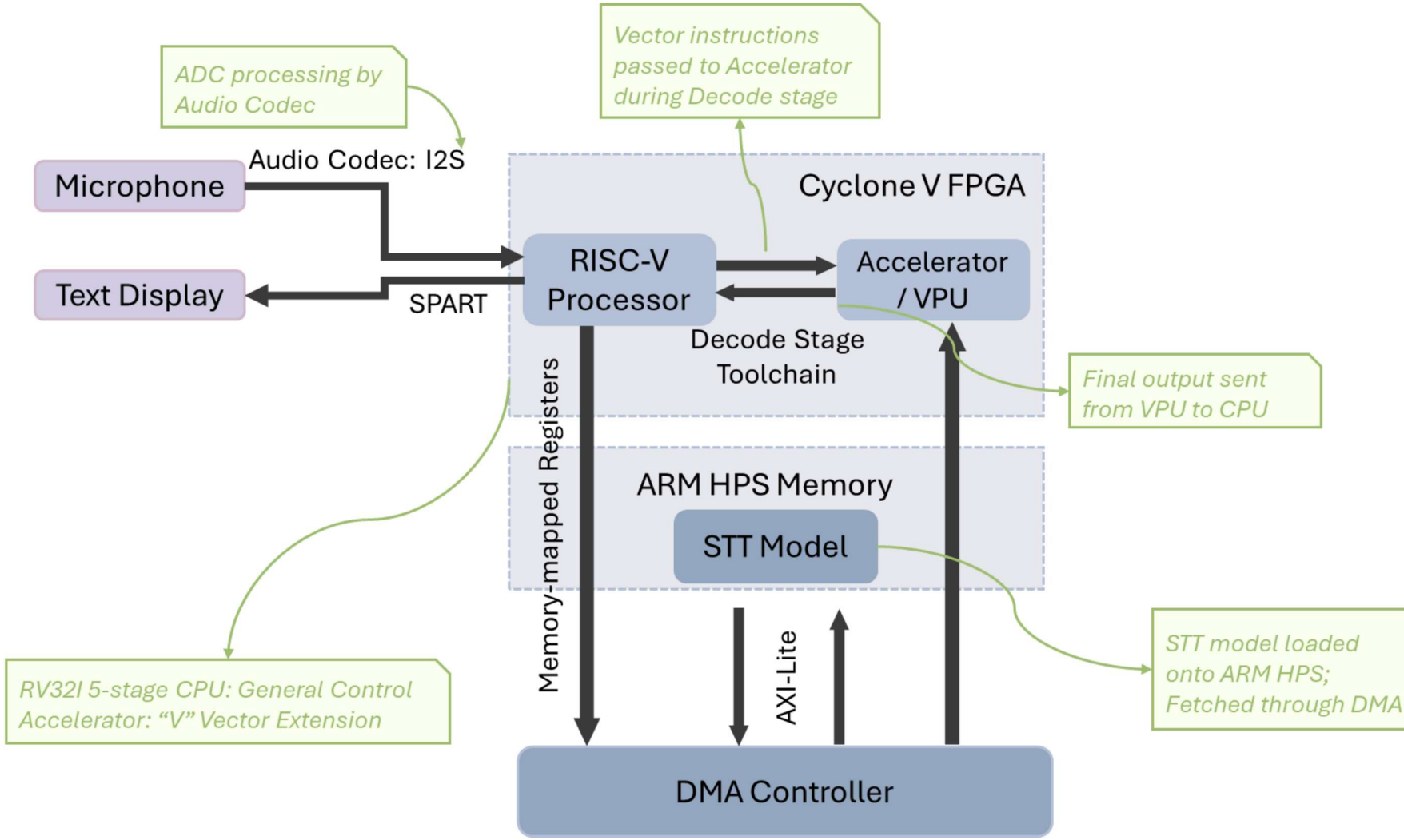
SpeakEasy: Architecture Review

Asish Das, Rohan Rao, Sadman Sakib, Aditi Shah
4th March 2025

High-Level Block Diagram

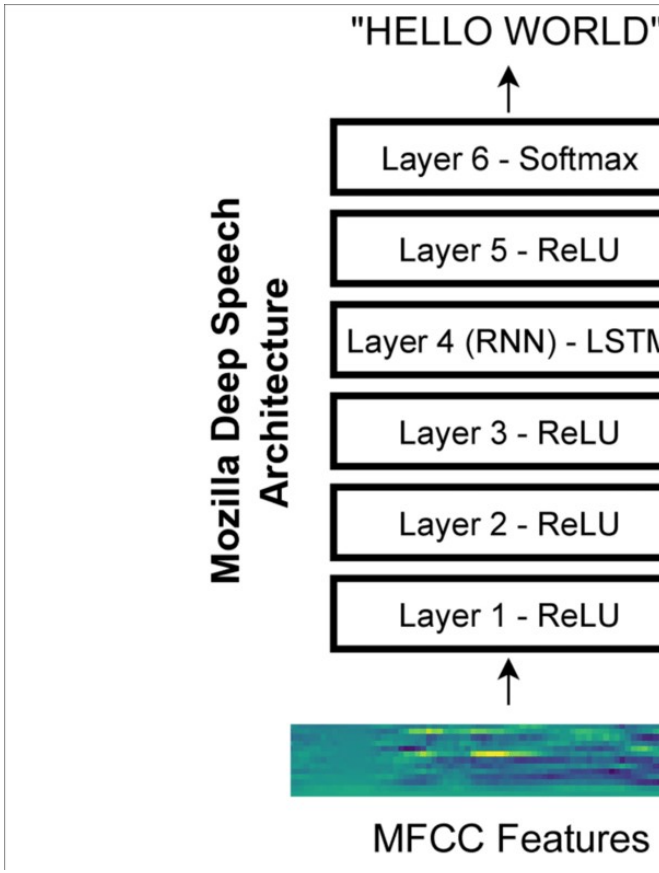


Flow of Process

(A) AUDIO INPUT
Microphone → Audio Codec (I^2S) → FIFO (BRAM)
↓
(B) AUDIO BUFFERING
FIFO → FPGA SDRAM (64MB)
↓
(C) FEATURE EXTRACTION (VPU)
FFT Engine → Filtering → FPGA SDRAM (Spectrogram)
↓
(D) STT MODEL INFERENCE (VPU)
Model data (Weight/biases) from HPS DDR3 → VPU
VPU Accelerator (LSTM, Matrix Multiplication) → FPGA SDRAM
↓
(E) CTC DECODING (RISC-V)
RISC-V CPU (Decoder) → Transcribed Text
↓
(F) SERIAL OUTPUT (SPART)
RISC-V CPU → SPART (UART) → Serial Monitor (PuTTY)

Algorithms: Speech-To-Text Model

- **Accelerating a Speech-To-Text (STT) Model:**
  - **DeepSpeech** versus **PicoVoice** versus **Vosk**
- Choose model based on size, simplicity of operations - loading on DDR3.
  - DeepSpeech – fully open source, complex (300MB-1GB)
  - PicoVoice – only certain models open source (50MB-200MB)
- RISC-V VPU Co-Processor can be very useful for acceleration of data intensive tasks leveraging SIMD
- STT Algorithm in **C code** with **RISC-V GNU ToolChain** using extensions.



Processor Implementation

Open-Source RV32I ISA

Features to Consider	PicoRV32	VexRiscv
Written in	Verilog (Easier with Extension)	SpinalHDL
ISA	RV32I	RV32I ++
Pipeline	Sequential - slower	5-stage
Size	Smaller	Medium
Linux support	No MMU	MMU available

RISC-V GNU Toolchain

- For compilation, assembling, linking and debugging
- Implies C-code for the model

