RISC-V Graphics ISA

Preliminary Design
Who we are?

- We are a group of enthusiasts who espouse powerful but low-cost open-source hardware for students, makers, commercial users and anyone else interested in customizable hardware

- Atif Zafar (Pixilica)
  - www.pixilica.com

- Grant Jennings (GOWIN Semiconductor)
  - www.gowinsemi.com

- Ted Marena (CHIPS Alliance and Western Digital)
  - www.chipsalliance.org
RISC-V Graphics ISA

• 3D graphics is now a standard part of many processor designs (Intel, ARM, Qualcomm, Samsung etc.) for consumer devices

• The RISC-V ISA is rapidly gaining industry backing due to its open-source license

• Some industry support for graphics
  – Imagination Technologies PowerVR
  – Libre Open-Source GPU effort

• A key part of the RISC-V ISA is that it is “extensible”
RISC-V ISA

- RISC-V has incremental modular ISA’s that add functionality and complexity to a core design:
  - Base Integer: RV32I and RV64I
  - Add:
    - M: Integer Multiply/Divide
    - A: Atomic Instructions (for thread synchronization)
    - F: Single-Precision Floating Point
    - D: Double-Precision Floating Point
    - G: IMAFD (all of the above extensions combined)
    - C: Compressed Instruction Set (for atomic thread sync ops)
    - Q: 128-bit quad-wide floating point format

- V: Vector Extensions (proposed)
### Base Integer Instructions: RV32I, RV64I, and RV128I

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Fmt</th>
<th>RV32I Base</th>
<th>RV64I (12B)</th>
<th>RV128I</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Loads</strong></td>
<td>Load Byte</td>
<td>L</td>
<td>LB</td>
<td>rd, rs1, imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Load Halfword</td>
<td>L</td>
<td>LH</td>
<td>rd, rs1, imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Load Word</td>
<td>L</td>
<td>LW</td>
<td>rd, rs1, imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Load Byte Unsigned</td>
<td>L(D)</td>
<td>rd, rs1, imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Load Halfword Unsigned</td>
<td>L(W)D</td>
<td>rd, rs1, imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Stores</strong></td>
<td>Store Byte</td>
<td>S</td>
<td>SB</td>
<td>rd, rs1, imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Store Halfword</td>
<td>SH</td>
<td>SH</td>
<td>rd, rs1, imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Store Word</td>
<td>S</td>
<td>SW</td>
<td>rd, rs1, imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Left Immediate</td>
<td>SLLI</td>
<td>rd, rs1, rs2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Right</td>
<td>S</td>
<td>SRL</td>
<td>rd, rs1, rs2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Right Immediate</td>
<td>SRLI</td>
<td>rd, rs1, rs2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Right Arithmetic</td>
<td>SRAI</td>
<td>rd, rs1, rs2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Shifts</strong></td>
<td>Shift Left</td>
<td>R</td>
<td>SL</td>
<td>rd, rs1, rs2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Right</td>
<td>R</td>
<td>SRL</td>
<td>rd, rs1, rs2</td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic</strong></td>
<td>ADD Immediate</td>
<td>I</td>
<td>ADDI</td>
<td>rd, rs1, imm</td>
<td>ADD(W)</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td>I</td>
<td>ADDI</td>
<td>rd, rs1, imm</td>
<td>ADD(W)</td>
</tr>
<tr>
<td></td>
<td>Subtract</td>
<td>I</td>
<td>SUB</td>
<td>rd, rs1, imm</td>
<td>SUB(W)</td>
</tr>
<tr>
<td><strong>Load Upper Imm</strong></td>
<td>ULI</td>
<td>ULI</td>
<td>rd, imm</td>
<td>ULI</td>
<td>rd, imm</td>
</tr>
<tr>
<td><strong>Logical</strong></td>
<td>XOR Immediate</td>
<td>I</td>
<td>XORI</td>
<td>rd, rs1, imm</td>
<td>XORI</td>
</tr>
<tr>
<td></td>
<td>OR</td>
<td>I</td>
<td>ORI</td>
<td>rd, rs1, imm</td>
<td>ORI</td>
</tr>
<tr>
<td></td>
<td>OR Immediate</td>
<td>I</td>
<td>ORI</td>
<td>rd, rs1, imm</td>
<td>ORI</td>
</tr>
<tr>
<td></td>
<td>AND</td>
<td>I</td>
<td>ANDI</td>
<td>rd, rs1, imm</td>
<td>ANDI</td>
</tr>
<tr>
<td></td>
<td>AND Immediate</td>
<td>I</td>
<td>ANDI</td>
<td>rd, rs1, imm</td>
<td>ANDI</td>
</tr>
<tr>
<td><strong>Compare</strong></td>
<td>Set &lt;</td>
<td>I</td>
<td>SLLT</td>
<td>rd, rs1, rs2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set &lt; Unaligned</td>
<td>I</td>
<td>SLLTU</td>
<td>rd, rs1, rs2</td>
<td></td>
</tr>
<tr>
<td><strong>Branches</strong></td>
<td>Branch #1</td>
<td>S</td>
<td>BEQ</td>
<td>rs1, rs2, imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Branch #2</td>
<td>S</td>
<td>BNE</td>
<td>rs1, rs2, imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Branch #3</td>
<td>S</td>
<td>BLT</td>
<td>rs1, rs2, imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Branch #4</td>
<td>S</td>
<td>BGE</td>
<td>rs1, rs2, imm</td>
<td></td>
</tr>
<tr>
<td><strong>Jump &amp; Link</strong></td>
<td>JAL</td>
<td>U</td>
<td>JAL</td>
<td>rd, imm</td>
<td></td>
</tr>
<tr>
<td><strong>Synch</strong></td>
<td>Sync Thread</td>
<td>I</td>
<td>FENCE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sync Intr &amp; Data</td>
<td>I</td>
<td>FENCE 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>System</strong></td>
<td>System CALL</td>
<td>I</td>
<td>SUB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### RV Privileged Instructions

