ECE 4750 Computer Architecture

Topic 5: Integrating Processors and Memories

http://www.csl.cornell.edu/courses/ece4750
School of Electrical and Computer Engineering
Cornell University

revision: 2024-11-04-14-38

List of Problems

Problem 1. Short Answer

Part 1.A Integrating Processors and Memories

For this problem, we will use the fully bypassed five-stage TinyRV1 processor discussed in lecture, composed with an L1 data cache with the following configuration:

- L1 Total Capacity : 256B cache
- L1 Cache Line Size : 16B cache lines
- L1 Num Cache Lines : 16
- L1 Hit Latency : 1 cycle
- L1 Style : fully associative
- L1 Replacement Policy : LRU
- L1 Write Policy : write-back, write-allocate
- L1 Miss Penalty : 2 cycles

The details of the L1 miss handling (e.g., the exact FSM states) are not important; just the fact that all misses (both read and write misses) always have a two-cycle miss penalty. The L1 data cache initially starts with every cache line invalidated. Consider the following C-code and corresponding static instruction sequence. Assume `x2` is initially `0x1000`, `x3` is initially `0x2000`, and `x4` is initially 64.

```
for ( int i = 0; i < n; i++ ) {
  int temp = A[i] + 1;
  A[i] = temp;
  B[i] = temp;
}
```

```
loop:
  addi x4, x4, -1
  lw   x1, 0(x2)
  addi x1, x1, 1
  sw   x1, 0(x2)
  sw   x1, 0(x3)
  addi x2, x2, 4
  addi x3, x3, 4
  bne  x4, x0, loop
```

Draw a pipeline diagram to illustrate how this assembly sequence executes on the processor and data cache composition. You should correctly capture stalls due to data cache misses, as well as the impact of resolving any hazards. You can use microarchitectural dependency arrows to illustrate how data is transferred between instructions, although this is not required. If you do draw arrows, then they must be correct. You will likely need to include more than one iteration to understand the overall execution; so two different pipeline diagrams are provided for you to use if you would like. **Estimate the overall execution time (in cycles) of this loop. You must show your work and explain your answer.**

Problem 2. TinyRV1 Instruction Cache

In this problem, we will be exploring adding an instruction cache to an TinyRV1 FSM processor. We will be using the TinyRV1 assembly program shown in Figure 1. The first column shows the instruction address for each instruction. Note that these addresses are byte addresses. The value of `x1` is initially 64, meaning that there are 64 iterations in the loop. In this problem, we will be considering the execution of this loop with a direct-mapped instruction cache microarchitecture with eight 16 B cache lines. This means each cache line can hold four instructions and the bottom four bits of an instruction address are the block offset. Hint: The first instruction in Figure 1 (i.e., `addi x1, x1, -1`), is in the middle of a cache line.

Part 2.A Categorizing Cache Misses

Create a table like the one shown in Figure 1. In the appropriate column, write compulsory, conflict, or capacity next to each instruction which misses in the instruction cache to indicate the type of instruction cache misses that occur in the first and second iteration of the loop. Assume that the instruction cache is initially completely empty.

Part 2.B Average Memory Access Latency

Calculate the instruction cache miss rate for 64 iterations of the loop. Calculate the average instruction cache memory access latency in cycles for 64 iterations of the loop. Assume the hit time is one cycle and that the miss penalty is 15 cycles. You must show your work, especially the various components of the average memory access latency. **Remark on which kind of miss is dominating the average memory access latency.**

Part 2.C Set-Associativity

Qualitatively, predict how the cache performance would change if we replace the eight-entry, direct-mapped cache with an eight-entry, two-way, set-associative cache. Both caches have a one-cycle hit latency. **What kind of misses would be present with this kind of cache microarchitecture?**

Addr	Instruction	Iteration 1	Iteration 2
	`loop:`		
0x108	`addi x1, x1, -1`		
0x10c	`addi x2, x2, 1`		
0x110	`jal x0, foo`		
	`...`		
	`foo:`		
0x218	`addi x6, x6, 1`		
0x21c	`bne x1, x0, loop`		

Figure 1: Example TinyRV1 Assembly Loop

Problem 3. Software Prefetching

In this problem, you will explore the potential of software prefetching. Prefetching is a technique where we request data from main memory well before we actually do the load/store that needs this data. Our specific implementation of software prefetching adds a new `pfetch` instruction. This instruction takes a register value and an offset as its operands, and it serves as a hint to the hardware to prefetch the cache line with the given effective address.