Accelerator Implementation

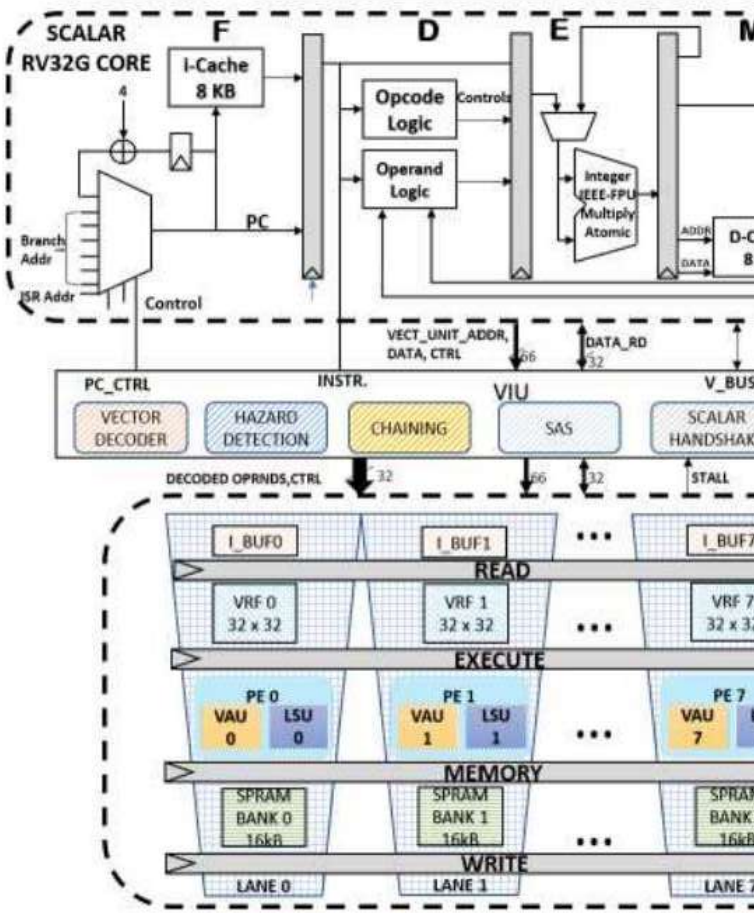
Vector Processing Unit:
RISC-V "V" Vector Extension ISA

Requirements:

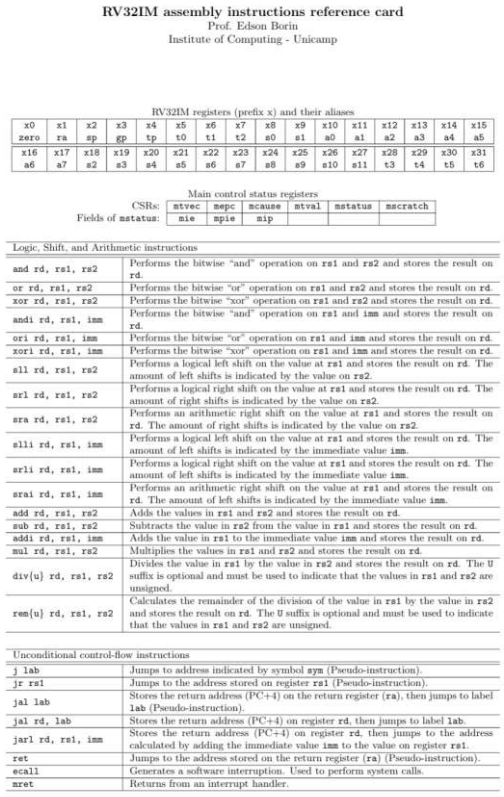
- For parallel data processing for LSTM, ReLU, Softmax, MAC, DSP
- Improves throughput by enabling SIMD operations

Details:

- Detects vector instructions from Decode Stage
- Schedules them into processor
- Has own Register File
- Details to follow in Microarchitectural Diagrams



Processor – Accelerator ISA



RV32IM assembly instructions reference card

Prof. Edson Borin

Institute of Computing - Unicamp

RV32IM registers (prefix x) and their aliases

x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14	x15
zero	ra	sp	gp	tp	t0	t1	t2	s0	s1	a0	a1	a2	a3	a4	a5
x16	x17	x18	x19	x20	x21	x22	x23	x24	x25	x26	x27	x28	x29	x30	x31
a6	a7	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	t3	t4	t5	t6

