Embedded System Design and Modeling

EE382N.23, Fall 2017

Lab #2

Refinement

Part (a) due: ~~November 1, 2017~~ November 6, 2017(11:59pm)
Part (b) due: ~~November 8, 2017~~ November 13, 2017 (11:59pm)

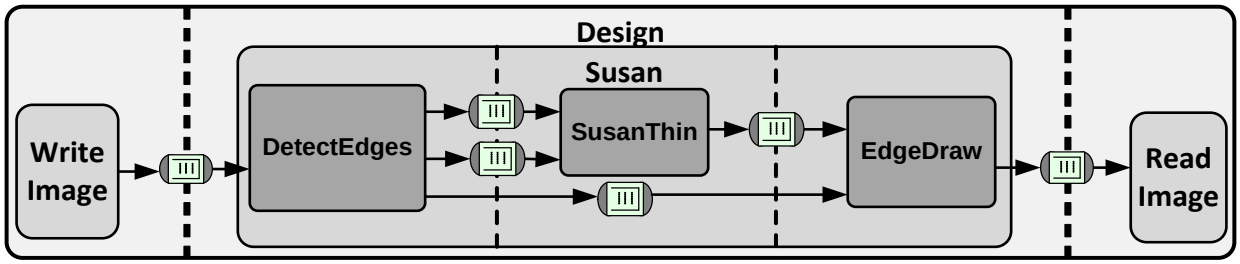
Instructions:

- Please submit your solutions via Blackboard. Submissions should include a lab report (single PDF) and a single Zip or Tar archive with the source and supplementary files (code should include a README and has to be compilable and simulatable by running 'make' and 'make test', respectively).
- You are allowed to work in teams of up to three people and you are free to switch partners between labs and the project. Please submit one solution per team.

SUSAN Edge Detector Model Refinement

The purpose of this lab is to perform computation and communication refinement on the SUSAN edge detector SpecC specification model. As discussed in the lectures, the purpose of refinement process is to compile the abstract specification model into detailed computation and communication models to represent corresponding design decisions both for performance analysis as well as for further synthesis and implementation. You can start the refinement from the specification model developed in Lab 1 (or the reference solution).

A diagram of the reference specification model is shown as below:

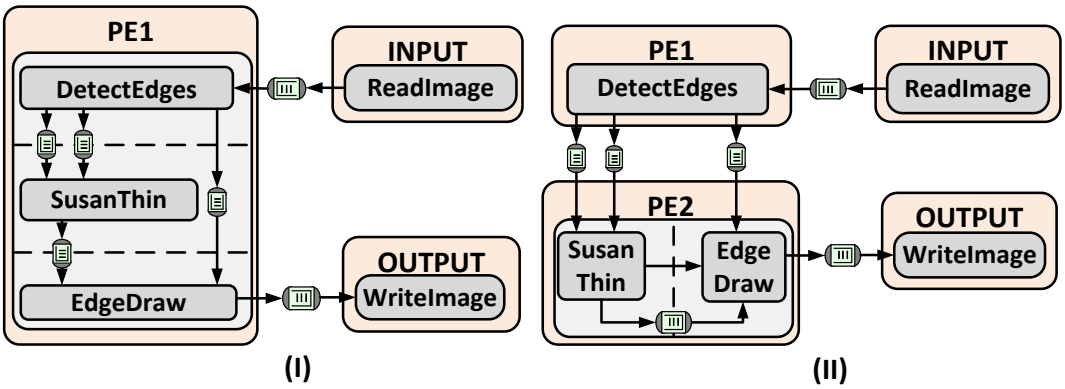


In this lab, we will refine this *Design* behavior all the way down to both transaction-level and pin-accurate communication models:

(a) First, manually refine the SUSAN edge detector specification model into a computation model where the top *Design* behavior reflects the partitioning and scheduling of all behaviors among available processing elements (PEs):

1. Before a computation refinement process, normally we need to perform a design space exploration and allocate PEs and partition the behaviors, e.g. based on some initial profiling. For this lab, we will assume a given mapping of behaviors to PEs, including associated performance/profiling metrics.

