High Performance Embedded Systems EEE4120F

FINAL EXAM MEMO!

19 June 2025

Venue: LESLIE SOCIAL: LS2B Time starting: 12h30 3 hours

3 hours MEMO!!!

Examination Prepared by: Simon Winberg
s
Last Modified: 19-May-2025

EEE4120F 2025 Final Exam Marking Memo

Section 1: Short Answers (2 x 10 marks = 20 marks)

Note: the student is expected to answer any **TWO** of the three questions given.

Question 1.1: Memory Hierarchy in HPES

High-performance embedded systems often utilize a complex memory hierarchy to achieve performance targets. Discuss the different levels typically found in a modern embedded system's memory hierarchy (e.g., caches, main memory, non-volatile storage) and explain how each level contributes to overall system performance. Include in your discussion the concepts of locality of reference (temporal and spatial) and how it is exploited by the memory hierarchy. [10 marks]

Marking Allocation:

- Identifying and briefly describing typical levels (e.g., Registers, Cache (L1, L2), Main Memory (RAM), Secondary/Non-volatile Storage (Flash, SSD)): 2-3 marks. This was covered much in term 2, the students should hopefully remember this well.
- Explaining the role and characteristics of each level (speed, size, cost per bit): 3-4 marks. Well, the student needs to just mention some problems of inefficient memory use that can be cause e.g. if data scattered in memory and not reused thus leading to much cached data being wasted.
- Explaining how the hierarchy improves performance (bridging speed gap between CPU and main memory, reducing average access time): 2 marks

- Explaining Temporal Locality and how the hierarchy exploits it (keeping recently used data/instructions in faster levels): 1-2 marks
- Explaining Spatial Locality and how the hierarchy exploits it (bringing blocks of data/instructions into faster levels): 1-2 marks

Sample Solution:
A modern high-performance embedded system typically employs a memory hierarchy to manage the trade-off between memory speed, size, and cost. The levels, from fastest/smallest to slowest/largest, include:

1. **Registers:** Smallest, fastest memory directly within the CPU, holding currently used data and instructions. Access is typically within one clock cycle.
2. **Cache Memory (L1, L2, etc.):** Small, fast SRAM placed between the CPU and main memory. L1 is typically split for data and instructions, L2 is larger and unified. Caches hold copies of frequently accessed data and instructions from main memory. Access time is a few clock cycles. They bridge the speed gap between the CPU and DRAM.
3. **Main Memory (DRAM):** Larger, slower, and less expensive than cache. Holds the majority of the active program code and data. Access time is tens to hundreds of clock cycles.
4. **Non-Volatile Storage (Flash, SSD, SD Card, etc.):** Largest, slowest, and cheapest per bit. Used for long-term storage of program code, operating system, and data that persists when the system is powered off. Access time is orders of magnitude slower than RAM.

(Ok, this maybe reads like a textbook answer in the most classic of ways, but that's totally OK; it's fine if the student gives a shorter and more creative answer which is nevertheless logical and relevant).
Main point is that the memory hierarchy improves overall system performance by exploiting **locality of reference**. Further benefits and explanations that the student might provide:

- **Temporal Locality:** If a data item or instruction is accessed, it is likely to be accessed again soon. The memory hierarchy exploits this by keeping recently accessed items in the faster memory levels (caches).
- **Spatial Locality:** If a data item or instruction is accessed, items with nearby memory addresses are likely to be accessed soon. The hierarchy exploits this by fetching blocks (cache lines) of data/instructions from slower levels into faster ones whenever an access occurs, anticipating future nearby accesses.

i.e. these points are more thing that I would expect, not just because I walk so much about the importance of temporal and spatial locality especially in terms of specialized HPES design… there isn't bonus marks awarded but this would certainly be well worthy of high marks if the student shows this sort of discussion that is very relevant to the course and the problem-based learning done. Main point for temporal, is keeping frequently used data and instructions in faster memory levels, the hierarchy minimizes the number of slow accesses to main memory or non-volatile storage, significantly reducing the average memory access time and improving CPU performance.

Question 1.2: Interrupts vs. Polling for I/O
Compare and contrast the use of interrupts and polling for handling I/O events in an embedded system. Discuss the advantages and disadvantages of each approach in the context of high-performance and real-time requirements. Provide scenarios where each method would be preferred (for example, you might want to refer to the Raspberry Pi part of Section 3 for ideas, where instead of bit-banging you decide on using a timer interrupt to send data to the ToneSoC being controlled).