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>RV mnemonic</th>
<th>Access</th>
<th>RV Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CSR Access</strong></td>
<td>Atomic R/W</td>
<td>CSR98N</td>
<td>rd, csr, rl1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Atomic Read &amp; Set Bit</td>
<td>CR95N</td>
<td>rd, csr, rl1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Atomic Read &amp; Clear Bit</td>
<td>CR95C</td>
<td>rd, csr, rl1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Atomic R/W Imm</td>
<td>CR95W</td>
<td>rd, csr, imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Atomic Read &amp; Set Bit Imm</td>
<td>CR95MI</td>
<td>rd, csr, imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Atomic Read &amp; Clear Bit Imm</td>
<td>CR95CI</td>
<td>rd, csr, imm</td>
<td></td>
</tr>
<tr>
<td><strong>Change Level</strong></td>
<td>Env. Call</td>
<td>SCALL</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Shifts</strong></td>
<td>Shift Left</td>
<td>R</td>
<td>SL</td>
<td>rd, rs1, rs2</td>
</tr>
<tr>
<td></td>
<td>Shift Right</td>
<td>R</td>
<td>SRL</td>
<td>rd, rs1, rs2</td>
</tr>
<tr>
<td><strong>Arithmetic</strong></td>
<td>ADD Immediate</td>
<td>I</td>
<td>ADDI</td>
<td>rd, rs1, imm</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td>I</td>
<td>ADDI</td>
<td>rd, rs1, imm</td>
</tr>
<tr>
<td></td>
<td>Subtract</td>
<td>I</td>
<td>SUB</td>
<td>rd, rs1, imm</td>
</tr>
</tbody>
</table>