   Assume that we have allocated an *ARM* processor (PE1) and a *HW* accelerator (PE2). In partition (I), all behaviors are allocated on PE1, while in partition (II), *SusanThin* and *EdgeDraw* are allocated on PE2 to accelerate the overall execution. For *ReadImage* and *WriteImage*, two dedicated virtual PEs, INPUT and OUTPUT, are allocated, respectively, and during your refinement you can ignore their execution durations:



For PE1 and PE2, the following table shows performance profiling metrics in the granularity of one loop iteration. Note that in *SusanEdges* and *EdgeDraw* there are two consecutive loops, which are referred to as *A* and *B* in the code and the following table:

	ARM (PE1)	*HW (PE2)*
SetupBrightnessLut	*2700*	*360*
SusanEdges_A	*19000000*	*3000000*
SusanEdges_B	*20000000*	*1200000*
SusanThin	*6400000*	*180000*
EdgeDraw_A	*12000000*	*600000*
EdgeDraw_B	*12000000*	*600000*

Implement wrapper behaviors for PEs and insert `waitfor()` statements with corresponding times in the correct loops for behavior *DetectEdges*, *SusanThin* and *EdgeDraw*. Simulate both partitions (I) and (II), and include the results in your report.

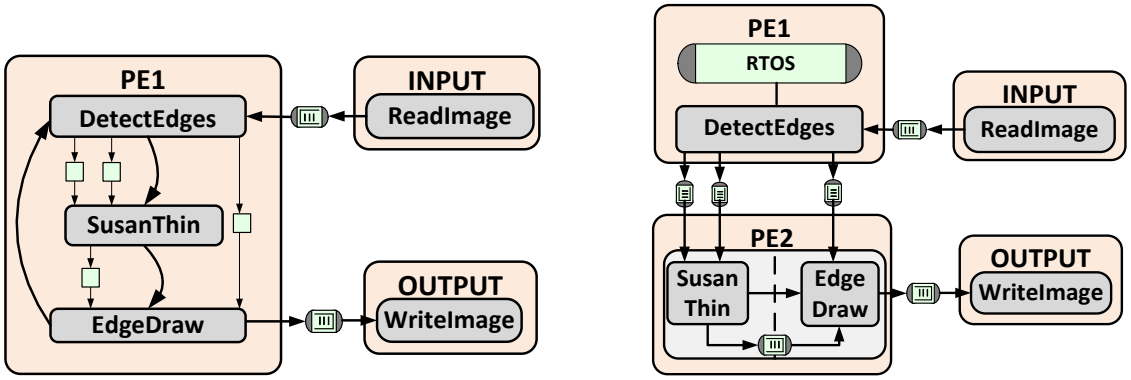
2. Assume that PE1 is a single-core ARM processor, which can only execute one thread at a time. In the specification model, we have parallelized the outer loops in each leaf behavior and multiple loop iterations can execute simultaneously. In this lab, we need to statically or dynamically serialize and schedule all parallel threads. In case of dynamic scheduling, an OS model is needed to emulate the online interleaving of threads in time.

   For partition (I), perform static scheduling. Serialize the three behaviors (*DetectEdges*, *SusanThin* and *EdgeDraw*) on PE1 and statically schedule them in an FSM loop. Also statically serialize all the parallelized loop threads in each of these three behaviors.

   For partition (II), assume dynamic scheduling under an operating system that uses preemptive multitasking. Introduce an OS channel in PE1 and register each loop thread in behavior *DetectEdges* in the OS model. Apply a round-robin scheduling strategy to make the execution switch between parallel threads. Refer to slides 19-21 in Lecture 8 and references [19] and [20] on the class website for more details about OS modeling.

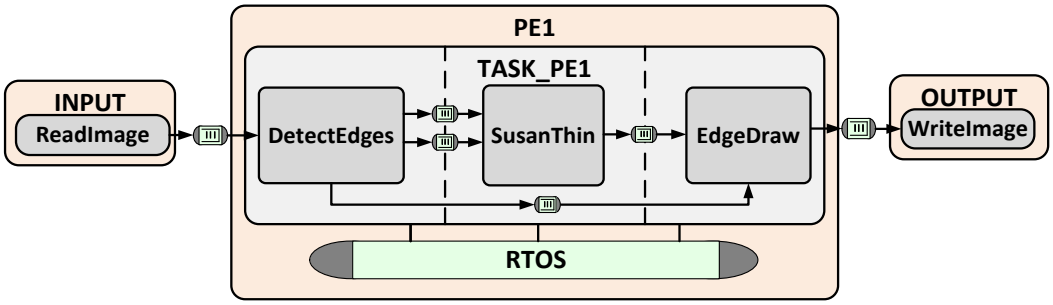
   Specifically, convert parallel behaviors into tasks that provide a method to register themselves with the OS and that wait for OS activation/terminate themselves at the beginning/end of their `main()`. Next, convert every `waitfor()` statement in each task into an `os_timewait()` call that models execution time and then yields the task by calling the OS-internal scheduler to allow for preemptive scheduling and context switching to another task. Every `par` statement in the original specification should be refined by wrapping it into a pair of `par_start()` / `par_end()` statements through which the OS model can dynamically fork/join tasks.

