JouleSort Using a Serial Array of Merge Sorts on AsAP2

AARON STILLMAKER
University of California, Davis

and

LUCAS STILLMAKER
University of California, Davis

Energy efficiency is an important part of embedded systems for many applications ranging from large data centers to mobile devices. Many applications, especially those used in data centers, make heavy use of sorting algorithms. We propose to take the AsAP2 chip that was specifically designed for low power DSP applications, and implement a novel sorting algorithm to sort large lists of data and measure its energy efficiency using the JouleSort. We will look at other methods of sorting and rationalize why we decided to use our method.

Categories and Subject Descriptors: E.5 [Data]: Files; H.2.8 [Database Management]: Database Applications

General Terms: Sorting, Database

Additional Key Words and Phrases: JouleSort, Database Sorting, ASAP2

1. INTRODUCTION

Energy efficiency is becoming more and more important in todays computer market with a growing number of mobile applications and the growing sizes of data centers. As the storage space, and physical size of the modern large scale data center grows, so does the cost to power it. One of the larger costs of running a data center is the cost to power it, and this cost is taken in high consideration as new data centers are being created. This leaves a large demand for a cheaper way to run these data centers.

Data centers spend a large amount of their processing time sorting large databases [Graefe 1993], which makes a convenient standard by which to measure energy efficiency of a system. The JouleSort [Rivoire et al. 2007] was developed at Stanford...
University for this exact reason, to measure the joules required to sort one database entry. They developed a set of standards that we will use to measure our results. In this method one can choose between sorting 10 GB, 100GB, or 1TB. Each database entry contains a 10 byte randomly permuted key and 100 bytes of data.

The Asynchronous Array of Processors, version 2.0, or AsAP2 [Truong et al. 2008] chip was developed at the University of California, Davis to offer a large array of simple low power processors for energy efficient digital signal processing applications. The chip contains 164 general-purpose processors and 3 specific task processors, with inputs coming in on one side and outputs leaving on the other. Each processor can communicate with its nearest neighbor or via long distance communication bypass through unconcerned processors. This setup works well with DSP applications that are generally easily parallelized in a pipeline type fashion, with data being computed then passed to the next processor for more computation.

We propose using the AsAP2 chip to perform our idea of a Serial Array of Merge Sorts, which we will call SAMSort. This chip has never been used for a database application, and was not designed for this purpose. However, we were able to exploit the chips traits to make it efficiently sort large amounts of data.

The rest of this paper will be organized as follows. Section 2 will take a look at different methods of sorting databases and how they would perform when implemented on the AsAP2 chip. It will then rationalize why we chose our method of sorting. Section 3 will discuss in detail how we designed our algorithm, and how we implemented it on the AsAP2 chip. Section 4 will look at the results we gathered from the simulations of our implementation, showing the power usage and comparing those to the top JouleSort results from Stanford. Section 5 will be our conclusion, containing the difficulties we ran into with this project, our findings and our thoughts on the project. Section 6 will discuss what we would like to do with future works.

2. SORTING ALGORITHMS

There has been much research into sorting algorithms for database systems. In years past the research has mainly focused on single-core sequential algorithms [Taniar and Rahayu 2002]. Now that multiple core processors are becoming more mainstream, in the last few years there has been more of a look into what algorithms would be most efficient for multi and many core processors.

The binary merge sort is one of the more widely used sorts currently in database sorting. Binary merge sorts work by merging two sorted lists, into one larger sorted list, then merging that larger sorted list with another list, continuing on in this way until there is only one list left [Bitton et al. 1983]. This uses multi-core processors fairly well, as all cores will be working in parallel until you get down to the last few phases. For example if you have a quad-core processor, all the processors can work simultaneously until you get down to the second to last phase, where you only have two pairs of lists to merge, then the last phase would only have one pair to merge, so you would use only two and one processors respectively. This is not to much of a problem when you are looking at list sizes that are large, and working with chips that do not use a large amount of processors, for example if you are sorting 10GB of information with 110byte list entries like in the JouleSort, you would need to use...
20 phases, and with a quad-core chip, only the last two phases would not use the full extent of the chip.

Where the binary merge sort stops being as efficient is if you are looking at many-core systems. For example we are using the AsAP processor that has 164 cores on the chip, this would mean that 9 phases from the end is where not all the processors will be used, and each phase after the 9th from the end will use half the amount of processors as the phase before it. For this example with 10GB of 110byte list entries, of the 20 phases, 9 would not use all of the processors. The other downside of binary merge sorts is the sheer amount of phases that need to be used, because you only reduce the amount of lists you have by half with every phase completed. Every phase of merge requires all data being looked at to be read and written, and as reads and writes are the main things that slow down external sorts, this will slower than an algorithm that could perform the sort in less phases.

Tenair et al. looked into which type of sort would achieve the best results for multi and many core systems, and came to the conclusion that the less amount of phases you can sort a list in, the better [Taniar and Rahayu 2000] [Taniar and Rahayu 2002]. He found, as we have previously reasoned, that if you must run many more phases, a binary merge sort ends up being not very efficient just because of the amount of reads and writes that would need to be performed. They proposed an innovative method of sorting where after every phase; the lists would be redistributed according to the range of their keys [Taniar and Rahayu 2000] [Taniar and Rahayu 2002]. For example if there are 100 possible keys, and you have 10 lists, you could section them off for the first 10 into one list, the second 10 into another list and so on. This helps out the sort because you would never need to actually merge all of the numbers back into one list, as each list would already be ordered correctly. Their proposed redistribution would not work well with our system because with the 10 byte key we are working with, there is a very large range of possible values which are not necessarily part of our randomly generated list and splitting the total possible values into ranges would make the size of each list vary considerably. They also decided that a merge all sort would be the most efficient algorithm because it reduces any size merge into just two phases, the only problem with the sort is that you would need to be able to effectively merge a very large amount of lists into one. This does not work well with our sort, because the AsAP2 could not easily merge hundreds of thousands of lists at the same time.

We looked into using other sorts, such as the QuickSort, but we quickly were able to rule these out. The AsAP2 chip has a limitation of 128 instructions per processor and more complicated sorting methods simply would not work with our hardware. This along with our research showing that merge sort was the best way to go for sorting databases made us turn towards modifying the merge sort to best fit our hardware.

2.1 Design
We focused our design on the 10GB option of the JouleSort. When we were deciding how to implement our sort, we looked into what had been done in the past to figure out why certain sorts did better than other ones. As external memory reads and writes take up a considerable amount of time in a system, we wanted to try and minimize them. We also noticed that the amount of time that an algorithm ran
would play a large role in the total energy used, so we attempted to minimize the amount of passes we made of reading and writing the data. We also knew that the amount of phases a sort had to go through before it was completely sorted would increase the amount of reads and write considerably, and the way to reduce the amount of phases is to increase the amount of lists that could be combined at once.

We knew that we were going to have entries from many different lists in the external memory, so we were going to need to keep track of each entry, and where it came from. Which would need to be accomplished by administrative processors. The local memory on each chip is 128 2-byte words that could hold two complete entries (the 10 byte key and 100 bytes of data). This means that the most efficient way we can use the algorithm would be to fill the processors up as much as many entries as possible.

We also decided that there are going to be two distinct modes for our sort. Because at the beginning absolutely every entry will be out of order, and we can fit 328 entries into the AsAP 2 processor, we decided that the first mode should just sort every entry into lists of 328 entries. This would be accomplished by simply inputting 328 entries to the processor at a time, and the output of the processor would be a sorted list. After this first mode is completed, there will be 330,892 sorted lists that each have 328 entries in them.

The second mode that we have implemented will merge all of the lists into one complete sorted list. To create an optimal sorting algorithm we knew we would need to minimize the number of memory accesses. If we used the Snake mode for merging multiple sorted lists together we would need to wait for the output to see what list the entry came from before we could send in another input. We know that one of the larger time consumer would be the memory accesses, so we designed a simulation (described later) that showed us that using the Snake mode would not be effective time wise for merging this large amount of presorted lists because of the amount of time that each entry would spend moving through the chip. We then decided to look into implementing a string of merges where we were able to buffer entries in processors, thereby getting rid of the need to wait for each new entry individually. In designing this we found that there is a trade off between merging more lists at one time, and fewer lists at one time. If we merge a large number of lists at one time, then we have a small number of phases before we are down to one list, meaning we will have fewer total reads and writes of the whole chunk of memory to be sorted. If we merge a smaller number of lists we would need more phases, but we would be able to have larger buffers, meaning we could reduce the chance of stalling the chip because a list in a buffer has been used up.

Because the largest downfall of our previous method was the amount of time each entry would take to move its way through 164 processors, we knew we would have to have each entry go through a smaller number of processors before it exited the chip. We decided that it would be more efficient to make a system that would use physical buffers to store extra entries from a list. We also knew that the larger the number of lists we could combine at once, the less amount of combines would need to occur before the lists were all merged into one. When we were looking at how to plan out the processor, we found that if we merged more than 24 entries at one time we would have to use a considerable amount of processors for transmission lines,
to the point that the chip would need to stall its output as entries worked their way around the processor, which is what we are trying to avoid. Because of this, we decided to use a setup that would allow us to combine 24 lists at a time. After deciding on how to execute the sort, we implemented it onto the AsAP2 processor.