Marking Allocation:

- Defining Polling (CPU repeatedly checks device status): 2 marks
- Defining Interrupts (Device signals CPU when attention is needed): 2 marks
- Advantages of Polling (Simpler implementation for simple cases, deterministic timing in simple loops, no complex interrupt handling): 2 marks
- Disadvantages of Polling (Wastes CPU cycles checking status, can miss events if not frequent enough, difficult to manage multiple devices): 2 marks
- Advantages of Interrupts (Efficient use of CPU time - only active when needed, responsive to asynchronous events, easier to manage multiple devices): 2 marks
- Disadvantages of Interrupts (More complex to implement, requires saving/restoring context, can introduce latency variability, potential for race conditions/re-entrancy issues): 2 marks
- Discussing suitability for HPES/Real-time (Interrupts generally better for responsiveness/efficiency, but interrupt latency and determinism are critical; Polling only suitable for slow, non-critical, or very simple/predictable I/O in HPES): 2 marks
- Providing appropriate scenarios (Polling: simple, low-speed, single device, e.g., checking a button press in a simple loop. Interrupts: High-speed devices, multiple devices, real-time events, e.g., network packet arrival, timer events, peripheral data ready): 2 marks

Sample Solution:
Polling involves the CPU repeatedly checking the status of an I/O device to see if it requires service. The CPU actively waits for the device.

Interrupts are hardware signals generated by an I/O device to the CPU when it requires attention or has completed an operation. The CPU's current task is suspended, an Interrupt Service Routine (ISR) is executed to handle the device, and then the CPU returns to the suspended task.

Comparison (HPES/Real-time Context):

- **CPU Utilization:** Polling is inefficient as the CPU spends time checking status even when the device doesn't need service. This wastes valuable processing time in HPES. Interrupts are efficient; the CPU is only diverted when an event occurs.
- **Responsiveness:** Polling's responsiveness depends on how frequently the device is checked. If checks are infrequent, events can be missed or delayed.

Interrupts provide faster response to asynchronous events, as the device signals immediately. This is crucial for meeting real-time deadlines.

- **Complexity:** Polling is simpler to implement for a single, slow device. Managing multiple devices with different timing needs via polling becomes complex. Interrupts require more complex setup (interrupt controllers, vectors, context saving/restoring) and careful handling of shared data and re-entrancy in ISRs, which is a challenge in complex HPES.
- **Determinism:** In simple, tightly controlled loops, polling can offer deterministic timing for the check itself. However, the *response* time depends on the polling frequency. Interrupt response time involves interrupt latency (time to acknowledge and start ISR) and execution time, which can be variable depending on other interrupts or system load, making strict real-time guarantees challenging without careful analysis and prioritization.

Preferred Scenarios:

- **Polling** is typically only suitable in HPES for very simple, non-time-critical, or infrequent I/O operations where the overhead of interrupts is not warranted, or in very simple bare-metal systems where deterministic checking within a simple loop is required. Example: Checking a configuration DIP switch status during initialization.
- **Interrupts** are preferred for most I/O in HPES, especially for high-speed peripherals (e.g., network interfaces, complex sensors, storage controllers) and in real-time systems where timely responses to external events are critical. They allow the CPU to perform other tasks while waiting for I/O, improving overall system throughput and responsiveness. Example: Handling data reception from a high-speed serial port or reacting to a critical sensor alarm within a guaranteed deadline.

Question 1.3: Parallel Program Design & Granularity [10 marks]
List the eight steps described in the lectures for the “steps in designing parallel programs” [3 marks]. Explain briefly why, even in the first step ("Understanding the Problem"), when you’re planning to develop a new version of an existing system, you still want to remember the saying “don’t lose sight of the forest for the trees” [4 marks]. Considering the types of benchmarking one might do on a system, explain why keeping this saying in mind is important during the start of a project. Finally, briefly explain what the term granularity means in the context of computing [3 marks].