Diagrams of the resulting models for both partitions are shown below:



3. Finally, introduce an OS model for PE1 in partition (I). Keep the three top behaviors running in parallel under control of the OS in a wrapper task *TASK_PE1*, which will be executed as soon as the system starts. In case of preemptive scheduling fully under OS control, you also need to perform synchronization refinement for PE-internal communication channels to be provided on top of and integrated with the OS model. As described in the lectures, this can be achieved by adding a pair of `pre_wait()`/`post_wait()` OS methods around original SpecC wait statements inside the channels. You can find the original code for all SpecC standard channels under `$SPECC/import`. Create a copy of each used channel and refine the code as needed.

   A brief diagram of the resulting model is shown below:



4. Simulate and verify your timed computation models from part (2) and (3). What is the latency for processing of a single picture for each model? What is the throughput for each model? How much speedup can be achieved by using the hardware accelerator in PE2?

5. (extra credit) Implement an OS model that uses priority scheduling. How is latency and throughput of your design affected by different task priority settings?

(b) Next, perform communication refinement and transform the computation models from (a2) and (a3) into TLM and PAM models.

1. We will use the same simple bus protocol as in Homework 3 for all inter-PE communication. A detailed pin-accurate implementation of the *HWBus* protocol that is ready for integration and inlining is available at:

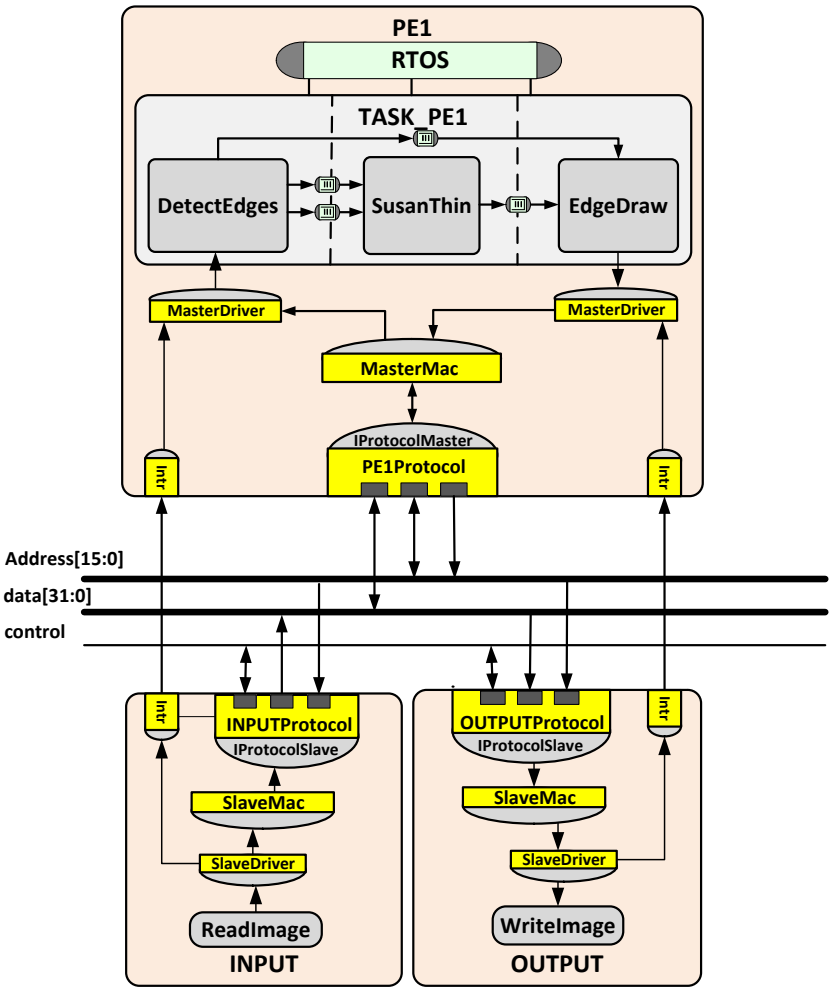
   `/home/projects/courses/fall_17/ee382n-23/HWBus.sc`