3. IMPLEMENTATION

We are going to be using two different modes to implement our sort. The first mode, which we will refer to as the Snake Mode, will be where we input unsorted data, and it will snake through a series of merge sorts, and will output sorted lists containing 328 entries. The second mode we will refer to as the Buffer Mode, where 24 sorted lists will be merged in the AsAP2 processor, while the processors not being used for the merging will be used to buffer data from the 24 sorted lists.

3.1 Snake Mode

We were working with the AsAP2 chip that has its own modified/simplified MIPS instruction set. Each individual processor has only room for 128 instructions in their instruction memory. Because of this restriction we focused on keeping the programming simple and short. Throughout the code we were able to make use of the address generation feature on the AsAP2 chip automatically increments to the next address in a preset list. This allowed us to handle entries that are 56 words long (55 2-byte words with a 2-byte tag), without going over out 128 instruction limit. We decided to have each entry tagged as it goes into the AsAP2 chip, this allows us to keep track of where each entry came from in memory, as well as send necessary administrative signals to the processors in the array.

Our program has a separate initializing configuration routine that runs and initializes all of the data we needed to use in the algorithm such as the masks to compare if certain codes are found in the keys of the entries. The main algorithm will first search to see if the tag has the reset command code, and if it does it will flush the entries currently saved in the processor, and output the reset command so the next processor will output and clear its memory as well. If there is no such code, the program will then check to see how many entries are already in the processor. If the processor is empty it will load the incoming entry to the higher slot. If there is only one entry already there it will call the sorting routine to determine where the two entries should be stored in processor. If there are already two entries it will move to the main part of the sorting algorithm.

In the main part of the algorithm the program will compare the keys, one word at a time, of the incoming entry and the lowest stored entry. If it finds that the incoming is lower then the lowest stored value, it will output the incoming entry directly to the output. If it finds that the incoming entry is higher then the lowest in the processor it will output the lowest stored entry, then call the sorting routine to determine which place the incoming and higher stored entry should be stored in.

The sorting routine will compare the keys of the two entries, one word at a time, and will store the lower entry in the location reserved for it, and store the higher entry in its respective slot. The code can be seen in Appendix A.

Because of the nature of our algorithm, we are able to load the same program on each of our processors and connect them all in a snake-like fashion. The AsAP2 does not currently have any protocol to input or output data from the chip, so
for testing purposes we used one processor to generate test data to input into the snake. We needed to make sure that we used as many processors as possible while keeping a continuous string of processors from the input of the chip to the output of the chip. We ultimately came up with a mapping path shown in Figure 1. This way we were able to keep a constant chain of processors communicating with their nearest neighbor throughout the AsAP2 chip. With this configuration we can sort up to 328 lists at the same time, with 2 entries in each of the 164 processors being used, then one more entry can be added that will work its way through the processor and pop out the correct value from the last processor.

We were then able to use the simulator to estimate the power consumption of our program to sort the 328 different entries that could be stored in the chip at one time. We were also able to successfully implement and run this program on the actual AsAP2 chip in the lab, and it ran the same way that the simulation stipulated it would.

3.2 Buffer Mode
There are two rows in the middle of the chip that merge the incoming list with the running list that is going down the chip, see Figure 2 where the colors mean: Light Blue = Merge Processors, Yellow = Transmission lines, and Dark Blue = Administrative processors. At the far right, the two merged lists are combined before they are output from the chip. The perimeter processors are used to transmit the data to fill the lists. We still use the bottom two rows for administrative processors, and the rest of the processors are used as buffer space, where their data memory and FIFOs are used to store three entries per processor. As the entries go into the processor they are tagged by an administrative processor, which specifies which list the entry came from.

The buffer processors are programmed to check and see if there is a new entry
waiting in the input FIFO and if there is one, take the next entry in line to be sent on from the memory, and place it in the output FIFO, then take the incoming entry and place it in the data memory where the previous entry was, then change the counter so that the other entry will be taken the next time. If there is no entry waiting to enter the processor, then it will output the next entry that is in the processor and next to go. In this way the buffer will continually be trying to empty itself into the line of merging processors. There are also input FIFO buffers in the chip that will store information that is going to be input into the chip, which can each hold 64 2-byte words. With this method of forwarding information as soon as possible to the FIFOs, we were able to fit 3 full entries in each buffer processor.

The merging processors are coded to load the keys from the two entries from the two input FIFOs, then it will compare the two keys, pass on the lower entry, and keep the higher key stored. Then it can take the next key from whichever input port that was passed through, compare that with what is already in the memory, and pass on the lower of the two entries.

As the entries leave the processor, it will read the tag that matches the entry with its list of origin, and then it will call for the next entry in that list to be sent to the processor. After a list runs though all its entries, it will send the reset tag that will make the merge processors always pass the other incoming entry. This will continue until both inputs have sent the reset tag, at which point it will forward this tag to the output then wait for new entries.

The administrative processors, besides tagging, would be in charge of calling the lists to be merged, keeping track of where all of the lists start and how long they are. Due to time constraints and lack of functionality in the AsAP2 for this project we are going to only be implementing the first of the two modes, and did not program the administrative processors. However, we did create rigorous simulations to calculate the optimal way to sort the remaining lists together. Thus the chip
will process until it is filled up, then it will receive a reset tag that will flush the newly sorted list out the output.

3.3 Simulation

We were able to model and design how the full sort would work, even though we did not physically implement all parts of the sort. We wrote a MatLab script to model how long the delay time would be for transferring and accessing memory, which varies depending on how many lists are to be merged at one time, and would find the optimal solution. We made this at first assuming we would reuse the snaking mode, and just input different lists, but we determined this would not be the most efficient way to implement the system. We decided it would work better if we used the processors to act as buffers and merge lists in the middle, so we tweaked our script to model this instead. We were able to compare our Matlab script output with the Verilog simulation output and they were very similar, validating our previous timing simulations, see Appendix G.

After much deliberation and simulation we decided that it would be best to merge 24 lists at the same time. We worked out different configuration for the processors inside the chip to optimize our buffer space, and found that we could fit up to 24 buffers in a way that we would have effectively no stalls (if all of the entries come from one of the two lists furthest from the output, there will be a few stalls, but this is very unlikely and even if it does occur will only slightly slow down the chip). When adding a 25th buffer to the chip, another transmission line and merge line would need to be added on top of the two of each we already had, which takes away a large amount of processors available for buffers, and would make stalls fairly prevalent, slowing down the chip considerably. After we knew that we could only go up to 24 merged lists at a time, we created a simulation to model one gigabyte of data being sorted using a varying number of merged lists from 2 to 24, see Figure 3. Our simulation informed us that 24 lists merged would give us the quickest speed, and because our main power cost are the fixed costs of the memory and the development board, 24 is the most power efficient solution as well, see Appendix G.

4. RESULTS

We were able to use a simulation tool built for the AsAP2 chip to calculate the power usage of the chip. The tool gave us an average power of 0.228997W usage on average to run all of the general purpose processors at 1Ghz. We were then able to take the average power information of the other pieces of our system to determine as simulated value of what the total power usage would be. The external hard drive we will be using is a 32GB SanDisk USB Flash Drive, which consumes around 0.6W on average. We were able to use an electricity usage monitor to find that the Xilinx Virtex-5 LX50 uses 7W. The Xilinx power reading includes all other power of the system except the AsAP2 chip that is powered by power supplies off board. We would then be adding some RAM memory, which would take about 1.9W on average. This brings the total average power to 9.728997W for the complete system.

We were able to use the equation: 50ns (RAM access time) + 34.375ns (transfer time for 110B at 3.2BG/s) + 149000ns (time to fill up all of the processors being used) + 97612893 (number of entries) * 1000ns (time to output one entry) which
showed us that it would take 97.613sec for the initial snake mode. We then used the MatLab script, Appendix G, which showed us that it would take 39.799sec to sort each individual 1GB of information. We used the equation: 50s (RAM access time) + 34.375s (transfer time for 110B at 3.2BG/s) + 156000s (time to fill up all of the processors being used) + 97612893 (number of entries) * 1000s (time to output one entry) then it would take 97.613sec to merge the 10-1Gb lists into one final list. This would take a total of 593.216sec of time to sort the 10GB database.

When you combine these two numbers, we get the total joules used for the sort, 9.728997W(J/s) x 593.216sec = 5,771.3967J. This means that our Joule cost per sorted record would be 97612893Recs/ 5,771.3967J = 16,913Recs/J.