`pfetch` instructions go down the pipeline just like normal loads and stores; they enter the memory system in the M stage. We will now have three kinds of memory requests: reads, writes, and prefetches. A prefetch memory request bypasses the L1 data cache and goes directly to a dedicated *prefetch buffer* (see Figure 2). The prefetch buffer handles making requests to main memory to retrieve cache lines ahead of time and then storing these cache lines within the prefetch buffer. For this problem you can ignore any impact prefetching has on memory bandwidth pressure (i.e., ignore the fact that prefetching might get in the way of normal eviction and refill requests). You can also assume the prefetch buffer is fully associative and quite large (i.e., we can ignore any conflict or capacity issues in the prefetch buffer). On an L1 miss, we will first check the prefetch buffer. If the line we need is already in the prefetch buffer, then we can immediately return this line to the L1 cache reducing the miss penalty to a single cycle. If the line we need is not in the prefetch buffer, then we will need to go to main memory to retrieve it.

We will examine the performance for a memcopy function written in assembly. Figure 3 shows the function without software prefetching, and Figure 4 shows the function with software prefetching. This function simply copies elements from an input array to an output array. Assume the following initial register values: `r4` initially holds the pointer to the input array; `r5` initially holds the pointer to the output array; `r6` is the size of both arrays. Assume that `r6` is initially 64 (i.e., the loop executes 64 times).

For this problem, you should assume we have a very large, fully-associative data cache such that there are no capacity nor conflict misses. The data cache uses 16B cache lines and is initially empty. The prefetch buffer is also initially empty. You should also assume that both the input and output array data structures are cache-line aligned. Cache-line aligned means that the first element of the array is at the very beginning of a cache line. Assume a constant cycle time of 1τ for both microarchitectures.

The data cache has a hit latency of one cycle. The main memory has an access latency of three cycles. When we add the prefetch buffer, it will add an extra cycle of latency to the miss penalty for the L1 data cache. The processor should stall for both read and write misses. As described in lecture, assume that load/store instructions stall in the M stage on a cache miss. **The most important difference between a `pfetch` instruction and a normal load/store, is that a `pfetch` instruction never stalls the processor pipeline on a miss!** This allows the `pfetch` instructions to generate prefetch memory requests, and the processor can then continue executing regular instructions.

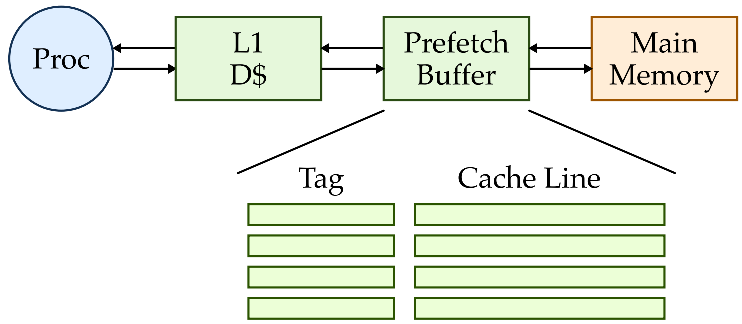

Figure 2: Integrating Prefetch Buffer into Memory System

```
loop:
  lw   x12, 0(x4)
  sw   x12, 0(x5)
  addi x4, x4, 4
  addi x5, x5, 4
  addi x6, x6, -1
  bne  x6, x0, loop
```

Figure 3: Memcopy without Software Prefetching

```
loop:
  pfetch 16(x4)
  pfetch 16(x5)
  lw     x12, 0(x4)
  sw     x12, 0(x5)
  addiu  x4, x4, 4
  addiu  x5, x5, 4
  addiu  x6, x6, -1
  bne    x6, x0, loop
```