Main control status registers

CSRs:	mtvec	mepc	mcause	mtval	mstatus	mscratch
Fields of mstatus:	mie	mpie	mip			

Logic, Shift, and Arithmetic instructions

`and rd, rs1, rs2`	Performs the bitwise "and" operation on `rs1` and `rs2` and stores the result on `rd`.
`or rd, rs1, rs2`	Performs the bitwise "or" operation on `rs1` and `rs2` and stores the result on `rd`.
`xor rd, rs1, rs2`	Performs the bitwise "xor" operation on `rs1` and `rs2` and stores the result on `rd`.
`andi rd, rs1, imm`	Performs the bitwise "and" operation on `rs1` and `imm` and stores the result on `rd`.
`ori rd, rs1, imm`	Performs the bitwise "or" operation on `rs1` and `imm` and stores the result on `rd`.
`xori rd, rs1, imm`	Performs the bitwise "xor" operation on `rs1` and `imm` and stores the result on `rd`.
`sll rd, rs1, rs2`	Performs a logical left shift on the value at `rs1` and stores the result on `rd`. The amount of left shifts is indicated by the value on `rs2`.
`srl rd, rs1, rs2`	Performs a logical right shift on the value at `rs1` and stores the result on `rd`. The amount of right shifts is indicated by the value on `rs2`.
`sra rd, rs1, rs2`	Performs an arithmetic right shift on the value at `rs1` and stores the result on `rd`. The amount of right shifts is indicated by the value on `rs2`.
`slli rd, rs1, imm`	Performs a logical left shift on the value at `rs1` and stores the result on `rd`. The amount of left shifts is indicated by the immediate value `imm`.
`srli rd, rs1, imm`	Performs a logical right shift on the value at `rs1` and stores the result on `rd`. The amount of left shifts is indicated by the immediate value `imm`.
`srai rd, rs1, imm`	Performs an arithmetic right shift on the value at `rs1` and stores the result on `rd`. The amount of left shifts is indicated by the immediate value `imm`.
`add rd, rs1, rs2`	Adds the values in `rs1` and `rs2` and stores the result on `rd`.
`sub rd, rs1, rs2`	Subtracts the value in `rs2` from the value in `rs1` and stores the result on `rd`.
`addi rd, rs1, imm`	Adds the value in `rs1` to the immediate value `imm` and stores the result on `rd`.
`mul rd, rs1, rs2`	Multiplies the values in `rs1` and `rs2` and stores the result on `rd`.
`div{u} rd, rs1, rs2`	Divides the value in `rs1` by the value in `rs2` and stores the result on `rd`. The `U` suffix is optional and must be used to indicate that the values in `rs1` and `rs2` are unsigned.
`rem{u} rd, rs1, rs2`	Calculates the remainder of the division of the value in `rs1` by the value in `rs2` and stores the result on `rd`. The `U` suffix is optional and must be used to indicate that the values in `rs1` and `rs2` are unsigned.

Unconditional control-flow instructions

`j lab`	Jumps to address indicated by symbol `sym` (Pseudo-instruction).
`jr rs1`	Jumps to the address stored on register `rs1` (Pseudo-instruction).
`jal lab`	Stores the return address (PC+4) on the return register (`ra`), then jumps to label `lab` (Pseudo-instruction).
`jal rd, lab`	Stores the return address (PC+4) on register `rd`, then jumps to label `lab`.
`jarl rd, rs1, imm`	Stores the return address (PC+4) on register `rd`, then jumps to the address calculated by adding the immediate value `imm` to the value on register `rs1`.
`ret`	Jumps to the address stored on the return register (`ra`) (Pseudo-instruction).
`ecall`	Generates a software interruption. Used to perform system calls.
`mret`	Returns from an interrupt handler.

Conditional set and control-flow instructions

`slt rd, rs1, rs2`	Sets `rd` with 1 if the signed value in `rs1` is less than the signed value in `rs2`, otherwise, sets it with 0.
`slti rd, rs1, imm`	Sets `rd` with 1 if the signed value in `rs1` is less than the sign-extended immediate value `imm`, otherwise, sets it with 0.
`sltu rd, rs1, rs2`	Sets `rd` with 1 if the unsigned value in `rs1` is less than the unsigned value in `rs2`, otherwise, sets it with 0.
`sltui rd, rs1, imm`	Sets `rd` with 1 if the unsigned value in `rs1` is less than the unsigned immediate value `imm`, otherwise, sets it with 0.
`seqz rd, rs1`	Sets `rd` with 1 if the value in `rs1` is equal to zero, otherwise, sets it with 0 (Pseudo-instruction).
`snez rd, rs1`	Sets `rd` with 1 if the value in `rs1` is not equal to zero, otherwise, sets it with 0 (Pseudo-instruction).
`sltz rd, rs1`	Sets `rd` with 1 if the signed value in `rs1` is less than zero, otherwise, sets it with 0 (Pseudo-instruction).
`sgtz rd, rs1`	Sets `rd` with 1 if the signed value in `rs1` is greater than zero, otherwise, sets it with 0 (Pseudo-instruction).
`beq rs1, rs2, lab`	Jumps to label `lab` if the value in `rs1` is equal to the value in `rs2`.
`bne rs1, rs2, lab`	Jumps to label `lab` if the value in `rs1` is different from the value in `rs2`.
`beqz rs1, lab`	Jumps to label `lab` if the value in `rs1` is equal to zero (Pseudo-instruction).
`bnez rs1, lab`	Jumps to label `lab` if the value in `rs1` is not equal to zero (Pseudo-instruction).
`blt rs1, rs2, lab`	Jumps to label `lab` if the signed value in `rs1` is smaller than the signed value in `rs2`.
`bltu rs1, rs2, lab`	Jumps to label `lab` if the unsigned value in `rs1` is smaller than the unsigned value in `rs2`.
`bge rs1, rs2, lab`	Jumps to label `lab` if the signed value in `rs1` is greater or equal to the signed value in `rs2`.
`bgeu rs1, rs2, lab`	Jumps to label `lab` if the unsigned value in `rs1` is greater or equal to the unsigned value in `rs2`.