#### Optional Compressed (16-bit) Instruction Extension: RVC

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Fmt</th>
<th>RVC</th>
<th>RV Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Loads</strong></td>
<td>Load Word</td>
<td>L</td>
<td>LW</td>
<td>rd, rl1, imm</td>
</tr>
<tr>
<td></td>
<td>Load Word SP</td>
<td>L</td>
<td>LW</td>
<td>rd, sp, imm*4</td>
</tr>
<tr>
<td></td>
<td>Load Double</td>
<td>L</td>
<td>LD</td>
<td>rd, rl1, imm*4</td>
</tr>
<tr>
<td></td>
<td>Load Double SP</td>
<td>L</td>
<td>LD</td>
<td>rd, sp, imm*4</td>
</tr>
<tr>
<td></td>
<td>Load Quad</td>
<td>L</td>
<td>LQ</td>
<td>rd, rl1, imm*8</td>
</tr>
<tr>
<td></td>
<td>Load Quad SP</td>
<td>L</td>
<td>LQ</td>
<td>rd, sp, imm*4</td>
</tr>
<tr>
<td><strong>Stores</strong></td>
<td>Store Word</td>
<td>S</td>
<td>SW</td>
<td>rs1, rs2, imm*4</td>
</tr>
<tr>
<td></td>
<td>Store Word SP</td>
<td>S</td>
<td>SW</td>
<td>sp, rs2, imm*4</td>
</tr>
<tr>
<td></td>
<td>Store Double</td>
<td>S</td>
<td>SD</td>
<td>rs1, rs2, imm*8</td>
</tr>
<tr>
<td></td>
<td>Store Double SP</td>
<td>S</td>
<td>SD</td>
<td>sp, rs2, imm*4</td>
</tr>
<tr>
<td><strong>Arithmetic</strong></td>
<td>ADD</td>
<td>I</td>
<td>ADDI</td>
<td>rd, rs1, imm</td>
</tr>
<tr>
<td></td>
<td>ADD Immediate</td>
<td>I</td>
<td>ADDI</td>
<td>rd, rs1, imm</td>
</tr>
<tr>
<td></td>
<td>ADD Word</td>
<td>I</td>
<td>ADDI</td>
<td>rd, rs1, imm</td>
</tr>
<tr>
<td></td>
<td>ADD Imm</td>
<td>I</td>
<td>ADDI</td>
<td>rd, rs1, imm</td>
</tr>
<tr>
<td></td>
<td>ADD SP Imm*4</td>
<td>I</td>
<td>ADDI</td>
<td>sp, imm*4</td>
</tr>
<tr>
<td><strong>Logical</strong></td>
<td>XOR</td>
<td>I</td>
<td>XORI</td>
<td>rd, rs1, imm</td>
</tr>
<tr>
<td></td>
<td>OR</td>
<td>I</td>
<td>ORI</td>
<td>rd, rs1, imm</td>
</tr>
<tr>
<td></td>
<td>AND</td>
<td>I</td>
<td>ANDI</td>
<td>rd, rs1, imm</td>
</tr>
<tr>
<td></td>
<td>AND Immediate</td>
<td>I</td>
<td>ANDI</td>
<td>rd, rs1, imm</td>
</tr>
<tr>
<td><strong>Compare</strong></td>
<td>Set &lt;</td>
<td>I</td>
<td>SLLT</td>
<td>rd, rs1, rs2</td>
</tr>
<tr>
<td></td>
<td>Set &lt; Unaligned</td>
<td>I</td>
<td>SLLTU</td>
<td>rd, rs1, rs2</td>
</tr>
<tr>
<td><strong>Branches</strong></td>
<td>Branch #1</td>
<td>S</td>
<td>BEQ</td>
<td>rs1, rs2, imm</td>
</tr>
<tr>
<td></td>
<td>Branch #2</td>
<td>S</td>
<td>BNE</td>
<td>rs1, rs2, imm</td>
</tr>
<tr>
<td></td>
<td>Branch #3</td>
<td>S</td>
<td>BLT</td>
<td>rs1, rs2, imm</td>
</tr>
<tr>
<td></td>
<td>Branch #4</td>
<td>S</td>
<td>BGE</td>
<td>rs1, rs2, imm</td>
</tr>
<tr>
<td><strong>Jump &amp; Link</strong></td>
<td>JAL</td>
<td>U</td>
<td>JAL</td>
<td>rd, imm</td>
</tr>
</tbody>
</table>

