Midterm Exam
CSE 510
CSU San Bernardino - Winter 2017
Aaron Chamberlain – ID: 003392908

1.1) An ASIP is an Application Specific Instruction-Set Processor. The name in of itself gives a very good description of the qualities that differentiate it from an ASIC. Typically an ASIP is designed with a very minimal Instruction Set Architecture (ISA) designed to fulfill the application itself with some room given for expansion or change [2]. This design approach of creating the chip around a custom ISA give flexibility and many options for the design. The design of an ASIP also typically involves co-development of the rest of the tool chain required for the final design. For example, a completely custom compiler may need to be written if the ISA varies far from a more commonly used ISA [3]. The ASIP design itself may vary as the high level programmers change the application they wish to run on it. For example, if the C programmer determines that the application algorithm chosen would benefit from pipe-lining or parallelism, then the ASIP designer may incorporate that into their choice of Instruction sets and Chip Architecture.

One of the most common applications for ASIPs are DSPs, or Digital Signal Processors. Speaking in general terms, a DSP is a chip that turns an Analog signal from a real world input such as Video, Audio, or temperature and converts into a compressed Digital Format stream in a well defined time constraint period. DSPs are applied across a wide range of the digital audio and video manipulation landscape, from performing FFTs (Fast Fourier Transforms) that derive signal levels and frequency spectrum analysis to all forms of filters. On the video side, DSPs can be used for simple tasks like currency counting where a value of currency can be defined by simple things like their color or reflectivity. Looking at a much more specific example, Texas Instrument (TI) has a wide range of DSPs that run Real Time Operating Systems (ROTS) in many aspects of the digital computation. The C6xxx series from TI for example, also integrates many connectivity standards so that the DSPs can be more readily integrated into full systems. These connectivity standards include: USB2, SATA, PCIe Gen2, Gigabit Ethernet, and Serial Ports [5].

1.2.1) The ARM Thumb ISA is a subset of the most commonly used 32-bit ARM instructions. Note that some of the instructions make use of the terms High and Low. In the Thumb ISA, the registers are split into two mode systems, where the programmer only has access to the ‘LOW’ registers and only certain instructions have access to the ‘HIGH’ registers for fast temporary storage [6]. The ‘LOW’ registers include: r0-r7, the PC (Program Counter), the SP (Stack Pointer), the LR (Link Register), and the CPSR (Current Program Status Register) [7]. The available instructions can be found listed in full below, it the combination of Figure 1.6 and Table 1.7 of [8]:

		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Move	Move Immediate	0	0	1	MOV			Rd				#8bit_ImmOffset					
	Move High to Low	0	1	0	0	0	1	MOV		H1	H2	Rs				Hd	
	Move Low to High	0	1	0	0	0	1	MOV		H1	H2	Hs				Rd	
	Move High to High	0	1	0	0	0	1	MOV		H1	H2	Hs				Hd	
Arithmetic	Add	0	0	0	1	1	1	ADD	#3bit_ImmOffset			Rs				Rd	
	Add Low and Low	0	0	0	1	1	1	ADD	Rn			Rs				Rd	
	Add High to Low	0	1	0	0	0	1	ADD		H1	H2	Hs				Rd	
	Add Low to High	0	1	0	0	0	1	ADD		H1	H2	Rs				Hd	
	Add High to High	0	1	0	0	0	1	ADD		H1	H2	Hs				Hd	
	Add Immediate	0	0	1	ADD			Rd				#8bit_ImmOffset					
	Add Value to SP	1	0	1	1	0	0	0	0	S		#7bit_ImmOffset					
	Add with Carry	0	1	0	0	0	0	ADD				Rs				Rd	
	Subtract	0	0	0	1	1	1	SUB	#3bit_ImmOffset			Rs				Rd	
	Subtract Immediate	0	0	1	SUB			Rd				#8bit_ImmOffset					
	Subtract with Carry	0	1	0	0	0	0	SUB				Rs				Rd	
	Negate	0	1	0	0	0	0	NEG				Rs				Rd	
	Multiply	0	1	0	0	0	0	MUL				Rs				Rd	
	Compare Low and Low	0	1	0	0	0	1	CMP		H1	H2	Rs				Rd	
	Compare Low and High	0	1	0	0	0	1	CMP		H1	H2	Hs				Rd	
	Compare High and Low	0	1	0	0	0	1	CMP		H1	H2	Rs				Hd	
	Compare High and High	0	1	0	0	0	1	CMP		H1	H2	Hs				Hd	
	Compare Negative	0	1	0	0	0	0	CMN				Rs				Rd	
	Compare Immediate	0	0	1	CMP			Rd				#8bit_ImmOffset					
Logical	AND	0	1	0	0	0	0	AND				Rs				Rd	
	EOR	0	1	0	0	0	0	EOR				Rs				Rd	
	OR	0	1	0	0	0	0	ORR				Rs				Rd	
	Bit Clear	0	1	0	0	0	0	BIC				Rs				Rd	
	Move NOT	0	1	0	0	0	0	MVN				Rs				Rd	
	Test Bit	0	1	0	0	0	0	TST				Rs				Rd	
Shift/Rotate	Logical Shift Left	0	0	0	LSL			#5bit_ImmOffset				Rs				Rd	
	Logical Shift Right	0	0	0	LSR			#5bit_ImmOffset				Rs				Rd	
	Arithmetic Shift Right	0	0	0	ASR			#5bit_ImmOffset				Rs				Rd	
	Rotate Right	0	1	0	0	0	0	ROR				Rs				Rd	
Branch	**Conditional:**																
	if Z set	1	1	0	1			BEQ				label					
	if Z clear	1	1	0	1			BNE				label					
	if C set	1	1	0	1			BCS				label					
	if C clear	1	1	0	1			BCC				label					
	if N set	1	1	0	1			BMI				label					
	if N clear	1	1	0	1			BPL				label					
	if V set	1	1	0	1			BVS				label					
	if V clear	1	1	0	1			BVC				label					
	if C set and Z clear	1	1	0	1			BHI				label					
	if C clear and Z set	1	1	0	1			BLS				label					
	if ((N set and V set) or (N	1	1	0	1			BGE				label					
	if ((N set and V clear) or if	1	1	0	1			BLT				label					
	if (Z clear and ((N or V set)	1	1	0	1			BGT				label					
	if (Z set or ((N set and V cl	1	1	0	1			BLE				label					
	Unconditional	1	1	0	1			B				label					
	Long branch with link	1	1	1	1	H						label					
	Optional state change:																
	to address held in low reg	0	1	0	0	0	1	BX		H1	H2	Rs					
	to address held in Hi reg	0	1	0	0	0	1	BX		H1	H2	Hs					
Load	**With immediate offset:**																
	word	0	1	1			#7bit_ImmOffset					Rb				Rd	
	halfword	0	1	1	B		#6bit_ImmOffset					Rb				Rd	
	byte	0	1	1	B	L	#5bit_ImmOffset					Rb				Rd	
	With register offset:																
	word	0	1	0	1	0	0	0		Ro		Rb				Rd	
	halfword	0	1	0	1	0	B	0		Ro		Rb				Rd	
	signed halfword	0	1	0	1	H	S	1		Ro		Rb				Rd	
	byte	0	1	0	1	L	B	0		Ro		Rb				Rd	
	signed byte	0	1	0	1	H	S	1		Ro		Rb				Rd	
	PC-relative	0	1	0	0	1	Rd					#8bit_ImmOffset					
	SP-relative	1	0	0	1	L	Rd					#8bit_ImmOffset					
	Address:																
	using PC	1	0	1	0	PC	Rd					#8bit_ImmOffset					
	using SP	1	0	1	0	SP	Rd					#8bit_ImmOffset					
	Multiple	1	1	0	0	L	Rb					RegList					
Store	**With immediate offset:**																
	word	0	1	1			#7bit_ImmOffset					Rb				Rd	
	halfword	0	1	1	B		#6bit_ImmOffset					Rb				Rd	
	byte	0	1	1	B	L	#5bit_ImmOffset					Rb				Rd	
	With register offset:																
	word	0	1	0	1	0	0	0		Ro		Rb				Rd	
	halfword	0	1	0	1	0	B	0		Ro		Rb				Rd	
	byte	0	1	0	1	L	B	0		Ro		Rb				Rd	
	SP-relative	1	0	0	1	L	Rd					#8bit_ImmOffset					
	Multiple	1	1	0	0	L	Rb					RegList					
Push/Pop	Push registers on stack	1	0	1	1	0	1	0	0			RegList					
	Push LR, and registers ont	1	0	1	1	L	1	0	R			RegList					
	Pop registers from stack	1	0	1	1	0	1	0	0			RegList					
	Pop registers and PC from	1	0	1	1	L	1	0	R			RegList					
Software Interrupt	Software Interrupt	1	1	0	1	1	1	1	1			#8bit_ImmOffset					

1.2.2) When tied to an ASIP design flow, it allows for a very minimal but functional ASIP design. The ISA chosen for the product design can literally be as minimal as the software running on it needs to be. In all actuality, the ARM Thumb instruction set and the full ARM ISA do not vary drastically in number of Instructions in the Architecture. The main difference is first in the design scheme, where the full ARM ISA involves 32-bit instructions and the Thumb ISA has primarily a 16-Bit instructions with an abstraction layer that maps all instructions to their 32-bit equivalents. At runtime all 16-bit Thumb Instruction are decompressed into full 32-bit ARM instruction in real time. This means that Thumb also maintains all attributes of an ARM 32-bit core such as the 32-bit address space, 32-bit registers, and a 32-bit shifter/ALU [9].