Data movement instructions

`mv rd, rs`	Copies the value from register `rs` into register `rd` (Pseudo-instruction).
`li rd, imm`	Loads the immediate value `imm` into register `rd` (Pseudo-instruction).
`la rd, rot`	Loads the label address `rot` into register `rd` (Pseudo-instruction).
`lw rd, imm(rs1)`	Loads a 32-bit `signed` or `unsigned word` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`lh rd, imm(rs1)`	Loads a 16-bit `signed halfword` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`lhu rd, imm(rs1)`	Loads a 16-bit `unsigned halfword` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`lb rd, imm(rs1)`	Loads a 8-bit `signed byte` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`lbu rd, imm(rs1)`	Loads a 8-bit `unsigned byte` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`sw rs1, imm(rs2)`	Stores the 32-bit value at register `rs1` into memory. The memory address is calculated by adding the immediate value `imm` to the value in `rs2`.
`sh rs1, imm(rs2)`	Stores the 16 least significant bits from register `rs1` into memory. The memory address is calculated by adding the immediate value `imm` to the value in `rs2`.
`sb rs1, imm(rs2)`	Stores the 8 least significant bits from register `rs1` into memory. The memory address is calculated by adding the immediate value `imm` to the value in `rs2`.
`L{W|H|HU|B|BU} rd, lab`	For each one of the `lw`, `lh`, `lhu`, `lb`, and `lbu` machine instructions there is a pseudo-instruction that performs the same operation, but the memory address is calculated based on a label (`lab`) (Pseudo-instruction).
`S{W|H|B} rd, lab`	For each one of the `sw`, `sh`, and `sb` machine instructions there is a pseudo-instruction that performs the same operation, but the memory address is calculated based on a label (`lab`) (Pseudo-instruction).