4.1 Comparison

The best system that was tested by Rivoire et al. could sort 10GB with an average power of 99.3W, complete the sorting in 86.6sec, which would give an energy usage of 8,600J [Rivoire et al. 2007]. This gives them 11,628SRecs/J. When compared to our simulated results, one can see that our system would be able to use a much lower power, while performing the sort at a slower rate, yet still able to complete the whole sort for about 2,828J less. This translates to allowing our system to be able to sort about 5,285 more records per Joule expended. Of course one must remember that ours are simulated numbers while the numbers from Stanford are realized values, yet we are fairly confident that our system should perform as we expect.

The paper from Rivoire et al. mentions that a large downside to their systems was that their CPUs were very underutilized, and that they were limited by the I/O components. In our system because of the slowness of computation, the CPUs should be utilized around 100% of the time, which makes our system more balanced.
Table I. The results of sorting 10GB from the winner of JouleSort (Rivoire et al.) and using the SAMSort on AsAP2.

<table>
<thead>
<tr>
<th>Sort</th>
<th>SortedRecs/J</th>
<th>Energy(kJ)</th>
<th>Power(W)</th>
<th>Time(Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMSort (Estimated)</td>
<td>16,913</td>
<td>5.77</td>
<td>9.73</td>
<td>593.216</td>
</tr>
<tr>
<td>JouleSortWinner</td>
<td>11,628</td>
<td>8.6</td>
<td>99.3</td>
<td>86.6</td>
</tr>
</tbody>
</table>

5. CONCLUSIONS

From our findings we are able to conclude that the AsAP2 chip can indeed be used to perform low power database sorts. It can be seen that this option would fit best if time were not the most important issue, because our sort would take much longer then other low powered sorts. Our idea also creates a good stepping-stone from which others can expand the idea of sorting with a many core system. There are currently few programs, and specifically sorting algorithms that are created to work with many core systems. Our project was a first for the AsAP2 chip, as it utilized all but 3 processors on the chip, which helps forward our groups research with utilizing the AsAP2 chip.

5.1 Difficulties

We ran into many obstacles while working on this project. We decided to use a processor that we had never worked with before, and was specially designed to be a low power DSP processor. It had its own modified instruction set which we needed to learn as well as many different rules specific to the processor. The processor can only take data in one side and output on the other. There are also many data hazards associated with different instructions. The data flow needed to be mapped through each processor and configured as such. There were many configuration specifications that we needed to learn and program within. There is currently no way to send data on or off the chip, so our experiment had to be limited to what data we could generate on chip.

During the process we discovered an issue with the scheduler tool that our team uses, and helped figure out the issue. In the end the scheduler could not be fixed in time, so we manually needed to reschedule our program and add the needed delays.

An implementation of this scope, using all of the general-purpose processors on the AsAP2 chip, has never been attempted. As such, we had to write scripts to help us program the 164 processors. This also meant that our simulation time using Verilog was very large, because it had to simulate the processing of 164 processors working at the same time.

As the AsAP2 processor was designed as a DSP processor, so it does not have very much memory on each processor or even on the whole chip. We were thus limited to need to use more reads and writes then we would have liked, because we could not sort as much of the list on chip then with a general processor with larger cache sizes.

A good deal of time was spent researching different sorting algorithms, and discussing with each other how they could be implemented, if possible. There is not that much research out there specifically on parallel sorting methods, as many-core processing systems are relatively new. On top of this we were limited because the
AsAP2 chip has a specific input and output system that was not designed to handle parallel loading or writing, which made many algorithms inefficient.

6. FUTURE WORK

Our main goal for the future would be to get a complete database sorting system operational. Currently we have implemented the sorting algorithm that we could use in the system, but we need everything outside of this to make a complete system. The largest obstacle we face is that the AsAP2 currently has no functionality in place to input or output data from the chip. Currently one needs to just program data into the chip and you can read a test out to make sure correct values are being output. We would like to create or help create a FIFO buffer for both output and input into the chip. From there we would need to be able to create a memory interface where the input and output buffers could read and write information from a memory source.

Another problem we can already see is the long sorting time in our sort. We believe this can largely be attributed to inefficiencies on our part when we were dealing with the different hazards, we created loops and were on the safe side of putting in NOP commands to make sure no hazards brought down the program. We would like to go back and find out exactly how long we should stall the processor so we do not have as many wasted clock cycles. There are many loops that have NOPs in them, and because of this we feel that we could drastically reduce the time required to do the SAMSort.

We also planned on having administrative processors in the array that would control tagging incoming entries that would identify the starting and ending address of that the entry was taken from. We would then be able to manage fetching more data from a certain list when the entry from that list was output. We would need to look into whether or not this would be possible on a AsAP2 processor, because with the limit of 128 instructions, the program might be too complicated, in which case we could implement it on a separate low powered processor, which would administrate the flow of data in and out of the AsAP2 chip.

With this running, we could implement the full sort, sorting larger database sizes, such as the 100GB or 1TB sizes mentioned in the JouleSort.

REFERENCES


Report for ECE284: Dr. Ghiasi, Vol. 0, No. 0, 09 2009.
APPENDIX

The following programs are the templates that we used to program the AsAP2 chip, the script we used to make the final large program, and the MatLab simulation file we used to estimate time.

A. SNAKESORT.ASSY

This code is loaded into every processor in the Snake Mode. It reads two entries and stores them in order according to their keys. It will then check a third incoming entry to determine which should be output, and where the two remaining entries should be stored.

begin x,y

#output (direction)

INPU:
MOVE ag2 Ibuf0 //save the tag of the incoming element
NOP NOP3
XOR NULL Dmem 112 Dmem 119 NOP3 //check to see if reset code is in the tag
BRNZ BELONG NOP3
RPTB #56 #2 NOP3 //flush the elements out of the processor
MOVE Obuf ag0pi NOP3
NOP NOP3
RPTB #56 #2 NOP3
MOVE Obuf ag1pi NOP3
NOP NOP3
MOVI Dmem 118 b0000000000000000 //resets the counter
MOVE Obuf Dmem 112 //send the reset signal to the next processor
BR INPU NOP1 //branch to the beginning to wait for the next input

BELONG:
XOR NULL Dmem 118 Dmem 120 NOP3 //check if counter is at 0
BRNZ BELONG2 NOP1
NOP NOP3
MOVI Dmem 118 b0000000000000001
MOVE ag1pi Dmem 112 //save the input if it is 0
RPTB #55 #2 NOP3
MOVE ag1pi Ibuf0 NOP3
NOP NOP3
BR INPU NOP1 //branch back to the beginning

BELONG2:
XOR NULL Dmem 118 Dmem 121 NOP3 //check if counter is at 1
BRNZ BELONG3 NOP3
MOVE NULL ag2pi NOP3 //increment the address pointer
MOVU Dmem 118 b0000000000000000
RPTB #5 #2 NOP3
MOVE ag2pi Ibuf0 NOP3 //save the input if it is 1
NOP NOP3
BR SORT NOP1 //branch back to the sorting code

BELONG3:
MOVE NULL ag2pi NOP3 //increment the address pointer
RPTB #5 #2 NOP3
NOP NOP3
MOVE ag2pi Ibuf0 NOP3 //save the key of the incoming element
XOR NULL Dmem 113 Dmem 1 NOP3
//check if the keys are equal one word at a time
NOP NOP3
BRNZ CHECK1 NOP3 //if they are not equal check which is bigger
XOR NULL Dmem 114 Dmem 2 NOP3
BRNZ CHECK2 NOP3
XOR NULL Dmem 115 Dmem 3 NOP3
BRNZ CHECK3 NOP3
XOR NULL Dmem 116 Dmem 4 NOP3
BRNZ CHECK4 NOP3
MAX NULL Dmem 117 Dmem 5 NOP3
BRMS2 PASS NOP1
NOP NOP3
RPTB #56 #2 NOP3
MOVE Obuf ag0pi NOP3
NOP NOP3
BR SORT NOP1

CHECK1: //if it belongs in the list sort, if not pass it
MAX NULL Dmem 113 Dmem 1 NOP3
BRMS2 PASS NOP1
NOP NOP3
RPTB #56 #2 NOP3 //output the lowest element
MOVE Obuf ag0pi NOP3
NOP NOP3
BR SORT NOP1 //branch to the sorting code

CHECK2:
MAX NULL Dmem 114 Dmem 2 NOP3
BRMS2 PASS NOP1
NOP NOP3
RPTB #56 #2 NOP3
MOVE Obuf ag0pi NOP3
A. and L. Stillmaker

NOP NOP3
BR SORT NOP1

CHECK3:
MAX NULL Dmem 115 Dmem 3 NOP3
BRMS2 PASS NOP1
NOP NOP3
RPTB #56 #2 NOP3
MOVE Obuf ag0pi NOP3
NOP NOP3
BR SORT NOP1

CHECK4:
MAX NULL Dmem 116 Dmem 4 NOP3
BRMS2 PASS NOP1
RPTB #56 #2 NOP3
MOVE Obuf ag0pi NOP3
NOP NOP3
BR SORT NOP1

PASS:

NOP
RPTB #6 #2 NOP3 //output the key and tag
MOVE Obuf ag2pi NOP3
NOP NOP3
RPTB #50 #2 NOP3 //then the rest of the element
MOVE Obuf Ibuf0 NOP3
NOP NOP3
BR INPU NOP1 //start again

SORT: // Sorts all remaining entries
XOR Null DMEM 113 DMEM 57 NOP3 // Checks to see if the first word
BRNZ maxx1 NOP1 // of the key is the same
XOR Null DMEM 114 DMEM 58 NOP3 // Checks second word
BRNZ maxx2 NOP1
XOR Null DMEM 115 DMEM 59 NOP3
BRNZ maxx3 NOP1
XOR Null DMEM 116 DMEM 60 NOP3
BRNZ maxx4 NOP1

//maxx5: If all words are different
MAX Null DMEM 117 DMEM 61 NOP3 // Find which word is larger
BRMS1 new_bigger NOP1
NOP NOP3
BR old_bigger NOP1
maxx1: // If the first word is different
   MAX Null DMEM 113 DMEM 57   NOP3 // Find which word is larger
   BRMS1 new_bigger NOP1
   BR old_bigger NOP1
maxx2:
   MAX Null DMEM 114 DMEM 58 NOP3
   BRMS1 new_bigger NOP1
   BR old_bigger NOP1
maxx3:
   MAX Null DMEM 115 DMEM 59 NOP3
   BRMS1 new_bigger NOP1
   BR old_bigger NOP1
maxx4:
   MAX Null DMEM 116 DMEM 60 NOP3
   BRMS2 old_bigger NOP1

new_bigger: // If the inputted entry is larger
   RPTB #56 #2 NOP3
   MOVE ag0pi ag1pi NOP3 // Moves the old entry down a rank
   NOP NOP3
   RPTB #6 #2 NOP3
   MOVE ag1pi ag2pi NOP3 // Puts the new key into the top rank
   NOP NOP3
   RPTB #50 #2 NOP3
   MOVE ag1pi Ibuf0 NOP3 // Moves the new entry into the top rank
   NOP NOP3
   BR INPU NOP1 // Loops back to the beginning

old_bigger:
   RPTB #6 #2 NOP3
   MOVE ag0pi ag2pi NOP3 // Puts the new key into the lower rank
   NOP NOP3
   RPTB #50 #2 NOP3
   MOVE ag0pi Ibuf0 NOP3 // Puts the new entry into the lower rank
   NOP NOP3
   BR INPU NOP1 // Branches back to the beginning
end

B. CFG00.ASSY - SNAKE AND BUFFER MODES

This program is used during the configuration setup of the AsAP2 chip to preload data into the memory. This code is modified slightly for a few of the Buffer Mode processors.

   begin x,y
//Initialize Address Generator

MOVI Dmem 118 b0000000000000000 //set the counter to zero
MOVI DCmem 2 b0000000000100000 //setup for the AG, br=0, dir=1, shr=0
MOVI DCmem 6 b0000000000100000
MOVI DCmem 10 b0000000000100000
MOVI DCmem 3 b0000000000110111 //start=0 end=55
MOVI DCmem 7 b0011100001101111 //start=56 end=111
MOVI DCmem 11 b0111000001110101 //start=112 end=117
MOVI DCmem 4 b0000000101111111 // stride=1, sml=1111111
MOVI DCmem 5 b0111111100000000 // mask_and=127, mask_or=0
MOVI DCmem 8 b0000000101111111 // stride=1, sml=1111111
MOVI DCmem 9 b0111111100000000 // mask_and=127, mask_or=0
MOVI DCmem 12 b0000000101111111 // stride=1, sml=1111111
MOVI DCmem 13 b0111111100000000 // mask_and=127, mask_or=0
MOVI Dmem 119 b1010100000000000 //set masks
MOVI Dmem 120 b0000000000000000
MOVI Dmem 121 b0000000000000001

LOOP:
BR LOOP

end

C. BUFFER.ASSY

This program is loaded into the processors which are acting as Buffers in the Buffer Mode. This will take in entries and check to see if it belongs in the buffer, and if it does it will save them. It will then output the entries, then fill the empty space back up, if there is ever not an input, it will still output, then fill in the empty memory once it gets an input.

begin x,y

#output (direction)

INPU:
MOVE Dmem 112 Ibuf0 //save the tag of the incoming element
NOP NOP3

XOR NULL Dmem 112 Dmem 119 NOP3 //check to see if reset code is in the tag
BRNZ BELONG NOP3

MOVE Dmem 117 Ibuf0
XOR NULL Dmem 117 Dmem 124 NOP3  
BRNZ INPU  

XOR NULL Dmem 125 Dmem 121 NOP3 //Check if the flag is set  
BRZ BELONG6 NOP3  
NOP NOP3  

XOR NULL Dmem 118 Dmem 120 NOP3 //check if counter is at 0  
BRZ DUMP NOP3  

XOR NULL Dmem 118 Dmem 122 NOP3 //check if counter is at 2  
BRZ CLEAR NOP3  

XOR NULL Dmem 118 Dmem 123 NOP3 //check if counter is at 3  
BRZ CLEAR2 NOP3  

BR BELONG5  

CLEAR:  
RPTB #56 #2 NOP3 //flush the elements out of the processor  
MOVE Obuf ag0pi NOP3  
NOP NOP3  
RPTB #56 #2 NOP3  
MOVE Obuf ag1pi NOP3  
NOP NOP3  
MOVI Dmem 118 b0000000000000000 //resets the counter  
MOVE Obuf Dmem 112 //send the reset signal to the next processor  
BR INPU NOP1 //branch to the beginning to wait for the next input  

CLEAR2:  
RPTB #56 #2 NOP3 //flush the elements out of the processor  
MOVE Obuf ag1pi NOP3  
NOP NOP3  
RPTB #56 #2 NOP3  
MOVE Obuf ag0pi NOP3  
NOP NOP3  
MOVI Dmem 118 b0000000000000000 //resets the counter  
MOVE Obuf Dmem 112 //send the reset signal to the next processor  
BR INPU NOP1 //branch to the beginning to wait for the next input  

BELONG:  

XOR NULL Dmem 112 Dmem 124 //Check to see if this belongs in this buffer
A. and L. Stillmaker

NOP NOP3
BRNZ DELETE NOP3
NOP NOP3

XOR NULL Dmem 118 Dmem 120 NOP3 //check if counter is at 0
BRNZ BELONG2 NOP1
NOP NOP3
MOVI Dmem 118 b0000000000000001
XOR NULL Dmem 125 Dmem 121 NOP3 //check to see if flag is set to 1
BRNZ BELONG1 NOP3
MOVI Dmem 118 b0000000000000001
//sets the counter to 3 since ag0 and 1 are full and 1 needs to leave first
MOVI Dmem 125 b0000000000000000 //clears the

BELONG1:

MOVE ag0pi Dmem 112 //save the input if it is 0
RPTB #55 #2 NOP3
MOVE ag0pi Ibuf0 NOP3
NOP NOP3
BRF0 BELONG5 NOP3
BR INPU

BELONG2:
XOR NULL Dmem 118 Dmem 121 NOP3 //check if counter is at 1
BRNZ BELONG3 NOP3
MOVE ag1pi Dmem 112 NOP3
MOVI Dmem 118 b0000000000000010
RPTB #55 #2 NOP3
MOVE ag1pi Ibuf0 NOP3 //save the input if it is 1
NOP NOP3
BRF0 BELONG5 NOP3
BR INPU

BELONG3:

XOR NULL Dmem 118 Dmem 122 NOP3 //check if counter is at 2
BRNZ BELONG4 NOP3
RPTB #56 #2 NOP3
MOVE Obuf ag0pi NOP3 //flush out the entry from the processor
NOP NOP3

MOVE ag0pi Dmem 112 NOP3
NOP NOP3
RPTB #55 #2 NOP3
MOVE ag0pi Ibuf0 NOP3 //save the incoming element
NOP NOP3

MOVI Dmem 118 b0000000000000011

NOP NOP3
BRF0 BELONG6 NOP3
NOP NOP3
BR INPU

BELONG4:

RPTB #56 #2 NOP3
MOVE Obuf ag1pi NOP3 //output the element
NOP NOP3

MOVE ag1pi Dmem 112 NOP3
NOP NOP3
RPTB #55 #2 NOP3
MOVE ag1pi Ibuf0 NOP3 //save the incoming element
NOP NOP3

MOVI Dmem 118 b0000000000000010

NOP NOP3
BRF0 BELONG5 NOP3 //if there is no input waiting, branch
NOP NOP3
BR INPU

BELONG5:

RPTB #56 #2 NOP3 //output an entry
MOVE Obuf ag0pi NOP3
NOP NOP3

XOR NULL Dmem 112 Dmem 119 NOP3
NOP
BRZ DUMP

XOR NULL Dmem 118 Dmem 121 NOP3 //check if counter is at 1
BRZ DONE1

MOVI Dmem 118 b0000000000000000
MOVI Dmem 125 b0000000000000001
BRF0 BELONG6 NOP3 //branch if there are no inputs waiting
NOP NOP3
BR INPU