Looking at the difference between the two ISAs also leads to some vital revelations about the difference between the two. Notably, while looking at the Thumb ISA, it is a very complete instruction set where almost any combinational logic may be derived to solve the problem, but each instruction has a small impact on the hardware complexity. The Thumb ISA also has very implicit and guaranteed inputs, whereas the ARM ISA allows for a wide range of varying inputs to an instruction [11]. The full ARM ISA for example adds in the following: incrementing before an operation, full stack operations, Swap, and the ability to move from registers to separate dedicated co-processors. Each of these instructions adds very obvious complexity to the silicon design of the chip that is to run the ISA itself. In the ARM design scheme, a co-processor is a simple term to mean any external logic to the main processor. These are often used to carry out more specialized logic that does not get reprogrammed as much as a general CPU would. So for example, a co-processor might be utilized to perform specific Image Processing Instructions that convert between between digital signal standards that are then passed on to the main Processor [10].

1.2.3) Before citing actual chip specification sheets from manufacturers, it is best to simply to estimate physical chip size and pinout based off of the Instruction Set Complexity. As listed above in 1.2.2, the ARM ISA has a more complex features like the allowance of co-processors that must find a way of being physically mapped into the address space of the main processor, thus requiring more pins for cases like those. We also know that as hardware complexity increase, more Flip-Flops, registers, and transistors, the more wattage the chip must consume out of simple necessity of keeping each of those active components functioning properly. Therefore we know that any chip implementing the full ARM ISA will also consume more power. Physical chip size itself is hard to determine because unlike desktop CPUs, whose size has been very defined for quite a while now, Embedded Processors do not have a single standard size. There are numerous PCB mounting and package standards like the SMT PZ, ZXH and RGC standards, as well as defined package sizes for BGA (Ball Grid Array) packages. The selection of these packages is up to the chip manufacturer however, and many other other design considerations like the transistor size, package price, etc.

With each of those things in mind, we can confirm our assumption by simply pulling up two data sheets for different ARM based chips. The chips I selected, were found on Mouser, an electronics vendor. The choices provided were selected due to their generation in the ARM Cortex series. As noted on the respective data sheets, the Atmel SAM C21E [12] uses just the Thumb-1 and Thumb-2 instruction sets and almost completely covers the standard. The Texas Instruments MSP432P401RIPZ [13] uses the full ARM ISA. While it perhaps isn’t a completely fair comparison as the TI chip adds numerous pin I/O for external ADCs, it is still fair to note that it has by far many more pins than the Atmel chip that just uses the Thumb ISA. The Atmel Thumb chip uses just 32-pins, and the TI chip uses a full 100 pins. As noted, even removing the many external ADC pins would still more than double the pins required to implement the more complex ISA.

1.3.1) To further on the idea of loop unrolling, the author of the paper utilizes two different pipelines in her benchmark tests to see the benefits of the compiler technology across different Architecture designs similar to one another. The base pipeline has a 2-pipe structure where the first pipe is directly responsible for Program Flow Control, or in other words, handling the location of the Program Counter via the Jump, Branch, and other instructions. The second pipe in the 2-pipe structure is used only for data access to the registers. In order to allow each pipe to take control of the registers, each pipe has access to a global Register File that hold all program states. When a command is issued on Pipe 1, a flush command is also issued to remove the current states of any of the pipes, so that the new state can be loaded correctly, albeit with added latency due to this operation. To determine if the processes that they intend to run on the architecture will benefit from more stages in the pipeline, they then increase the number of pipes in addition to the base 2-pipe structure. Each additional pipe simply gets a subset of the entire Thumb ISA so that if more than one instruction is scheduled at a particular timeslot, they can be simultaneously loaded and carried out.

