Model: GPT-4

(230325c) Decision to implement only RV32E in virtual processor interpreter VMEX, because RV32C decoder is really very complex and its hard to prvide efficient encoding for 16bit instructions. RV32E transformed and optimized is perfectly okay and will be easy to implement in compact way. We will have optimized encoding so will need to provide transpilation postprocessing from real RV32E(I) code to VMEX and back to use even C compiled code for RV32E target by Segger IDE for example. VMEX engine will be compact, will there be possibility to implement MUL/DIV/REM "M" extensions, altough also not mandatory and we will have NET MAUI coded virtual programmable VMEX "calculator" on all platforms. Any thoughts?

Implementing RV32E in the virtual processor interpreter VMEX is a reasonable decision, especially considering the complexity of RV32C decoding and the goal of achieving a compact and efficient implementation. RV32E, being a subset of RV32I, provides a simpler and more streamlined instruction set that should be easier to implement and optimize.

Transpilation postprocessing between real RV32E(I) code and VMEX should allow the use of C compiled code for RV32E targets, such as those generated by Segger IDE. This approach should increase compatibility with existing development tools and ecosystems.

Regarding the possibility of implementing the MUL/DIV/REM "M" extensions, it is not mandatory, but it could be beneficial in certain use cases, depending on the performance requirements and the target applications for the VMEX engine. If the primary use case is for a virtual programmable VMEX "calculator" on all platforms, implementing the "M" extension could provide additional functionality and improved performance for multiplication and division operations.

That said, the implementation of the "M" extension would likely increase the complexity and size of the VMEX engine. It is essential to weigh the benefits of this added functionality against the increased complexity and resource requirements of the engine.

In conclusion, adopting RV32E and optimizing it for the VMEX engine is a viable approach. The decision to implement the "M" extension depends on the desired functionality and performance trade-offs. I would recommend analyzing the target applications and user requirements to determine whether the additional complexity is warranted.

---

which may be code size of routines for 32bit multiply/divide/remainder for targets AVR, PIC16, PIC18, PIC24, 6502, Z80, 8085 ?

It is difficult to provide exact code sizes for multiply, divide, and remainder routines for each of the mentioned microcontroller targets, as they will depend on the specific implementation, optimization techniques, and compiler used. However, I can give you a rough estimate of the code size based on the architectural complexity and the instruction set of each target.

1. AVR: The AVR architecture has native support for 8-bit multiplication, so 32-bit multiplication, division, and remainder routines would require more operations. The code size can range between 100-200 bytes for each operation, depending on the implementation and optimization.
2. PIC16 and PIC18: These microcontrollers have 8-bit architectures and require additional instructions to perform 32-bit arithmetic operations. The code size for each operation can range between 150-300 bytes, depending on the implementation and optimization.
3. PIC24: As a 16-bit architecture, the PIC24 requires fewer instructions for 32-bit arithmetic operations than 8-bit microcontrollers. The code size for each operation can range between 100-200 bytes, depending on the implementation and optimization.
4. 6502, Z80, and 8085: These are 8-bit architectures with limited instruction sets, making the implementation of 32-bit arithmetic operations more challenging. The code size for each operation can range between 200-400 bytes, depending on the implementation and optimization.

Please note that these are rough estimates and can vary significantly based on the specific implementations, optimization techniques, and compilers used. To get more accurate numbers, you can implement the routines yourself or search for existing implementations tailored to each of the target platforms.

do you know microchip mplab harmony libraries/framework?

Yes, I am familiar with Microchip MPLAB Harmony. MPLAB Harmony is a flexible, fully integrated embedded software development framework for Microchip's PIC32 and SAM microcontrollers. It is designed to streamline and accelerate the development of embedded applications by providing a rich set of peripheral libraries, middleware, and drivers.

MPLAB Harmony provides a modular architecture, allowing developers to select and configure the components needed for their specific application. Some of the features and components included in MPLAB Harmony are:

1. Peripheral Libraries: These libraries provide an abstraction layer for the microcontroller's hardware, making it easier to develop and maintain application code by offering a consistent API across different devices and families.
2. Middleware: MPLAB Harmony includes various middleware components, such as USB, TCP/IP, and graphics libraries, which simplify the integration of complex functionality into the application.
3. Drivers: The framework includes drivers for a wide range of peripheral devices, such as I2C, SPI, and UART, that make it easier to interface with external hardware components.
4. Code Generation and Configuration Tools: MPLAB Harmony comes with tools that can generate and configure code based on the developer's requirements, speeding up the development process and reducing the chance of errors.
5. Real-Time Operating System (RTOS) support: The framework supports multiple RTOS options, such as FreeRTOS and Microchip's MPLAB Harmony RTOS, which can be integrated into the application to manage tasks, resources, and timing more efficiently.
6. Board Support Packages (BSPs): MPLAB Harmony offers BSPs for a variety of development boards, enabling easy access to board-specific features and configurations.

By using MPLAB Harmony, developers can save time and effort in developing embedded applications for Microchip's microcontrollers, focusing on their application logic rather than low-level hardware and software details.

do you know way how to inject into build system of say Seggeer IDE posprocessing of generated binary code for RV32I(E) to recode output binary to different ISA encoding compatible/mappable to RV32I(E)?

To inject post-processing into the build system for converting RV32I(E) binary output to a different ISA encoding compatible with RV32I(E), you can use custom build scripts or modify the build process. Here's a general outline of the process using a custom script:

1. Write a custom post-processing script or tool: Develop a script or tool in a language of your choice (e.g., Python, C, or C++) that takes the compiled RV32I(E) binary as input, converts the binary to your desired ISA encoding, and outputs the converted binary. This tool should be able to parse the input binary, identify the instructions, and convert them to the new encoding while preserving the original program logic.
2. Modify the build process: Integrate your custom post-processing tool into the build process of the Segger IDE or any other development environment you are using. You can achieve this by:
   a. Custom build step: Add a custom build step in your IDE to call your post-processing script or tool after the standard compilation and linking steps. In Segger Embedded Studio, you can add a custom build step by going to Project -> Edit Options -> User Build Step -> Post-build. Add the command to execute your script/tool, specifying input and output files as necessary.
   b. Makefile or build script: If your project uses a Makefile or a build script, add a rule or step that calls your post-processing script or tool after the compilation and linking steps.
3. Test and verify: Once you have integrated the post-processing tool into your build system, build your project and ensure that the post-processing step is executed as expected. Test the output binary on your target platform or in an emulator to verify that the conversion process has been successful and