BElong6:

RPTB #56 #2 NOP3 //output an entry
MOVE Obuf ag1pi NOP3
NOP NOP3
XOR NULL Dmem 112 Dmem 119 NOP3
NOP
BRZ DUMP

XOR NULL Dmem 118 Dmem 120 NOP3 //check if counter is at 0
BRZ DONE1

MOVI Dmem 118 b0000000000000001

BRF0 BELONG5 NOP3 //branch if there are no inputs waiting
NOP NOP3
BR INPU

DONE1:
MOVI Dmem 118 b0000000000000000 //reset the counter and flag
MOVI Dmem 125 b0000000000000000
BR INPU

DUMP:
MOVE Obuf Dmem 112 //output the reset signal
MOVI Dmem 118 b0000000000000000 //reset the counter and flag
MOVI Dmem 125 b0000000000000000
BR INPU

DELETE:
RPT #55 NOP3
MOVE NULL Ibuf0 NOP3 //deletes the waiting entry
BR INPU

end
D. TRANSFER.ASSY

This program is loaded into the processors which are acting as transfer lines in the Buffer Mode. This simply will pass entries through if the entry belongs on the certain output.

begin x,y

#output (direction)

INPU:
MOVE Dmem 112 Ibuf0 NOP3 //save the incoming key
NOP NOP3

XOR NULL Dmem 112 Dmem 119 NOP3 //check to see if reset code is in the tag
BRNZ BELONG NOP3

MOVE Dmem 117 Ibuf0 NOP3 //load in secondary reset key
XOR NULL Dmem 117 Dmem 124 NOP3 //check to see if the code is in the tag
BRZ INPU NOP3 //branch back to begining

MOVE Obuf Dmem 112 NOP3 //outputs the reset key
MOVE Obuf Dmem 117 NOP3 //outputs the secondary key
NOP NOP3
BR INPU //branch back to the begining

BELONG:

XOR NULL Dmem 112 Dmem 124 NOP3 //check to see if code is in the key
NOP NOP3
BRZ DELETE //branch to delete
NOP NOP3

MOVE Obuf Dmem 112 NOP3 //outputs the key
NOP NOP3

RPT #55 NOP3 //flush the element from the input to the output
MOVE Obuf Ibuf0 NOP3
BR INPU

DELETE:
RPT #55 NOP3 //delete the incoming elements
MOVE NULL Ibuf0 NOP3
BR INPU

end
E. MERGE.ASSY

This program is used for the processors that merge, it will look at its two input buffers and send out the lower of the two numbers. If a reset command is found on one line, the program will continually send out numbers from the other buffer. If a second reset command is found, a reset command will be sent out to the output buffer, and the processor will wait for new inputs to come to its input buffers.

begin x,y

#output (direction)

INPU0:
MOV Dmem 123 b1111111111111111
//changes the counter to show no reset commands have come
MOVE Dmem 50 Ibuf0 //save the tag of the incoming element
NOP NOP3
XOR NULL Dmem 50 Dmem 119 NOP3 //check to see if reset code is in the tag
BRNZ INPU01 NOP //branches if no reset command is found

MOV Dmem 123 b0000000000000000
//sets the counter to show that a reset command has been received
BR RINPU0 //branches to a specific part of code that

INPU0:
MOVE Dmem 50 Ibuf0 //save the tag of the incoming element
NOP NOP3
XOR NULL Dmem 50 Dmem 119 NOP3 //check to see if reset code is in the tag
BRNZ BELONG0 NOP3

MOV Dmem 123 b0000000000000000
//sets the counter to show that a reset command has been received
MOVE Obuf Dmem 52 //outputs the tag
RPTB #5 #2 NOP3 //outputs the key
MOVE Obuf ag2pi NOP3
NOP NOP3
RPTB #50 #2 NOP3 //outputs the data
MOVE Obuf Ibuf1 NOP3
NOP NOP3
BR RINPU0

INPU01:
RPTB #5 #2 NOP3 //saves the tag
MOVE ag1pi Ibuf0 NOP3
NOP NOP3

INPU1:
MOVE Dmem 52 Ibuf1 //save the tag of the incoming element
NOP NOP3
XOR NULL Dmem 52 Dmem 119 NOP3 //check to see if reset code is in the tag
BRNZ BELONG1 NOP3

MOVI Dmem 123 b0000000000000000 //sets the counter to show that a reset command has been recived
MOVE Obuf Dmem 50 //outputs the tag, key, and data
RPTB #5 #2 NOP3
MOVE Obuf ag1pi NOP3
NOP NOP3

RPTB #50 #2 NOP3
MOVE Obuf Ibuf0 NOP3
NOP NOP3

BR RINPU1

RST:
MOVE Obuf Dmem 52 //outputs the reset command and goes to the top of the program
BR INP0 //to wait for more inputs

RINPU0:
MOVE Dmem 52 Ibuf1 //save the tag of the incoming element
NOP NOP3
XOR NULL Dmem 52 Dmem 119 NOP3 //check to see if reset code is in the tag
BRZ RST NOP3

MOVE Obuf Dmem 52 //outputs the tag, key, and data
RPTB #55 #2 NOP3
MOVE Obuf Ibuf1
NOP NOP3

BR RINPU0

RINPU1:
MOVE Dmem 50 Ibuf0 //save the tag of the incoming element
NOP NOP3
XOR NULL Dmem 50 Dmem 119 NOP3 //check to see if reset code is in the tag
BRZ RST NOP3 //RIPNU0 will only take place after one reset has been found,  
//so this is the second reset, so the next code to be executed is  
//to output a reset command and wait for the next input

MOVE Obuf Dmem 50 //outputs the tag, key, and data  
RPTB #55 #2 NOP3  
MOVE Obuf Ibuf0  
NOP NOP3  
BR RINPU1  
//since there is a reset, the next instructions will be to check to  
//see if there is a reset, then just output from the same input buffer

BELONG0:

RPTB #5 #2 NOP3  
//these two sections of code will save the incoming  
//key then compare it to what is already saved  
MOVE ag1pi Ibuf0 NOP3  
NOP NOP3  
BR COMP

BELONG1:

RPTB #5 #2 NOP3  
MOVE ag2pi Ibuf1 NOP3  
NOP NOP3

COMP: //this is the same compare as from our  
//snakesort code, it compares the two keys to find the lowest

XOR NULL Dmem 113 Dmem 1 NOP3  
//check if the keys are equal one words at a time  
NOP NOP3  
BRNZ CHECK1 NOP3 //if they are not equal check which is bigger  
XOR NULL Dmem 114 Dmem 2 NOP3  
BRNZ CHECK2 NOP3  
XOR NULL Dmem 115 Dmem 3 NOP3  
BRNZ CHECK3 NOP3  
XOR NULL Dmem 116 Dmem 4 NOP3  
BRNZ CHECK4 NOP3  
MAX NULL Dmem 117 Dmem 5 NOP3  
BRNS2 PASS1 NOP1
JouleSort Using a Serial Array of Merge Sorts on ASAP2

NOP
BR PASS0 NOP1

CHECK1: //if it belongs in the list sort, if not pass it
MAX NULL Dmem 113 Dmem 1 NOP3
BRMS2 PASS1 NOP1
NOP
BR PASS0 NOP1 //branch to the sorting code

CHECK2:
MAX NULL Dmem 114 Dmem 2 NOP3
BRMS2 PASS1 NOP1
NOP
BR PASS0 NOP1

CHECK3:
MAX NULL Dmem 115 Dmem 3 NOP3
BRMS2 PASS1 NOP1
NOP
BR PASS0 NOP1

CHECK4:
MAX NULL Dmem 116 Dmem 4 NOP3
BRMS2 PASS1 NOP1
NOP
BR PASS0 NOP1

PASS0:
//this will output from input buffer 0,
//as it will have the lower key if it gets here
NOP
MOVE Obuf Dmem 50 //outputs the tag, key and data from Ibuf0
RPTB #5 #2 NOP3
MOVE Obuf ag1pi NOP3
NOP NOP3
RPTB #50 #2 NOP3
MOVE Obuf Ibuf0 NOP3
NOP NOP3
BR INPU0 NOP1