   Copy the code to your directory and browse the bus model to try to understand its structure. It is easiest to start with the channel *HardwareBus*, which shows a demo instantiation of the bus. It first defines physical layer realizations for interrupt detection

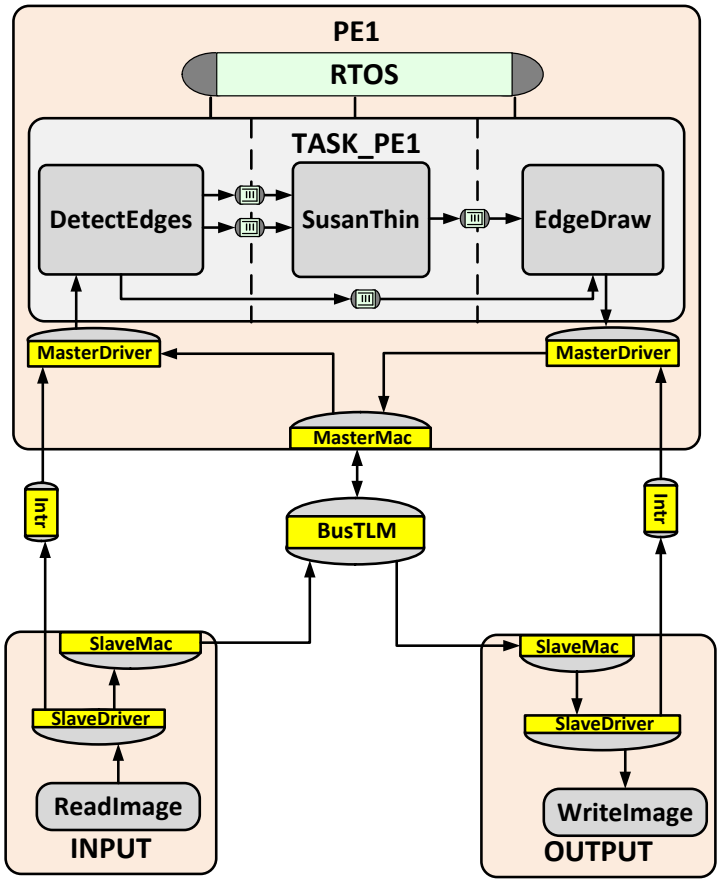
(*MasterHardwareSyncDetect*) and interrupt generation (*SlaveHardwareSyncGenerate*). The bus model then defines the bus wires and a protocol-level (physical) interface each for master (*MasterHardwareBus)* and slave (*SlaveHardwareBus*) sides, including how physical layers connect to bus wires. Finally, media access (MAC) channels (named (*Master|Slave)HardwareBusLinkAccess)* show the methods of how to access the bus.

2. Refine the partition (I) into a PAM, where PE1 is bus master and I/O blocks (INPUT, OUTPUT) are slaves. In a PAM, the pin-accurate physical and MAC layer bus protocols are inlined into each PE. Note that, to avoid deadlocks in case of communication with multiple bus slaves, each bus slave needs to send an interrupt signal to inform its corresponding master when it is ready. The PAM will need to include corresponding physical interrupt layer half channels inlined into each PE. All communications are happening through connected signal wires. Within each PE, there should be half-channels representing the drivers. The *DetectEdges* and *EdgeDraw* behaviors do not call the bus/interrupt channels directly. Rather, they should remain unrefined, and a driver adapter should translate the `send()`/`receive()` calls in the behaviors into bus communication, including all necessary synchronization. For the driver implementation, queue buffering semantics are not necessary and the original *c_typed_queue* are assumed to be refined down to non-buffered communication with double handshake semantics.