Control and Status Read and Write instructions

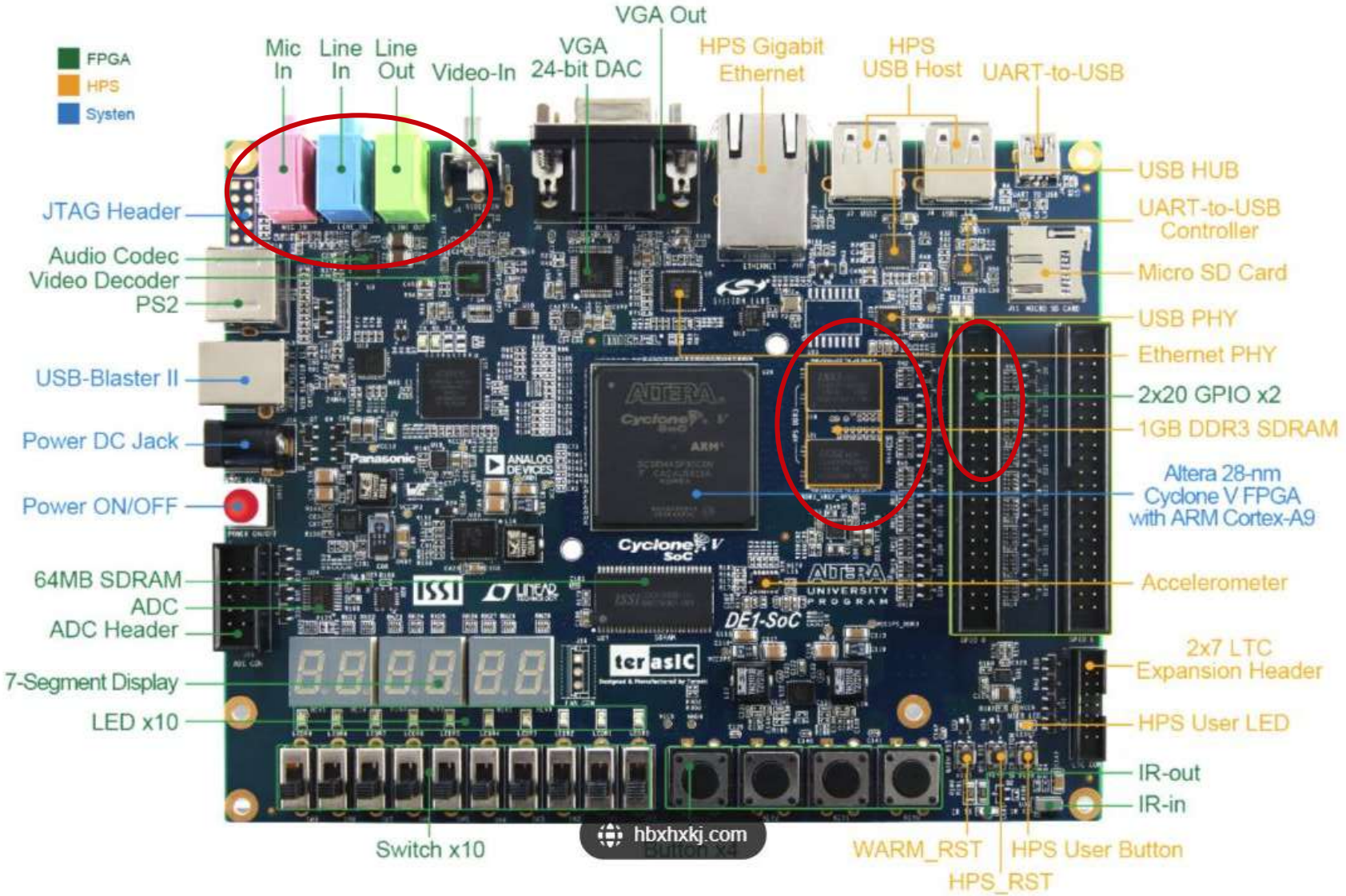
`csrr rd, csr`	Copies the value from the control and status register `csr` into register `rd` (Pseudo-instruction).
`csrw csr, rs`	Copies the value from register `rs` into the control and status register `csr` (Pseudo-instruction).
`csrrw rd, csr, rs1`	Copies the value from the control and status register `csr` into register `rd` and the value from the `rs1` register to the control and status register `csr`. If `rd=rs1`, the instruction performs an atomic swap between registers `csr` and `rs1`.
`csrc csr, rs`	Clears control and status register (`csr`) bits using the contents of the `rs` register as a bit mask. (Pseudo-instruction).
`csrs csr, rs`	Sets control and status register (`csr`) bits using the contents of the `rs` register as a bit mask. (Pseudo-instruction).

RV32I Processor

Category	Instr
Vector Setup	vsetvli
Vector Load/Store	vld, vlb, vlh,
	vst, vseb, vse
Vector Arithmetic	vadd, vsub, v
	vfmul, vfadd,
Vector Comparisons	vseq, vslt, vsg
	vmin, vmax
Vector Reductions	vredsum, vre vredmin
Vector Permutations	vperm
Vector Scalar Operations	vadd.vi, vsub
Vector Fused Operations	vfmacc, vfnm
Vector Shifting	vshl, vshr, va
Vector Masking	vmsk

Co-Processor

External Interfaces



External Interfaces

- **Audio CODEC**:
  - 🔊 **I2S bus** captures and transmits audio data.
  - 🎤 Connects to microphone.

- **Memory**:
  - 💾 **AXI** accesses DDR3 memory from DMA controller
  - DDR3 is used to store model weights and audio buffers.
  - On-chip RAM for intermediate data.

- **GPIO Pins**:
  - 📡 **SPART interface** communicates serially to interface with PuTTY.
  - 🖥️ Used for debugging and text output.