---

### Proposed V Extension State

**Standard RISC-V scalar x and f registers**

|-----|-----|--------|--------|

- **Vector configuration**: CSR
  - vcf

- **Vector length**: CSR
  - vlr

- **Vector is maximum vector length, implementation and configuration dependent, but MVL > 4**

- **8 vector predicate registers, with 1 bit per element**

---

**RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RV1 base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RV1 instruction. See risc.org.**
RISC-V Vector Extensions

• Instructions are defined for vector:
  – Data Movement
    • Move/load/store
  – Arithmetic
    • Add/Sub
    • Mul/Madd/Msub/Div
    • Min/Max/Clip
    • Sqrt/Dot prod
  – Logical ops
    • Shift/Rotate
    • And/Or/Xor/Merge
  – Branching
    • Gr/Le/Eq

• https://github.com/riscv/riscv-v-spec
RISC for Graphics

• We propose a new set of **graphics** instructions designed for 3d graphics and media processing

• *These new instructions build on the base vector instruction set. We add support for new data types that are graphics specific.*

• We will denote this as the “X” extension so it doesn’t interfere with other RISC-V nomenclature.

• Advantage is this is a **fused CPU-GPU ISA**

• We call this **RV32X**
RV32X

• Motivation and Goals:
  – We want small, area-efficient designs
  – Custom programmability and extensibility
  – Low cost of IP ownership and development
  – Does not compete with commercial offerings
  – FPGA and ASIC targets
  – Free and open-source
  – Targeted to low-power microcontrollers
  – Strive to be DirectX (Shader Model 5) and OpenGL/ES and Vulkan compliant as we progress development
RV32X ISA

- Instructions will be 32-bits long (compliant with RISC-V ISA)
- Programming model is SIMD with some scalar operations for special functions
- Hardware will be a graphics vector accelerator attached to a host RISC-V core
- Instruction decode will happen partially on the host and partially in the accelerator using a special bit field in the instruction
- Architecture will support FPGA and ASIC designs as with the base RISC-V ISA
  - 16-bit fixed point (ideal for FPGAs)
  - 32-bit floating point (ASICs or FPGAs)
- Architecture defines a set of:
  - Hardwired base graphics and media instructions
  - User-Configurable RAM based micro-coded instructions for a run-time application-defined ISA extension
RV32X Data Types

- Scalars (8, 16, 24 and 32 bit fixed and floats)
  - Transcendentals (sincos, atan, pow, exp, log, rcp, rsq, sqrt etc.)
- Vectors (RV32-V)
  - We will support 2-4 element (8, 16 or 32 bits/element) vector operations along with specialized instructions for a general 3d graphics rendering pipeline
  - Points, Pixels, Texels (essentially “special” vectors)
    - XYZW points (64 and 128 bit fixed and floats)
    - RGBA pixels (8, 16, 24 and 32 bit pixels)
    - UVW texels (8, 16 bits per component)
    - Lights and Materials (Ia, ka, Id, kd, ls, ks…)
- Matrices
  - 2x2, 3x3 and 4x4 matrices will be supported as a native data type along with memory structures to support them
  - Attribute Vectors (XYZWRGBAUWHNxNyNz…)
    - Essentially represented in a 4x4 matrix
RV32X Register Set

• 136-bit “configurable” vector registers
  – Splits: 4x32b, 8x16b or 16x8b
  – 8-bit Configuration bits define data types:
    • Pixels (RGBA)
    • Points (XYZW)
    • Vectors (2,3,4 components)
    • Matrix stacks (16 component across registers)
    • Lighting, Viewing, Projection Parameters
    • Texture Coordinates
    • Pixel and Frame-buffer Operations (ROPs, bitblt)
    • Attribute Arrays and Differentials
    • Scalar constants and variables (i.e. math functions)

• Any other user defined types
  – Have 256 possible data types that can be application defined
  – For example AOS or SOA formats (ArrayOfStructure or StructureOfArray)

• Register Files will support random access by register name or using a push-pop FIFO like functionality.
RV32X Vector/Math Instructions