Marking Allocation:

- Listing the eight steps in designing parallel programs (allow for slight variations in wording depending on lecture content, but core concepts should be present): 3 marks (e.g., 3 all there in right sequence, 2 most there but sequence possibly a bit wrong, 1 missing quite a few but some attempt)
    - Step 1: Understanding the Problem / Partitioning
    - Step 2: Partitioning / Decomposition
    - Step 3: Communication
    - Step 4: Synchronization

    - Step 5: Mapping / Assignment / Task Scheduling
    - Step 6: Performance Tuning / Optimisation
    - Step 7: Verification / Debugging
    - Step 8: Evaluation / Analysis *(Note: The first step is explicitly given. Accept any reasonable list of 8 steps covering decomposition, communication, mapping, tuning, etc., if the exact list wasn't strictly numbered 1-8 in lectures, or if a different standard was used. The key is that they list a set of 8 distinct steps from your course.)*
- Explaining "don't lose sight of the forest for the trees": Understanding it means focusing on the big picture/overall system goals (forest) rather than getting bogged down in minor details or individual components (trees). 1 mark
- Explaining why this is important in the "Understanding the Problem" step for a *new version* of an *existing* system: It's easy to focus only on improving the existing parts ("trees") without questioning if the overall approach or architecture ("forest") is still the best solution for the problem, or if fundamental changes are needed. A holistic view is needed. 2 marks
- Explaining why keeping this in mind is important during initial benchmarking: Benchmarking should align with overall system goals and identify significant bottlenecks ("forest"). Focusing too early on micro-benchmarks of trivial components ("trees") that don't significantly impact overall performance is a waste of effort and can be misleading. 1 mark
- Defining Granularity in computing: The size of code/tasks that can be executed independently between communication or synchronization events. 3 marks (e.g., mention task size, relation to communication/synchronization, fine vs coarse grain).

Sample Solution:

The eight steps described for designing parallel programs are typically:

1. Understanding the Problem (or Partitioning/Decomposition)
2. Partitioning/Decomposition
3. Communication
4. Synchronization
5. Mapping/Assignment
6. Performance Tuning/Optimisation
7. Verification/Debugging
8. Evaluation/Analysis

Even in the first step, "Understanding the Problem", especially when creating a new version of an existing system, it's crucial to remember "don't lose sight of the forest for the trees". This saying means not getting so focused on the details of individual components or optimising existing code structures ("the trees") that you lose sight of the overall system requirements, goals, and potential alternative high-level designs ("the forest"). When developing a new version, you must consider if the original overall approach is still optimal or if a fundamentally different parallelisation strategy

or algorithm might be better suited to the problem, rather than just tweaking the existing "trees". This holistic view is essential for achieving significant performance gains.

Keeping this saying in mind during initial benchmarking is also important. Early benchmarking should focus on identifying the major performance bottlenecks and understanding the behaviour of the system at a high level ("the forest"). Benchmarking should target critical sections that impact overall performance. Getting lost in detailed micro-benchmarks of non-critical components ("the trees") too early can consume significant time and effort without yielding insights into the primary factors limiting the system's overall high performance.

In the context of computing, **granularity** refers to the size of the individual tasks or work units that a program is divided into for parallel execution. It describes the ratio of computation performed within a task to the amount of communication or synchronization required between tasks. **Fine-grained** parallelism involves many small tasks with frequent communication, while **coarse-grained** parallelism involves fewer large tasks with less frequent communication.

Section 2: Multiple Choice (10 x 5 marks = 50 marks)

Choose the **most appropriate** answer for each of the following questions.
Correct Answers:
2.1: (c) Dedicated function and often real-time constraints

2.2: (b) Provide high-speed storage for frequently accessed data and instructions

2.3: (c) Semaphores

2.4: (b) Increase the number of instructions executed per unit of time

2.5: (c) To allow peripherals to transfer data to or from memory without direct CPU intervention

2.6: (c) Detect and recover from system malfunctions (e.g., due to software errors) by resetting the system

2.7: (b) Simpler implementation in hardware and often faster execution

2.8: (b) Ensuring proper synchronization and avoiding race conditions between cores

2.9: (b) The ability of the system to complete tasks within a guaranteed time frame

2.10: (c) I2C (Inter-Integrated Circuit)

Section 3: ToneSoC Development Scenario (30 marks)

Question 3.1: Design and Implementation of build_msg (20 marks)

Q3.1a: Design (8 marks)

Provide pseudocode and a state machine diagram for the build_msg module.