Verification of Original Plan

- Transitioned from preprocessed stored data to AXI-based real-time streaming for direct, low-latency data transfer from peripherals.
- No processing on HPS core, only using DDR3 memory
- Multi-stage VPU with RF instead of simple SIMD accelerator
- Using the RISC-V V Extension

Updated Gantt Chart

554 Team3

Asish Das, Aditi Shah, Rohan Rao, M Sadman Sakib

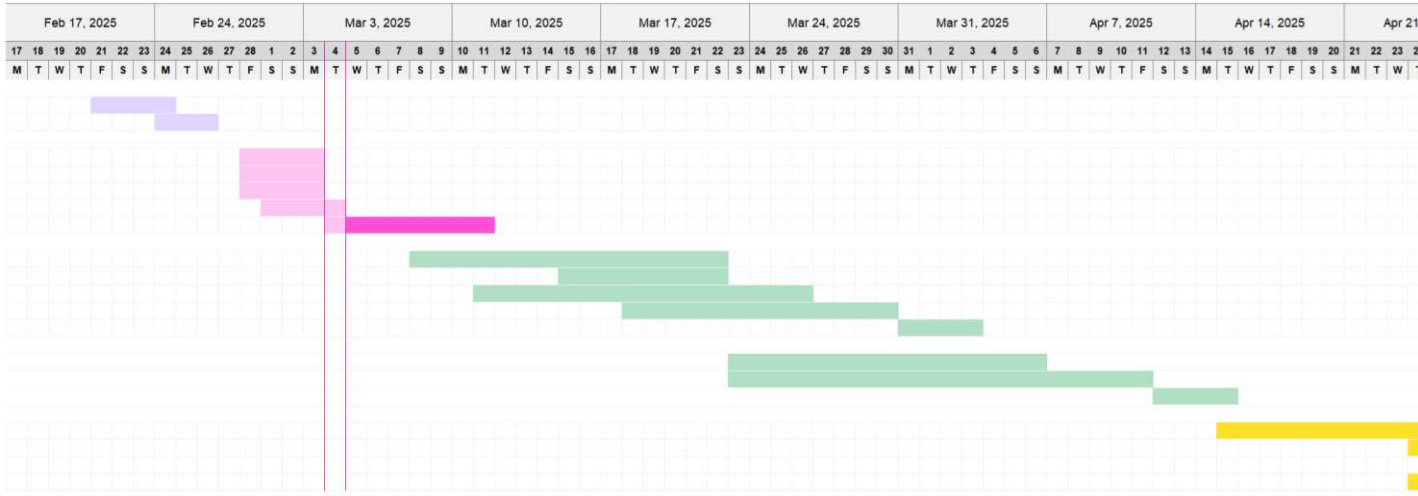
SpeakEasy

Project start: Fri, 2-21-2025

Display week: 1

TASK	ASSIGNED TO	PROGRESS	START	END
Initiation				
Project Discussion	AD AS RR SS	100%	2-21-25	2-24-25
Lightning Talk Planning	AD AS RR SS	100%	2-24-25	2-26-25
Architeture and Microarchitecture				
Block Diagram	AS SS	100%	2-28-25	3-3-25
Processor + Co-processor	AS RR	100%	2-28-25	3-3-25
Interfaces	AD RR	100%	2-28-25	3-3-25
Integration	AD AS RR SS	100%	3-1-25	3-4-25
Microarchitecture	AD AS RR SS	20%	3-4-25	3-11-25
Processor and Co-processor				
Processor Implementation	AD AS RR SS	0%	3-8-25	3-22-25
Processor Testing	AD AS RR SS	0%	3-15-25	3-22-25
Accelerator Implementation	AD AS RR SS	0%	3-11-25	3-26-25
Accelerator Testing	AD AS RR SS	0%	3-18-25	3-30-25
Slack	AD AS RR SS	0%	3-31-25	4-3-25
Model and Memory				
Model	AD AS RR SS	0%	3-23-25	4-6-25
AXI Interfacing	AD AS RR SS	0%	3-23-25	4-11-25
Slack	AD AS RR SS	0%	4-12-25	4-15-25
Presentation				
Poster Design	AD AS RR SS	0%	4-15-25	4-25-25
Internal Demo + Review	AD AS RR SS	0%	4-24-25	4-25-25
Poster Presentation	AD AS RR SS	0%	4-30-25	4-30-25
Final Report	AD AS RR SS	0%	4-24-25	5-4-25

Insert new rows ABOVE this one



Thank you! Any Questions?