PASS1: //this will output from input buffer 1

NOP
MOVE Obuf Dmem 52
RPTB #5 #2 NOP3
MOVE Obuf ag2pi NOP3
F. WRITEASSY.M

This program is used to compile all of the different programs into one large input file to program the AsAP2 chip.

```matlab
%writeassy.m
% This script will take multiple AsAP2 programs and write them out according to the mapping you input.
% In the matrix, each entry is XY, where X=in-direction and Y=out-direction
% 0=north, 1=east, 2=south, 3=west, 4=not used
% 5=ns, 6=ne, 7=es,
% function[]=writeassy()
%initialize variable

file1 = fopen('bufmerg.assy','w');
file2 = fopen('buffer.assy','rt');
file3 = fopen('bufmerg.cfg','w');
file4 = fopen('cfg.tem','rt');
file5 = fopen('cfg00.assy','w');
file6 = fopen('merge.assy','rt');
file7 = fopen('trans.assy','rt');
file8 = fopen('bufferchek.assy','rt');
file9 = fopen('transchek.assy','rt');
file10 = fopen('transchek2.assy','rt');
file11 = fopen('cfg2.tem','rt');
str = fread(file2, '*char');
str2 = fread(file4, '*char');
str3 = fread(file6, '*char');
str4 = fread(file7, '*char');
str5 = fread(file8, '*char');
str6 = fread(file9, '*char');
str7 = fread(file10, '*char');
str8 = fread(file11, '*char');
```
yin=6;
xin=0;

yout=5;
xout=12;

mapin0=[2 3 3 3 3 3 3 3 3 3 3 3 3;
        2 3 0 0 0 0 0 0 0 0 0 0;
        2 0 0 0 0 0 0 0 0 0 0 0;
        2 0 0 0 0 0 0 0 0 0 0 0;
        2 0 0 0 0 0 0 0 0 0 0 0;
        2 0 0 0 0 0 0 0 0 0 0 0;
        3 2 2 2 2 2 2 2 2 2 2 2;
        0 2 2 2 2 2 2 2 2 2 2 2;
        0 2 2 2 2 2 2 2 2 2 2 2;
        0 3 2 2 2 2 2 2 2 2 2 2;
        0 3 3 3 3 3 3 3 3 3 3 3];

mapin1=[4 4 4 4 4 4 4 4 4 4 4 4 4;
        4 4 4 4 4 4 4 4 4 4 4 4 4;
        4 4 4 4 4 4 4 4 4 4 4 4 4;
        4 4 4 4 4 4 4 4 4 4 4 4 4;
        4 4 3 3 3 3 3 3 3 3 3 3 3;
        4 4 3 3 3 3 3 3 3 3 3 3 3;
        4 4 3 3 3 3 3 3 3 3 3 3 3;
        4 4 4 4 4 4 4 4 4 4 4 4 4;
        4 4 4 4 4 4 4 4 4 4 4 4 4;
        4 4 4 4 4 4 4 4 4 4 4 4 4;
        4 4 4 4 4 4 4 4 4 4 4 4 4];

mapout=[1 1 7 7 7 7 7 7 7 7 7 7 2;
         6 2 2 2 2 2 2 2 2 2 2 2 2;
         0 2 2 2 2 2 2 2 2 2 2 2 2;
         0 2 2 2 2 2 2 2 2 2 2 2 2;
         0 2 2 2 2 2 2 2 2 2 2 2 3;
         0 1 1 1 1 1 1 1 1 1 1 1 1;
         5 1 1 1 1 1 1 1 1 1 1 1 0;
         2 0 0 0 0 0 0 0 0 0 0 0 0;
         2 0 0 0 0 0 0 0 0 0 0 0 0;
         7 0 0 0 0 0 0 0 0 0 0 0 0;
         1 1 6 6 6 6 6 6 6 6 6 6 0];

whichprog=[0 0 0 0 0 0 0 0 0 0 0 0 1;
           0 1 1 1 1 1 1 1 1 1 1 1 1;
           0 1 1 1 1 1 1 1 1 1 1 1 1;]
\begin{verbatim}

[M N] = size(mapin0);

for n=0:1:M-1
    for m=0:1:N-1
        if mapin0(n+1,m+1)==0
            dirin0='north';
        elseif mapin0(n+1,m+1)==1
            dirin0='east';
        elseif mapin0(n+1,m+1)==2
            dirin0='south';
        elseif mapin0(n+1,m+1)==3
            dirin0='west';
        end

        if mapin1(n+1,m+1)==0
            dirin1='north';
        elseif mapin1(n+1,m+1)==1
            dirin1='east';
        elseif mapin1(n+1,m+1)==2
            dirin1='south';
        elseif mapin1(n+1,m+1)==3
            dirin1='west';
        end

        if mapout(n+1,m+1)==0
            dirout='north';
        elseif mapout(n+1,m+1)==1
            dirout='east';
        elseif mapout(n+1,m+1)==2
            dirout='south';
        elseif mapout(n+1,m+1)==3
            dirout='west';
        elseif mapout(n+1,m+1)==5
            dirout='ns';
        elseif mapout(n+1,m+1)==6
            dirout='nw';
        else
            dirout='invalid';
        end

end

end
\end{verbatim}
dirout='ne';
elseif mapout(n+1,m+1)==7
  dirout='se';
end

fprintf(file3,'#%d,%d ext_clk b1\n%d,%d ibuf0 %s\n',m,n,m,n,dirin0);
if mapin1(n+1,m+1)==4
  fprintf(file3, '\n');
else
  fprintf(file3,'%d,%d ibuf1 %s\n\n',m,n,dirin1);
end

fprintf(file1,'\n\nbegin %d,%d',m,n);
if m==xin & n==yin
  fprintf(file1, ', chipin %s',dirin0);
end
if m==xout & n==yout
  fprintf(file1, ', chipout %s', dirout);
end
fprintf(file1,')

#output %s
', dirout);
fprintf(file5,'\n\nbegin %d,%d

',m,n);
if n==5 & m==0
  fprintf(file5,'MOVI Dmem 124 b0000000000010000
',m,n);
else if n==7 & m==0
  fprintf(file5,'MOVI Dmem 124 b0000000000010000
',m,n);
end

fprintf(file1,'%s',str7);
else if n==7 & m==0
  fprintf(file5,'MOVI Dmem 124 b0000000000010000
',m,n);
else if n==7 & m==0
  fprintf(file5,'MOVI Dmem 124 b0000000000010000
',m,n);
end

fprintf(file1,'\n\nbegin %d,%d\n
',m,n);
if n==5 & m==0
  fprintf(file5,'MOVI Dmem 124 b0000000000010000
',m,n);
else if n==7 & m==0
  fprintf(file5,'MOVI Dmem 124 b0000000000010000
',m,n);
end