1.3.2) The entire rest of the article focuses on benchmarks that will numerically provide evidence as to whether the programs that the end user would like to run would benefit from more pipelines or the standard 2-pipe architecture. This is very valid since it is one of the main benefits of ASIP design, wherein the architecture and ISA itself can be reformed to ensure the desired runtime qualities are ensured. With each additional pipe stage, we must add two extra read ports and one extra write ports. For each additional pipe we must also provide connections to a new control unit that also has access to the Internal Memory. This design choice means that each additional pipe adds a great deal of complexity and area to the circuit design. As discussed above, this also means that the power consumption will increase in order to supply power to all the new active components. The reason we are thus testing whether a multi-pipeline architecture is beneficial in terms of clock cycles to perform execution is so that we can then weigh the trade offs that will be required to fulfill these greater power requirements. In other words, if the performance outweighs the cost of implementation and power consumption, then it is the optimal choice to choose a greater number of pipes in the pipeline.

As noted in the article, the pipes will have a narrow subset of the entire ISA. The functional units that are included in each pipe will be determined by the number of pipes that are currently being tested. For example, in the 2-pipe architecture, we must have at least and ALU in pipe 1 and a DMAU (Direct Memory Access Unit) in Pipe 2. After this is taken care of, with each additional pipe we have the possibility of splitting the functional units of one pipe into another. It would be highly unwise to simply provide each pipe access to the entire required architecture. It would add unnecessary complexity to a system where a particular instruction may never even be executed in a particular pipe due to how the program is loaded.

1.3.3) There are numerous particular hazards that could occur in such an architecture. The first does not fall into a traditional hazard category itself, but could result in a great deal of lost time. The idea to flush all pipes when a branch instruction occurs on Pipe 1 is a valid option, as it ensures that correct data is loaded into the registers at any given moment of execution, however, it was not discussed at an even lower level if this architecture will force all instructions to be carried out in one clock cycle or not. If it is not the case, then a pipe could be loaded with an instruction that carries out over 4 clock cycles, executes one, and then is flushed before finalizing and storing the output of the computation. This problem of course goes away if we are to assume that each instruction occurs in one clock cycle. Given the simplicity of the ISA, we shall assume such is the case for the rest of this question.

Structural hazards are also present in this design, as each pipe has direct access to the register file. If suppose for example, that we have 3 pipes and each makes a calculation to be stored in R7, then of course, only one of the pipes would have successfully carried out it’s operation. The compiler will

thus have to be written with close attention to detail to ensure that at any given clock cycle, instructions will not be loaded that compete for the same data path.

Most of the Data Hazards will be largely avoided if the same points listed above are applied to the compiler itself. It is also crucial to note that each pipe in the pipeline has forwarding enabled within the pipeline and between pipelines. Forwarding is helpful in cases like the one listed above where multiple instructions may want to access the same register. Rather than write the result from one pipe to the register file, load it into another pipe, and then carry out the instruction, the relevant data may simply be forwarded to another pipe that wishes to make use of the data. In this way, many of the data hazards are avoided.

Lastly, we must take into account any control hazards that may be present in the system. A control hazard is very similar to a RAW Data Hazard in that one pipe wishes to make a conditional decision on data that is not made available. In our architecture, this will most likely mean that some pipes will stall for one clock cycle until at the very least, the information may be forwarded to the pipe requesting the conditional information.

1.4) It is difficult to gather in depth information on the software ASIP Meister as it seems to not be in active development, nonetheless, the original intention and use of the software can easily understood from what remains of Academic papers that made use of the software and the remaining blog posts from it's development. The software was originally created as Osaka University in Japan [15]. As discussed in 1.1, one of the more complex portions dealing with ASIP design has almost nothing to do with the hardware design architecture itself, but the surrounding development environment and tool chain required to make anything of functional value.

As noted in 1.3.2, we wish to quickly determine if adding another pipe to our pipeline is of benefit to our computation time. Without ASIP Meister, that would mean that we would have to decide on which functional units we like to split off into the new pipe, and then additionally re-configure the compiler and assembler so that they can make use of the newly optimized pipes. It would be completely senseless to have additional pipes if the compiler will never assign anything to them, after all. Consequently, it becomes apparent that a large amount of development man hours would be placed into the engineering of the hardware, but also the software environment for the ASIP design.

ASIP Mesiter had intentions to fix these ailments by automatically generating a valid ASIP design in HDL as well as it's matching compiler and assembler. In this way, the software is what truly enables us to try out small variants such as adding another pipe, or changing out what functional units belong to each pipe itself [16].

1.5) Figure 04: Design Flow of Phase II provides a simple graphical representation of what the compilation process entailed for the tests described in this section. Given a specific ASIP design as derived from ASIP Meister, several select C programs were passed through the compilation process to derive the Binary file that was ultimately run for these benchmarking tests.