Marking Allocation (Q3.1a):

- Pseudocode correctly outlining the steps: initializing, bit counting, receiving bit-by-bit on sclk edge, assembling packet, decoding MSG on completion, updating outputs: 3 marks
- State Machine Diagram structure: Clear states (e.g., IDLE, RECEIVING, DECODING/APPLYING): 2 marks
- State Transitions: Correct triggers based on sclk edges, reset, bit count: 2 marks
- Actions within states: Assembling packet, updating outputs (en, ntone, vol) based on MSG: 1 mark *(Note: Marks can be awarded for either pseudocode or diagram demonstrating understanding of a step)*

Sample Solution (Q3.1a):

Pseudocode for build_msg:

Marking comment: yes, excessively detailed response, students are not expected to provide such a complehensive near real solution, they just need to mention some of the main things like responding to reset, checking posedge of sclk, etc.

```
Code snippet
Initialize state = IDLE
Initialize received_packet = 0 (12 bits)
Initialize bit_count = 0

On reset signal high:
    state = IDLE
    received_packet = 0
    bit_count = 0
    en = 0
    ntone = 0
    vol = 0

On positive edge of sclk:
    If state is IDLE:
        // Start receiving a new packet
        state = RECEIVING
        bit_count = 0
        received_packet = 0 // Clear previous packet data

    If state is RECEIVING:
        // Shift in the new bit from din
        received_packet = (received_packet << 1) | din

        // Increment bit counter
        bit_count = bit_count + 1

        // Check if 12 bits have been received
        If bit_count == 12:
```

```
            state = DECODING_APPLYING
            // Packet is now fully in received_packet
```

On state transition to DECODING_APPLYING (occurs on the sclk posedge that receives the 12th bit):

```
    // Extract MSG and DAT fields
    msg = received_packet[11..9]
    dat = received_packet[8..0]

    // Decode MSG and update outputs accordingly
    Case msg:
        000 (Nop):
            // Do nothing, outputs retain previous values or default
            // state remains DECODING_APPLYING briefly to finish, then back to IDLE
            // Or, transition to IDLE immediately after decoding
            state = IDLE // Ready for next packet
        001 (tone):
            en = 1        // Enable tone generation
            ntone = dat   // Set the new tone number
            // vol retains its previous value
            state = IDLE // Ready for next packet
        010 (volume):
            // en retains its previous value (tone might still be enabled)
            // ntone retains its previous value
            vol = dat     // Set the new volume level
            state = IDLE // Ready for next packet
        011 (fadeout):
            // This command requires internal logic (not part of build_msg outputs directly)
            // A signal indicating "fadeout initiated" and the fadeout time (dat)
            // would need to be passed to another module (e.g., the volume control logic,
            // potentially part of or interacting with tone_gen).
            // For build_msg's direct outputs: en/ntone/vol state might depend on fadeout logic.
            // Simple approach for this Q: build_msg just decodes, assumes other logic handles fade.
            // Alternative: Pass 'fadeout_en' and 'fadeout_time' outputs.
            // Let's assume for this question, build_msg's direct role is just to provide *inputs* to
tone_gen based on other commands. Fadeout is handled elsewhere triggered by this MSG.
            // So, maybe just decode and ignore for this module's outputs?
            // Or, signal that a fadeout command was received:
            // output reg fadeout_cmd_received;
            // output reg [8:0] fadeout_duration;
            // fadeout_cmd_received = 1;
            // fadeout_duration = dat;
            // ... and reset these after 1 CLK cycle or upon next packet.
            // Given the prompt's focus on outputs for tone_gen (en, ntone, vol),
            // the simplest interpretation is that build_msg's main job is to set these.
            // Let's assume fadeout is just a "received event" build_msg could signal,
            // but doesn't directly change en/ntone/vol in a simple way here.
            // Let's stick to just handling 000, 001, 010 for updating en, ntone, vol.
            // Any other MSG could be ignored by build_msg, or keep previous outputs.
            state = IDLE // Ready for next packet
        Default (other MSG values):
            // Ignore or handle as Nop
            state = IDLE // Ready for next packet

    // Note: Outputs (en, ntone, vol) are registered and hold their values
    // until updated by a new command.
```

State Machine Diagram (Conceptual Description):
A very simple design could have only two states: *(this simple solution is adequate!!)*

- IDLE: Reset state. Transitions to RECEIVING.
- RECEIVING: On posedge sclk, capture din, shift into register, increment bit count. If bit count reaches 12, decode MSG/DAT and update outputs, then reset bit count and go back to receiving the *next* packet immediately. Reset always goes to IDLE. This is simpler but might miss the first bit if the packet starts without a preceding idle sclk state. A counter and shift register clocked by posedge sclk are key.