• Vector/Matrix Processing: 2,3,4 components
  – SetVec/SetMat
  – Push/Pop (vec/mat)
  – MatAddSub/MatMul
  – VecMat/ScalarVec
  – Dot/Cross
  – Dist/Len
  – Trans/Inv/Det/Norm
  – Swz (swizzle components, bits)
  – Lerp/Slerp

• Transcendental Math (scalar)
  – Sincos, atan, exp, pow, log, rcp, rsq, sqrt, cordic
  – Min/Max/Rnd/Floor/Ceil/Lerp/Slerp
RV32X Pixel/Texture Instructions

- **Pixel Instructions**
  - SetPix/ClrPix/GetPix
  - Blend
  - Ztest
  - ROP

- **Texture Instructions**
  - Tex2d, Tex3d
  - TexEnv
  - TexGen
  - MipMap
  - Persp
  - TexLoad
  - TexCodec
RV32X Frame Buffer Instructions

- Frame Buffer Instructions
  - SetZ/ClrZ
  - SetArea/ClerArea
  - Sync/Scanout
  - Compress/Decomp
  - BitBlt
  - Improc
  - ConfigBuffer

- Frame Buffer itself can be configured:
  - Pixel Buffer, Geometry Buffer, Texture Buffer, A-Buffer, etc.
RV32X Graphics Instructions

- Optional Graphics Instructions (micro-coded)
  - ModelView
  - Backface
  - Lookat
  - Proj
  - Clip2/Clip3
  - Lit (a,d,s)
  - Persp
  - InterpStep (i.e. Bresenham DDA or scanline attributes)
  - Window
  - TexMap
  - Z-Test
  - AlphaBlend
  - FragMerge

- MicroCode Instructions
  - LoadMicroCode (instruction to load custom micro-instructions into ucode RAM)
  - ClearMicroCode
Advantages of Fused CPU-GPU ISA

• Can implement a standard graphics pipeline in microcode
• Support for Custom Shaders
• Can implement Ray-Tracing extensions
• Vector support for Numerical Simulations
• 8-bit integer data types for AI/Machine Learning
• Can implement Custom rasterizers
  – Splines
  – SubDiv Surfaces
  – Patches
• Can implement Custom pipeline “stages”
  – Custom geometry/pixel/frame buffer stages
  – Custom tessellators
  – Custom instancing operations
RV32X Reference Implementation

- Instruction/Data SRAM Cache (32KB)
- Microcode SRAM (8KB)
- Dual Function Instruction Decoder
  - Hardwired implementing RV32V and X
  - Micro-coded Instruction Decoder for custom ISA
- Quad Vector ALU (32 bits/ALU – fixed/float)
- 136-bit Register Files (1K elements)
- Special Function Unit
- Texture Unit
- Configurable “local” Frame Buffer
RV32X Hardware

- Level-1 Cache (32KB)
- MicroCoded Controller and Hardwired Instruction Decode and Execute Logic
- uCode SRAM (8K)
- 32-bit DSP
- 32-bit DSP
- 32-bit DSP
- 32-bit DSP
- Splittable 136-bit Graphics Register Files (1024 elements) x 4
- Frame Buffer
- Special Function Unit
- Texture Unit
Scalable Design

Stand-Alone Low-End Graphics Microcontroller

or

Larger RISC-V CPU

Used as “shaders” in a multicore design
Novel Ideas

- Fused unified CPU-GPU ISA

- Configurable registers for custom data types

- User-defined SRAM based micro-code for application defined custom hardware extensions
  - Custom rasterizer stages
  - Ray tracing
  - Machine Learning
  - Computer Vision

- Same design serves both as a stand-alone graphics microcontroller or scalable shader unit

- Data formats support FPGA-native or ASIC implementations
Key Points and Next Steps

• This is a very early spec in development, subject to change based on stakeholder and your input. We will have a discussion forum set up.

• Immediate goal is to build a sample implementation
  – Instruction Set Simulator
  – FPGA implementation using open-source IP
  – Custom IP designed as open-source project
  – Demos and Benchmarks

• If anyone interested in helping with this please reach out to one of us. Thank you!
Thank You! Questions?

Atif Zafar

Atif@Pixilica.com