To be more specific, the C file was passed through the process of Loop Unrolling with it's specified factors. Loop unrolling is a traditional compiler strategy that tries to optimize the execution time of a loop by ultimately repeating the inside commands numerous times and attempts to eliminate the need for branch checking, which in of itself may require pointer arithmetic and end of loop tests. The Loop Unrolling Factors are design contraints where we specify how often we wish for the program to check for branching conditions and end of loop conditions as compared to the desired speed up.

At this point, we then have optimized the assembly code via loop unrolling. Now, based off of the number of pipes available in our pipeline on any given test, we break the program into bits that can be scheduled for the pipeline. For example, in a very simple program where we wish to calculate two numbers, the compiler will break the program into two smaller programs. In each of these programs, they would each carry out exactly one calculation on each of the pipes and then return their results. If

this were not the case, they would all run on the same single pipe and we would see no benefit to the added hardware complexity at all.

Once the program has been scheduled for the ASIP design currently being tested, the regular functions of the assembler take place where the assembly is optimized one final time and translated to the binary machine code. In this step, all other potentially required processes are also assumed to occur such as linking.

Finally, this program that has gone through the full process of being loop unrolled and scheduled for the actual ASIP design is run various times to measure the performance in terms of the number of clock cycles as well as keeping track of the code size. While the speed of execution is obviously important, it is equally important to consider the size of the program on an embedded device. Embedded devices are notorious for having very limited memory, and thus, if no real performance is gained despite having a much larger program file size, then we can scale back on the loop unrolling factors that would otherwise result in the desired balance between performance and file size. Tables 1 and 2 of this section provide results of the described tests.

1.6) Overall, it could be said that from the results provided in Tables 1 and 2 of this section, the Three-Pipeline architecture is worth implementing. The tables are extremely simple, providing the number of clock cycles required for execution of a program that has not gone through a loop unrolling procedure of the compiler, the clock cycles of the loop unrolled program, and then improvement which is defined as the percentage difference between the two programs.

Looking first at just Table 1, we can definitely see that the compilation time required to perform loop unrolling is definitely worth it in the end. The end user does not see this complexity and only sees the optimized execution times that occur. Of the six chosen C programs, all saw improvements in their performance by at least 10%, with two of the six seeing nearly 30% improvement.

Now looking at the two tables together, we must weigh whether the choice of additional pipes in the pipeline are justified. In this case, it is readily apparent that they are not justified. We must remember that with an additional pipeline we will see increases to the power consumption of the architecture. Therefore, we must provide sufficient gains to justify the increased power consumption, which limits their application in remote cases and the like. The Bubble Sort, Bit Count, and CRC32 programs all executed in nearly the exact same amount of time, with any improvements being negligible. The Bit String and Bit Shifter programs both saw slight decreases in performance, and only the encryption program saw notable improvements of 2%. Thus to reiterate, if this chip is being sold as a re programmable device, then it most certainly is not beneficial to add a third pipeline. If however, this chip was being designed by an Encryption company who wants to market the device as only being used for this specific encryption algorithm, then perhaps you may have further discussion on whether the performance increases warrant the increased price.

References:

1) http://web.engr.oregonstate.edu/~qassimy/index_files/Final_ECE570_ASP_2012_Project_Report.pdf
2) http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5351271
3) https://riscv.org/wp-content/uploads/2016/12/Tue1100-RISC-V-Workshop-RoHC-ASIP-Accelerator-Cox-Synopsys.pdf
4) https://inst.cs.berkeley.edu/~cs152/fa04/handouts/keutzer.pdf
5) http://www.ti.com/lsds/ti/processors/dsp/c6000_dsp/overview.page
6) http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0210c/ch02s06s04.html
7) http://users.ece.utexas.edu/~valvano/EE345M/Arm_EE382N_4.pdf, Page 8.

8) http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0210c/I1040101.html
9) http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0210c/CACBCAAE.html
10) http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0333h/Bgbhbiah.html
11) https://en.wikipedia.org/wiki/ARM_architecture#Thumb
12) http://www.mouser.com/ds/2/268/Atmel-42365-SAM-C21_Datasheet-Summary-1068326.pdf
13) http://www.ti.com/lit/ds/symlink/msp432p401r.pdf
14) http://www.cs.virginia.edu/kim/courses/cs3330/notes/PipelineHazards.pdf
15) http://www-ise1.ist.osaka-u.ac.jp/lab/research/processor/
16) http://aranea.ics.es.osaka-u.ac.jp/dac2003/