The state machine description to be more standard for serial reception:

- **States:**
  - IDLE: Waits for the start of a packet (implicitly, the first posedge sclk with valid data after a period of inactivity, or assumes packets are back-to-back). On reset, goes here.
  - RECEIVING: Active while receiving the 12 bits.
- **Transitions:**
  - IDLE -> RECEIVING: On posedge sclk. Action: Initialize bit_counter = 0, received_packet = 0. Shift in the first bit: received_packet = (received_packet << 1) | din. bit_counter = 1.
  - RECEIVING -> RECEIVING: On posedge sclk if bit_counter < 12. Action: Shift in the next bit: received_packet = (received_packet << 1) | din. bit_counter = bit_counter + 1.
  - RECEIVING -> IDLE: On posedge sclk if bit_counter == 11 (after receiving the 12th bit). Action: Shift in the last bit: received_packet = (received_packet << 1) | din. Decode received_packet, update en, ntone, vol based on MSG. bit_counter = 0 (ready for next packet).
- **Reset:** From any state to IDLE on reset high. Reset bit_counter, received_packet, en, ntone, vol.

This simpler state machine assumes packets arrive contiguously or build_msg is always processing bits. A more robust design might require an idle period detection or start bit, but the prompt doesn't mention this, so the simpler counter/shift register approach clocked by sclk is likely sufficient.

Q3.1b: Verilog Implementation (12 marks)
Implement the build_msg module in Verilog.
Marking Allocation (Q3.1b):

- Correct module interface: clk, sclk, din, reset inputs; en, ntone, vol outputs: 1 mark
- Shift register or equivalent logic to assemble the 12-bit packet: 3 marks
- Counter to track received bits (0 to 11): 2 marks
- Logic to update shift register and counter on the posedge sclk: 3 marks
- Logic to detect when 12 bits are received (e.g., counter reaches 12 or 11 depending on counting style): 1 mark
- Decoding the MSG field (bits 11:9) of the received packet: 1 mark
- Logic to update en, ntone, vol based on the decoded MSG (specifically for 001 and 010): 3 marks

- Reset logic correctly implemented (asynchronous reset for state, counters, outputs): 2 marks
- Code clarity, comments, and general Verilog correctness: 1 mark *(Total marks might exceed 12 for partial marks. Adjust as needed.)*

Sample Code Solution (Q3.1b):

```
localparam [0:0] RECEIVING = 1'b0, DECODE = 1'b1; // Simple states driven by bit_count

always @(posedge sclk or posedge reset) begin
    if (reset) begin
        bit_count <= 4'd0;
        received_packet <= 12'd0;
        en <= 1'b0;
        ntone <= 9'd0;
        vol <= 9'd0;
    end else begin
        received_packet <= {din, received_packet[11:1]}; // Shift right, MSB first

        if (bit_count == 11) begin // This is the 12th bit
            bit_count <= 4'd0; // Reset for next packet
            // Packet {din, received_packet[11:1]} is now the full packet
            // Decode and apply command immediately

            reg [11:0] current_packet = {din, received_packet[11:1]};
            reg [2:0] msg = current_packet[11:9];
            reg [8:0] dat = current_packet[8:0];

            case (msg)
                3'b001: begin // tone
                    en <= 1'b1;
                    ntone <= dat;
                end
                3'b010: begin // volume
                    vol <= dat;
                end
                3'b000: begin // Nop
                    // No change
                end
                3'b011: begin // fadeout
                    // No change to en/ntone/vol by build_msg
                end
                default: begin
                    // No change
                end
            endcase

        end else begin // Not the 12th bit yet
            bit_count <= bit_count + 1;
        end
    end
end
/
 // This statemachine is cleaner and directly implements the logic
 // Shift in bit, if 12th bit, decode and reset count. Otherwise, increment count.
```

```
    always @(posedge sclk or posedge reset) begin
        if (reset) begin
            bit_count <= 4'd0;
            received_packet <= 12'd0;
            // Outputs reset to a silent state
            en <= 1'b0;
            ntone <= 9'd0;
            vol <= 9'd0;
        end else begin
            // Shift the new bit into the MSB position (assuming MSB first)
            received_packet <= {din, received_packet[11:1]};

            if (bit_count == 4'd11) begin // This is the 12th bit
                bit_count <= 4'd0; // Reset counter for the next packet

                // The complete 12-bit packet is now in received_packet *after* this clock edge.
                // We can decode the packet value captured on this edge.
                reg [11:0] complete_packet_this_cycle = {din, received_packet[11:1]}; // Capture the
value *after* shift
                reg [2:0] msg = complete_packet_this_cycle[11:9];
                reg [8:0] dat = complete_packet_this_cycle[8:0];

                // Decode the MSG and update outputs
                case (msg)
                    3'b001: begin // tone
                        en <= 1'b1;
                        ntone <= dat;
                        // vol retains previous value
                    end
                    3'b010: begin // volume
                        // en retains previous value
                        // ntone retains previous value
                        vol <= dat;
                    end
                    3'b000: begin // Nop
                        // Outputs remain unchanged
                    end
                    3'b011: begin // fadeout - build_msg doesn't directly change en/ntone/vol for this
                        // Assume outputs remain unchanged; fadeout logic elsewhere
                    end
                    default: begin // Other/Invalid MSG
                        // Outputs remain unchanged (or revert to a safe default)
                    end
                endcase

            end else begin // Not the 12th bit yet
                bit_count <= bit_count + 1;
            end
        end
    end

endmodule
```