fprintf(file1,'%s',str7);
fprintf(file1,'%s',str7);
else n==0 & m==0
    fprintf(file5,'MOVI Dmem 124 b0000000000000001\n',m,n);
    fprintf(file1,'%s',str6);
else n==0 & m==3
    fprintf(file5,'MOVI Dmem 124 b0000000000000001\n',m,n);
    fprintf(file1,'%s',str6);
else n==0 & m==4
    fprintf(file5,'MOVI Dmem 124 b0000000000000001\n',m,n);
    fprintf(file1,'%s',str6);
else n==0 & m==5
    fprintf(file5,'MOVI Dmem 124 b0000000000000001\n',m,n);
    fprintf(file1,'%s',str6);
else n==0 & m==6
    fprintf(file5,'MOVI Dmem 124 b0000000000000001\n',m,n);
    fprintf(file1,'%s',str6);
else n==0 & m==7
    fprintf(file5,'MOVI Dmem 124 b0000000000000001\n',m,n);
    fprintf(file1,'%s',str6);
else n==0 & m==8
    fprintf(file5,'MOVI Dmem 124 b0000000000000001\n',m,n);
    fprintf(file1,'%s',str6);
else n==0 & m==9
    fprintf(file5,'MOVI Dmem 124 b0000000000000001\n',m,n);
    fprintf(file1,'%s',str6);
else n==0 & m==10
    fprintf(file5,'MOVI Dmem 124 b0000000000000001\n',m,n);
    fprintf(file1,'%s',str6);
else n==0 & m==11
    fprintf(file5,'MOVI Dmem 124 b0000000000000001\n',m,n);
    fprintf(file1,'%s',str6);
else n==10 & m==0
    fprintf(file5,'MOVI Dmem 124 b0000000000000001\n',m,n);
    fprintf(file1,'%s',str6);
else n==10 & m==3
    fprintf(file5,'MOVI Dmem 124 b0000000000000001\n',m,n);
    fprintf(file1,'%s',str6);
else n==10 & m==4
    fprintf(file5,'MOVI Dmem 124 b0000000000000001\n',m,n);
    fprintf(file1,'%s',str6);
else n==10 & m==5
    fprintf(file5,'MOVI Dmem 124 b0000000000000001\n',m,n);
    fprintf(file1,'%s',str6);
else n==10 & m==6
    fprintf(file5,'MOVI Dmem 124 b0000000000000001\n',m,n);
JouleSort Using a Serial Array of Merge Sorts on ASAP2

```c
fprintf(file1,'%s',str6);
elseif n==10 & m==7
fprintf(file5,'MOVI Dmem 124 b0000000000010110\n',m,n);
fprintf(file1,'%s',str6);
elseif n==10 & m==8
fprintf(file5,'MOVI Dmem 124 b0000000000010111\n',m,n);
fprintf(file1,'%s',str6);
elseif n==10 & m==9
fprintf(file5,'MOVI Dmem 124 b0000000000011000\n',m,n);
fprintf(file1,'%s',str6);
elseif n==10 & m==10
fprintf(file5,'MOVI Dmem 124 b0000000000011001\n',m,n);
fprintf(file1,'%s',str6);
elseif n==10 & m==11
fprintf(file5,'MOVI Dmem 124 b0000000000011010\n',m,n);
fprintf(file1,'%s',str6);
elseif n==1 & m==1
fprintf(file5,'MOVI Dmem 124 b0000000000000001\n',m,n);
fprintf(file1,'%s',str5);
elseif n==1 & m==2
fprintf(file5,'MOVI Dmem 124 b0000000000000010\n',m,n);
fprintf(file1,'%s',str5);
elseif n==1 & m==3
fprintf(file5,'MOVI Dmem 124 b0000000000000011\n',m,n);
fprintf(file1,'%s',str5);
elseif n==1 & m==4
fprintf(file5,'MOVI Dmem 124 b0000000000000100\n',m,n);
fprintf(file1,'%s',str5);
elseif n==1 & m==5
fprintf(file5,'MOVI Dmem 124 b0000000000000101\n',m,n);
fprintf(file1,'%s',str5);
elseif n==1 & m==6
fprintf(file5,'MOVI Dmem 124 b0000000000000110\n',m,n);
fprintf(file1,'%s',str5);
elseif n==1 & m==7
fprintf(file5,'MOVI Dmem 124 b0000000000000111\n',m,n);
fprintf(file1,'%s',str5);
elseif n==1 & m==8
fprintf(file5,'MOVI Dmem 124 b0000000000001000\n',m,n);
fprintf(file1,'%s',str5);
elseif n==1 & m==9
fprintf(file5,'MOVI Dmem 124 b0000000000001001\n',m,n);
fprintf(file1,'%s',str5);
elseif n==1 & m==10
fprintf(file5,'MOVI Dmem 124 b0000000000001010\n',m,n);
fprintf(file1,'%s',str5);
```
fprintf(file1,'%s',str5);
elseif n==1 & m==11
fprintf(file5,'MOVI Dmem 124 b0000000000001011\n',m,n);
fprintf(file1,'%s',str5);
elseif n==0 & m==12
fprintf(file5,'MOVI Dmem 124 b0000000000001100\n',m,n);
fprintf(file1,'%s',str5);
elseif n==9 & m==1
fprintf(file5,'MOVI Dmem 124 b0000000000010001\n',m,n);
fprintf(file1,'%s',str5);
elseif n==9 & m==2
fprintf(file5,'MOVI Dmem 124 b0000000000010010\n',m,n);
fprintf(file1,'%s',str5);
elseif n==9 & m==3
fprintf(file5,'MOVI Dmem 124 b0000000000010011\n',m,n);
fprintf(file1,'%s',str5);
elseif n==9 & m==4
fprintf(file5,'MOVI Dmem 124 b0000000000010100\n',m,n);
fprintf(file1,'%s',str5);
elseif n==9 & m==5
fprintf(file5,'MOVI Dmem 124 b0000000000010101\n',m,n);
fprintf(file1,'%s',str5);
elseif n==9 & m==6
fprintf(file5,'MOVI Dmem 124 b0000000000010110\n',m,n);
fprintf(file1,'%s',str5);
elseif n==9 & m==7
fprintf(file5,'MOVI Dmem 124 b0000000000010111\n',m,n);
fprintf(file1,'%s',str5);
elseif n==9 & m==8
fprintf(file5,'MOVI Dmem 124 b0000000000011000\n',m,n);
fprintf(file1,'%s',str5);
elseif n==9 & m==9
fprintf(file5,'MOVI Dmem 124 b0000000000011001\n',m,n);
fprintf(file1,'%s',str5);
elseif n==9 & m==10
fprintf(file5,'MOVI Dmem 124 b0000000000011010\n',m,n);
fprintf(file1,'%s',str5);
elseif n==9 & m==11
fprintf(file5,'MOVI Dmem 124 b0000000000011011\n',m,n);
fprintf(file1,'%s',str5);
elseif n==10 & m==12
fprintf(file5,'MOVI Dmem 124 b0000000000011100\n',m,n);
fprintf(file1,'%s',str5);
else if whichprog(n+1,m+1)==0
fprintf(file1,'%s',str4);
elseif whichprog(n+1,m+1)==1
fprintf(file1,'%s',str);
else whichprog(n+1,m+1)==2
fprintf(file1,'%s',str3);
end

if whichprog(n+1,m+1)==2
fprintf(file5,'%s',str8);
else
fprintf(file5,'%s',str2);
end
end
end
end

message = ferror(file1)
fclose('all');
end

G. TIMETOSORT.M
This program is used simulate the time needed to sort 1 GB of entries in the AsAP2 chip using the SAMSort.

%timetosort.m
%
%Simulates a merge between a variable
%amount of lists, from 2-24, of one
%gigabyte of entries and saves the
%quickest method, along with the times of every other method tried.
%
%The simulation will not count a partial merge,
%just as our program will not do a partial merge, it will leave
%the remaining lists to be merged at the next pass through.
%
%The simulation will keep the number of lists merged
A. and L. Stillmaker

%the same for every pass, until it comes to the last pass.
%There the simulation will merge the remaining lists,
%and output how many lists were remaining for the last merge.

function [TimeToSortFinal, NumberOfListsMergedFinal, SortTime] = timetosort()

SortTime = 0; % Initialization of variables
z = 1;
y = 1;
NumberOfListsMergedFinal = zeros(1,15);
NumberOfListsMerged = zeros(1,15);
TimeToSortFinal = inf;
TimeToSort = 0;
LeftToMerge = 30164; % There are 30164 lists to sort in one gigabyte of data

for x1 = 2:1:24 % The loop that will take the simulation from 2 to 24
    y = x1;
    NumberOfListsMerged = zeros(1,15);
    % Clears the variables for the start of another loop run
    TimeToSort = 0;
    LeftToMerge = 30164;
    NumberOfListsMerged(1) = x1;
    % Saves the current number as the number of lists merged for the first pass
    TimeToSort = TimeToSort + floor(LeftToMerge / x1) * (50 + 34.375 + 156000) + 9893611 * 1000;
    % Adds the correct amount of time for the first pass
    LeftToMerge = floor(LeftToMerge/x1) + rem(LeftToMerge,x1);
    % Lowers the amount of lists left to merge, leaving the remainder lists unmerged
    if (LeftToMerge > 1) % Will continue only if there are lists left to merge
        x2 = y; % Sets the second pass number of merges to the same as the previous pass
        NumberOfListsMerged(2) = x2;
        % Saves the current number as the number of lists merged for the second pass
        TimeToSort = TimeToSort + floor(LeftToMerge / x2) * (50 + 34.375 + 156000) + 9893611 * 1000;
        LeftToMerge = floor(LeftToMerge/x2) + rem(LeftToMerge,x2);
        if (LeftToMerge > 1)
            x3 = y;
            NumberOfListsMerged(3) = x3;
            TimeToSort = TimeToSort + floor(LeftToMerge / x3) * (50 + 34.375 + 156000) + 9893611 * 1000;
            LeftToMerge = floor(LeftToMerge/x3) + rem(LeftToMerge,x3);
            if (LeftToMerge > 1)
                yi = y;
        % This for loop will check to see if the number of lists left to
        % merge is reducable to 1 by any number from 2-24,
for yt = 2:1:y % if it is not reducible to 1, then the previous number of merges is used
  if (LeftToMerge/yt == 1)
    yi=yt;
  end
  end
  x4 = yi;
  NumberOfListsMerged(4) = x4;
  TimeToSort = TimeToSort + floor(LeftToMerge / x4) * (50 + 34.375 + 156000) + 
              LeftToMerge = floor(LeftToMerge/x4) + rem(LeftToMerge,x4);
  if (LeftToMerge > 1)
    yi=y;
  for yt = 2:1:y
    if (LeftToMerge/yt == 1)
      yi=yt;
    end
    end
    x5 = yi;
    NumberOfListsMerged(5) = x5;
    TimeToSort = TimeToSort + floor(LeftToMerge / x5) * (50 + 34.375 + 156000) + 
                LeftToMerge = floor(LeftToMerge/x5) + rem(LeftToMerge,x5);
    if (LeftToMerge > 1)
      yi=y;
    for yt = 2:1:y
      if (LeftToMerge/yt == 1)
        yi=yt;
      end
      end
      x6 = yi;
      NumberOfListsMerged(6) = x6;
      TimeToSort = TimeToSort + floor(LeftToMerge / x6) * (50 + 34.375 + 156000) + 
                  LeftToMerge = floor(LeftToMerge/x6) + rem(LeftToMerge,x6);
      if (LeftToMerge > 1)
        yi=y;
      for yt = 2:1:y
        if (LeftToMerge/yt == 1)
          yi=yt;
        end
        end
        x7 = yi;
        NumberOfListsMerged(7) = x7;
        TimeToSort = TimeToSort + floor(LeftToMerge / x7) * (50 + 34.375 + 156000) + 
                    LeftToMerge = floor(LeftToMerge/x7) + rem(LeftToMerge,x7);
        if (LeftToMerge > 1)
          yi=y;
        for yt = 2:1:y
          if (LeftToMerge/yt == 1)
\[ yi = yt; \]
\[ end \]
\[ end \]
\[ x8 = yi; \]
\[ NumberOfListsMerged(8) = x8; \]
\[ TimeToSort = TimeToSort + \text{floor}(LeftToMerge / x8) \times (50 + 34.375); \]
\[ LeftToMerge = \text{floor}(LeftToMerge / x8) + \text{rem}(LeftToMerge, x8); \]
\[ if (LeftToMerge > 1) \]
\[ yi = y; \]
\[ for yt = 2:1:y \]
\[ if (LeftToMerge / yt == 1) \]
\[ yi = yt; \]
\[ end \]
\[ end \]
\[ x9 = yi; \]
\[ NumberOfListsMerged(9) = x9; \]
\[ TimeToSort = TimeToSort + \text{floor}(LeftToMerge / x9) \times (50 + 34.375); \]
\[ LeftToMerge = \text{floor}(LeftToMerge / x9) + \text{rem}(LeftToMerge, x9); \]
\[ if (LeftToMerge > 1) \]
\[ yi = y; \]
\[ for yt = 2:1:y \]
\[ if (LeftToMerge / yt == 1) \]
\[ yi = yt; \]
\[ end \]
\[ end \]
\[ x10 = yi; \]
\[ NumberOfListsMerged(10) = x10; \]
\[ TimeToSort = TimeToSort + \text{floor}(LeftToMerge / x10) \times (50 + 34.375); \]
\[ LeftToMerge = \text{floor}(LeftToMerge / x10) + \text{rem}(LeftToMerge, x10); \]
\[ if (LeftToMerge > 1) \]
\[ yi = y; \]
\[ for yt = 2:1:y \]
\[ if (LeftToMerge / yt == 1) \]
\[ yi = yt; \]
\[ end \]
\[ end \]
\[ x11 = yi; \]
\[ NumberOfListsMerged(11) = x11; \]
\[ TimeToSort = TimeToSort + \text{floor}(LeftToMerge / x11) \times (50 + 34.375); \]
\[ LeftToMerge = \text{floor}(LeftToMerge / x11) + \text{rem}(LeftToMerge, x11); \]
\[ if (LeftToMerge > 1) \]
\[ yi = y; \]
\[ for yt = 2:1:y \]
\[ if (LeftToMerge / yt == 1) \]
\[ yi = yt; \]
\[ end \]
end