   A diagram of partition (I)'s resulting PAM is shown below:

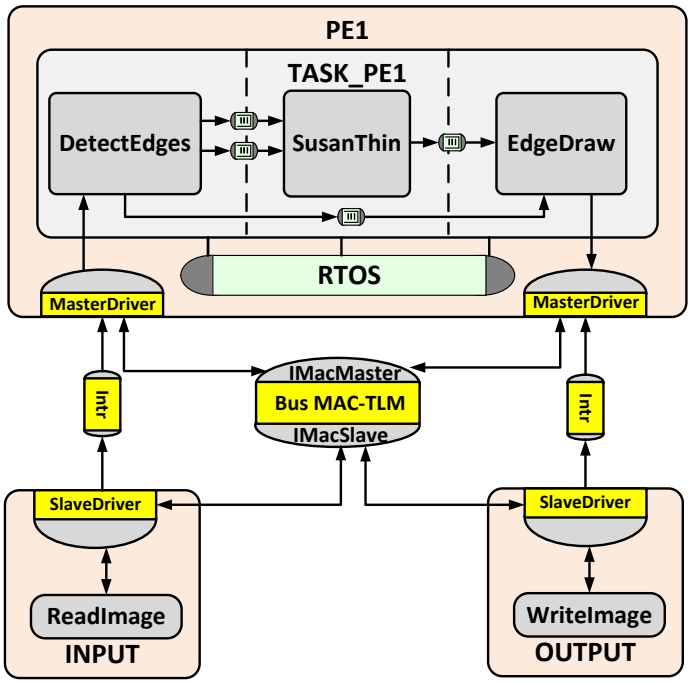


3. The protocol-level interface (both master and slave side) can be exchanged with a single *HardwareBusProtocolTLM* channel (where the communication is not performed via the wires previously instantiated, but through events as a transaction-level model). Implement a corresponding transaction-level channel model of the bus and refine the existing PAM into a TLM-based communication model. A TLM-based communication model has a similar structure. However, physical layers and wires are replaced with a TLM bus channel that will transfer the data word by word. Furthermore, TLM versions of interrupts are realized by replacing physical interrupt layers and wires with instances of standard *c_single_handshake* channels (that act as interrupt TLMs):



Does your TLM reach the same accuracy (in measured latencies) as the PAM? What simulation speedup does the TLM achieve compared to the PAM?

4. Finally, the MAC channels and TLM channels together can be exchanged by a single MAC-TLM channel (where both the MAC and physical layers are abstracted away and data will be directly communicated without being sliced into bus words), where in MAC-TLM, the image data is transferred all at once instead of transaction by transaction. Implement a MAC-TLM version of the bus and refine the communication model into a MAC-TLM based variant. A diagram of MAC-TLM communication model of partition (I) is shown below. In this case, a MAC-TLM channel encapsulates all the access and communication protocols. Note that interrupt models remain unchanged, and are still realized as instances of standard *c_handshake* channels:



Can your MAC-TLM reach the same accuracy (in measured latencies) as the PAM and/or TLM? In general, under what conditions will a MAC-TLM be able to provide the same simulated timing behavior as a TLM or PAM? What is the expected and actual speedup between PAM, TLM and MAC-TLM?

5. (extra credit) Implement PAM, TLM and MAC-TLM communication models for partition (II). Note that in this case, PE2 will be both master and slave on the bus. As such, proper bus arbitration between two masters will have to be implemented. You can assume a simple central arbiter component that receives requests and grants access to each PE. In the PAM, corresponding request and grant wires can be included and the central bus arbiter can be implemented as a separate behavior communicating with two masters through such wires. In the TLM and MAC-TLM, equivalent arbitration functionality needs to be included inside the bus channels. What about speed and accuracy of communication models in this case?

Document the transformation steps you applied and include listings of your modified source code. Throughout the refinement process, continuously validate your model after each change to make sure that it is still syntactically and functionally correct. Compile and simulate all models to validate their correctness. Explain the quantitative and qualitative composition of and contributions to the simulated delays observed in each model. Report on the differences in lines of code and simulation runtimes/speed between the models. To compute the lines of code for a SpecC model, you can use the `sir_stats` tool that is part of the SpecC tool set. Also, to obtain simulation runtimes, you can prepend the Unix `time` command in front of the simulation command line. Note, however, that you will have to increase the `time` resolution by averaging over a large number of simulation runs or a larger input test vector file.