Q3.2: Tone Generation Design (10 marks)
Explain how a tone could be produced by the tone_gen module.
Marking Allocation (Q3.2):

- Understanding the goal: Produce an output waveform at a frequency determined by ntone using the 100KHz clk. Volume controlled by vol. 2 marks
- Concept of generating a periodic waveform digitally (e.g., using a counter and lookup table, or an algorithm like Direct Digital Synthesis (DDS) - simple counter/accumulator is sufficient for explanation): 3 marks
- Explaining how the 100KHz clk is used as the sampling rate/update rate for the DAC output: 1 mark
- Explaining how ntone determines the tone frequency (e.g., controlling the step size of a counter/accumulator that indexes a waveform lookup table): 3 marks
- Explaining how vol controls the amplitude of the output waveform (e.g., scaling the values from the lookup table before sending to DAC): 2 marks
- Mentioning the DAC's role in converting digital values to analog: 1 mark *(Allow for different valid approaches like simple square waves, sine waves via LUT, etc. Focus on the core concepts of frequency and amplitude control in the digital domain.)*

Sample Solution (Q3.2):
The tone_gen module needs to produce a digital representation of an audio waveform (like a sine wave or square wave) at a specific frequency and amplitude, updating its output (aout conceptually, although the module outputs digital values ntone and vol which control the *generation* parameters) at the 100KHz clk rate. This digital value would then be sent to the 'cheap and nasty' DAC to produce the analog output.
The 100KHz clk serves as the **sampling clock** or update rate for the digital waveform generation. This means the digital value sent to the DAC can change at most 100,000 times per second.
To generate a tone at a frequency of ntone * 100 Hz, we can use a method like **Direct Digital Synthesis (DDS)** or a simpler counter-based approach with a **lookup table**. A common approach involves:

1. **A Phase Accumulator:** A counter (e.g., 16-bit or more) that increments on each positive edge of the 100KHz clk.
2. **A Frequency Control Word:** The ntone input can be mapped to a "step size" for the phase accumulator. A higher ntone (and thus higher target frequency) corresponds to a larger step size. This means the accumulator will wrap around more often. The frequency of the output waveform is proportional to this step size and the clock frequency, scaled by the accumulator's bit-width (F_out=fracStepSize2AccumulatorWidthtimesF_clk). We need to map ntone (0-511) to a step size such that ntone=1 gives 100Hz, ntone=500 gives 50KHz. The step size would be proportional to ntone.
3. **A Waveform Lookup Table (LUT):** The most significant bits of the phase accumulator's output are used as an index into a Read-Only Memory (ROM) or array storing one cycle of a desired waveform (e.g., a sine wave). This lookup table provides the digital amplitude value of the waveform at that point in its cycle. The resolution of the accumulator bits used for indexing determines the number of samples per waveform cycle, affecting output quality.

4. **Volume Control:** The vol input (0-511) is used to scale the output from the lookup table. This is typically done by multiplying the waveform sample value by the volume value and then potentially shifting or dividing to fit the DAC's input range. A volume of 0 would result in a constant zero output, and 511 would apply maximum scaling.
5. **DAC Output:** The final, scaled digital sample value is output to the DAC at the 100KHz clk rate.

For example, to generate a 100Hz tone (ntone=1) at 100KHz clock rate: We need 1000 samples per cycle (100textKHz/100textHz=1000). A phase accumulator would need to step by 2AccumulatorWidth/1000 on each clock cycle. The value of ntone would scale this step size.
In summary, the tone_gen module uses the 100KHz clk to clock a phase accumulator. The ntone input controls the step size of this accumulator, determining how quickly it cycles through the waveform. The accumulator's output indexes a lookup table containing the waveform shape. The vol input scales the amplitude of the value read from the lookup table. This scaled digital value is the output, driving the DAC.

end of solutions and marking memo