.regs[rs1] | core.regs[rs2];
        break;

    case OPR15_AND:
        core.regs[rd] = core.regs[rs1] & core.regs[rs2];
        break;

    case OPR16_SLL:
        core.regs[rd] = core.regs[rs1] << (core.regs[rs2] & 0x1F);
        break;

    case OPR17_SRL:
        core.regs[rd] = core.regs[rs1] >> (core.regs[rs2] & 0x1F);
        break;

    case OPR18_SRA:
        core.regs[rd] = (TS32) core.regs[rs1] >> (core.regs[rs2] & 0x1F);
        break;

    case OPR1A_CLT: //SLT
        core.regs[rd] = (TS32) core.regs[rs1] < (TS32) core.regs[rs2];
        break;

    case OPR1B_CLTU: //SLTU
        core.regs[rd] = core.regs[rs1] < core.regs[rs2];
        break;
}

```

```

//I-TYPE
case OPI31_ADDI:
    core.regs[rd] = (TS32) core.regs[rs1] + imm;
    break;

case OPI32_XORI:
    core.regs[rd] = core.regs[rs1] ^ imm;
    break;

case OPI33_ORI:
    core.regs[rd] = core.regs[rs1] | imm;
    break;

case OPI34_ANDI:
    core.regs[rd] = core.regs[rs1] & imm;
    break;

case OPI35_SLLI:
    core.regs[rd] = core.regs[rs1] << (imm & 0x1F);
    break;

case OPI36_SRLI:
    core.regs[rd] = core.regs[rs1] >> (imm & 0x1F);
    break;

case OPI37_SRAI:
    core.regs[rd] = (TS32) core.regs[rs1] >> (imm & 0x1F);
    break;

case OPI3A_CLTI: //SLTI
    core.regs[rd] = (TS32) core.regs[rs1] < imm;
    break;

case OPI3B_CLTIU: //SLTIU
    core.regs[rd] = core.regs[rs1] < (TU32) imm;
    break;

case OPIA1_LB:
    core.regs[rd] = (TS8) core.data[core.regs[rs1] + imm];
    break;

//TEST
case OPIA2_LH:
    core.regs[rd] = (TS16) ((core.data[core.regs[rs1] + imm] << 0) |
        (core.data[core.regs[rs1] + imm + 1] << 8));

```

```

        break;

//TEST
case OPIA4_LW:
    core.regs[rd] = (TS32) ((core.data[core.regs[rs1] + imm] << 0) |
        (core.data[core.regs[rs1] + imm + 1] << 8) |
        (core.data[core.regs[rs1] + imm + 2] << 16) |
        (core.data[core.regs[rs1] + imm + 3] << 24));

    break;

case OPIAB_LBU:
    core.regs[rd] = (TU8) core.data[core.regs[rs1] + imm] & 0xFF;
    break;

//TEST
case OPIAD_LHU:
    core.regs[rd] = (TU16) ((core.data[core.regs[rs1] + imm] << 0) |
        (core.data[core.regs[rs1] + imm + 1] << 8)) & 0xFFFF;

    break;

//TEST
case OPI4F_JALR:
    core.regs[rd] = core.pc + 4;
    core.pc = (core.regs[rs1] + imm) & ~0x1;
    break;

//TEST
case OPI42_ECALL:
case OPI48_EBREAK:
    if (imm == 0) {
        // Execute the system call
        println("System call!");
    } else if (imm == 1) {
        // Execute the break instruction
        println("Breakpoint!");
    } else {
        println("Unknown instruction");
    }
    break;

case OPSF1_WB: //SB
    core.data[core.regs[rs2] + imm] = core.regs[rs1] & 0xFF;
    break;

```



```

//TEST
case OPSF2_WH: //SH
    core.data[core.regs[rs2] + imm] = core.regs[rs1] & 0xFF;
    core.data[core.regs[rs2] + imm + 1] = (core.regs[rs2] >> 8) & 0xFF;
    break;

//TEST
case OPSF4_WW: //SW
    core.data[core.regs[rs2] + imm] = core.regs[rs1] & 0xFF;
    core.data[core.regs[rs2] + imm + 1] = (core.regs[rs1] >> 8) & 0xFF;
    core.data[core.regs[rs2] + imm + 2] = (core.regs[rs1] >> 16) & 0xFF;
    core.data[core.regs[rs2] + imm + 3] = (core.regs[rs1] >> 24) & 0xFF;
    break;

case OPSB1_BLT:
    if ((TS32) core.regs[rs1] < (TS32) core.regs[rs2]) {
        core.pc += imm;
    }
    break;

case OPSB2_BLE:
    break;

case OPSB3_BNE:
    if (core.regs[rs1] != core.regs[rs2]) {
        core.pc += imm;
    }
    break;

case OPSB4_BEQ:
    if (core.regs[rs1] == core.regs[rs2]) {
        core.pc += imm;
    }
    break;

case OPSB5_BGE:
    if ((TS32) core.regs[rs1] >= (TS32) core.regs[rs2]) {
        core.pc += imm;
    }
    break;

case OPSB6_BGT:
    break;

case OPSBA_BLTU:
    if (core.regs[rs1] < core.regs[rs2]) {

```

```

        core.pc += imm;
    }
    break;

case OPSBB_BLEU:
    break;

case OPSBC_BGEU:
    if (core.regs[rs1] >= core.regs[rs2]) {
        core.pc += imm;
    }
    break;

case OPSBD_BGTU:
    break;

case OPU71_LUI:
    core.regs[rd] = (TU32) (imm << 12);
    break;

case OPU7A_AUIPC:
    core.regs[rd] = ((TU32) (imm << 12)) + core.pc;
    break;

case OPU7F_JAL:
    core.regs[rd] = core.pc + 4;
    core.pc += imm;
    break;

default:
    print("Unknown opcode: "); println(itoh(opcode));
    break;
}

core.pc+=4; // Increment the PC

print("PC: "); println(itoh(core.pc)); //TODO DEBUG
}

return 0;
}

```