x12 = yi;
NumberOfListsMerged(12) = x12;
TimeToSort = TimeToSort + floor(LeftToMerge / x12) * 
LeftToMerge = floor(LeftToMerge/x12) + rem(LeftToMerge, x12)
if (LeftToMerge > 1)

yi=y;
for yt = 2:1:y
if (LeftToMerge/yt == 1)
yi=yt;
end
end

x13 = yi;
NumberOfListsMerged(13) = x13;
TimeToSort = TimeToSort + floor(LeftToMerge / x13) * 
LeftToMerge = floor(LeftToMerge/x13) + rem(LeftToMerge, x13)
if (LeftToMerge > 1)

yi=y;
for yt = 2:1:y
if (LeftToMerge/yt == 1)
yi=yt;
end
end

x14 = yi;
NumberOfListsMerged(14) = x14;
TimeToSort = TimeToSort + floor(LeftToMerge / x14) * 
LeftToMerge = floor(LeftToMerge/x14) + rem(LeftToMerge, x14)
if (LeftToMerge > 1)

yi=y;
for yt = 2:1:y
if (LeftToMerge/yt == 1)
yi=yt;
end
end

x15 = yi;
NumberOfListsMerged(15) = x15;
TimeToSort = TimeToSort + floor(LeftToMerge / x15) * 
LeftToMerge = floor(LeftToMerge/x15) + rem(LeftToMerge, x15)
if (LeftToMerge ==1) % This will
if (TimeToSort < TimeToSortFinal) % This will
TimeToSortFinal = TimeToSort; % as separate
NumberOfListsMergedFinal = NumberOfListsMerged;
end
SortTime(z) = TimeToSort; % Saves time
z = z + 1;
LeftToMerge=0; % Resets left to merge variable
end
end
if (LeftToMerge ==1)
  if (TimeToSort < TimeToSortFinal)
      TimeToSortFinal = TimeToSort;
      NumberOfListsMergedFinal = NumberOfListsMerged;
  end
  SortTime(z) = TimeToSort;
  z = z + 1;
  NumberOfListsMerged(14) = 0;
  LeftToMerge=0;
end
end
if (LeftToMerge ==1)
  if (TimeToSort < TimeToSortFinal)
      TimeToSortFinal = TimeToSort;
      NumberOfListsMergedFinal = NumberOfListsMerged;
  end
  SortTime(z) = TimeToSort;
  z = z + 1;
  NumberOfListsMerged(13) = 0;
  LeftToMerge=0;
end
end
if (LeftToMerge ==1)
  if (TimeToSort < TimeToSortFinal)
      TimeToSortFinal = TimeToSort;
      NumberOfListsMergedFinal = NumberOfListsMerged;
  end
  SortTime(z) = TimeToSort;
  z = z + 1;
  NumberOfListsMerged(12) = 0;
  LeftToMerge=0;
end
end
if (LeftToMerge ==1)
  if (TimeToSort < TimeToSortFinal)
      TimeToSortFinal = TimeToSort;
      NumberOfListsMergedFinal = NumberOfListsMerged;
  end
  SortTime(z) = TimeToSort;
  z = z + 1;
  NumberOfListsMerged(11) = 0;
  LeftToMerge=0;
end
if (LeftToMerge == 1)
    if (TimeToSort < TimeToSortFinal)
        TimeToSortFinal = TimeToSort;
        NumberOfListsMergedFinal = NumberOfListsMerged;
    end
    SortTime(z) = TimeToSort;
    z = z + 1;
    NumberOfListsMerged(10) = 0;
    LeftToMerge = 0;
end
end
if (LeftToMerge == 1)
    if (TimeToSort < TimeToSortFinal)
        TimeToSortFinal = TimeToSort;
        NumberOfListsMergedFinal = NumberOfListsMerged;
    end
    SortTime(z) = TimeToSort;
    z = z + 1;
    NumberOfListsMerged(9) = 0;
    LeftToMerge = 0;
end
end
if (LeftToMerge == 1)
    if (TimeToSort < TimeToSortFinal)
        TimeToSortFinal = TimeToSort;
        NumberOfListsMergedFinal = NumberOfListsMerged;
    end
    SortTime(z) = TimeToSort;
    z = z + 1;
    NumberOfListsMerged(8) = 0;
    LeftToMerge = 0;
end
end
if (LeftToMerge == 1)
    if (TimeToSort < TimeToSortFinal)
        TimeToSortFinal = TimeToSort;
        NumberOfListsMergedFinal = NumberOfListsMerged;
    end
    SortTime(z) = TimeToSort;
    z = z + 1;
    NumberOfListsMerged(7) = 0;
    LeftToMerge = 0;
end
end
if (LeftToMerge == 1)
if (TimeToSort < TimeToSortFinal)
    TimeToSortFinal = TimeToSort;
    NumberOfListsMergedFinal = NumberOfListsMerged;
end
    SortTime(z) = TimeToSort;
    z = z + 1;
    NumberOfListsMerged(6) = 0;
    LeftToMerge=0;
end
end
if (LeftToMerge ==1)
    if (TimeToSort < TimeToSortFinal)
        TimeToSortFinal = TimeToSort;
        NumberOfListsMergedFinal = NumberOfListsMerged;
    end
    SortTime(z) = TimeToSort;
    z = z + 1;
    NumberOfListsMerged(5) = 0;
    LeftToMerge=0;
end
end
if (LeftToMerge ==1)
    if (TimeToSort < TimeToSortFinal)
        TimeToSortFinal = TimeToSort;
        NumberOfListsMergedFinal = NumberOfListsMerged;
    end
    SortTime(z) = TimeToSort;
    z = z + 1;
    NumberOfListsMerged(4) = 0;
    LeftToMerge=0;
end
end
if (LeftToMerge ==1)
    if (TimeToSort < TimeToSortFinal)
        TimeToSortFinal = TimeToSort;
        NumberOfListsMergedFinal = NumberOfListsMerged;
    end
    SortTime(z) = TimeToSort;
    z = z + 1;
    NumberOfListsMerged(3) = 0;
    LeftToMerge=0;
end
end
if (LeftToMerge ==1)
    if (TimeToSort < TimeToSortFinal)
        TimeToSortFinal = TimeToSort;
NumberOfListsMergedFinal = NumberOfListsMerged;
end
SortTime(z) = TimeToSort;
z = z + 1;
NumberOfListsMerged(2) = 0;
LeftToMerge = 0;
end
end
save timtos . \% saves all of the variables into a MatLab readable file
end