Figure 4: Memcopy with Software Prefetching

Part	Software Prefetching	Instructions / Program	Avg Cycles / Instruction	Time (τ) / Cycle	Time (τ) / Program
3.A	**no**			1	
3.B	**yes**			1	

Figure 5: Processor Performance for Memory Copy with/without Software Prefetching

Part	Software Prefetching	L1 Hit Latency (cycles)	L1 Miss Rate	L1 Miss Penalty (cycles)	AMAL (cycles)
3.A	**no**	1			
3.B	**yes**	1			

Figure 6: Memory Performance for Memory Copy with/without Software Prefetching
Do not include prefetch requests in your calculations. Calculate average miss penalty for given transaction sequence.

Part 3.A Performance without Software Prefetching

In this part, we will calculate the execution time for the assembly sequence given above executing without software prefetching. You should assume the processor is identical to the fully bypassed five-stage PARCv1 pipeline discussed in lecture.

Draw two pipeline diagrams illustrating how the first and second iteration of the loop execute on this microarchitecture. Use microarchitectural dependency arrows to illustrate how data is transferred between instructions. You may want to include the first instruction of the next iteration to illustrate the impact of the branch resolution latency. **Based on these pipeline diagrams estimate both the execution time in units of τ and the average memory access latency in cycles. Fill in the appropriate rows of the tables in Figures 5 and 6. You must show your work.**

Part 3.B Performance with Software Prefetching

In this part, we will calculate the execution time for the assembly sequence given above executing with software prefetching. You should assume the processor is identical to the fully bypassed five-stage PARCv1 pipeline discussed in lecture except with support for `pfetch` instructions. Remeber that checking the prefetch buffer adds an extra cycle of latency. So if the cache line is in prefetch buffer then the miss penalty will be one cycle, and if the cache line is not in the prefetch buffer then the miss penalty will be four cycles.

Draw two pipeline diagrams illustrating how the first and second iteration of the loop execute on this microarchitecture. Use microarchitectural dependency arrows to illustrate how data is transferred between instructions. You may want to include the first instruction of the next iteration to illustrate the impact of the branch resolution latency. You will not really be able to estimate the performance from just these two pipeline diagrams. **Draw one more pipeline diagram for a later iteration which you think will help you estimate the performance of the entire loop. Based on these pipeline diagrams estimate both the execution time in units of τ and the average memory access latency in cycles. Fill in the appropriate rows of the tables in Figures 5 and 6. You must show your work.** Read/write requests that miss in the L1 data cache but hit in the prefetch buffer still count as misses; they just have a reduced miss penalty. Since the miss penalty will vary from one cycle when the desired line is in the prefetch buffer to four cycles with the desired line is not in the prefetch buffer, report the miss penalty as the average miss penalty for the given assembly sequence. Do not count prefetch requests as memory requests for the purpose of calculating the miss rate and average miss penalty.

Problem 4. Array vs. List Cache Behavior

In this problem, you will explore the cache behavior for a basic operation on two common software data structures. Figure 7 illustrates a basic linear array and a doubly linked list. Each node in the doubly linked list has a 4B value field, a 4B pointer that points to the next node in the list, and a 4B pointer that points to the previous node in the list. The previous pointer for the head node is defined to be zero, and the next pointer for the tail node is defined to be zero. For this problem, you should assume that both data structures contain 64 4B values (i.e., the array is 64 elements long, and the linked list contains 64 nodes).

We wish to explore the performance of reversing the values in each data structure. Conventional wisdom in a basic computer science course on data structures might suggest that the time to complete this operation on both the array and linked list is $O(n)$ where n is the number of elements in the data structure. "Big-O" notation is useful when analyzing asymptotic behavior as n grows very large, but it abstracts many important "constant factors" that can dominate the performance for reasonable sized data structures on real architectures.

For this problem, you should assume we have a very large, fully-associative data cache such that there are no capacity nor conflict misses. Assume a write-back, write-allocate cache. The data cache uses 16B cache lines and is initially empty. All data cache accesses will result in either a hit or a compulsory miss. You should also assume that the array data structure is cache-line aligned, each node in the linked list is also cache-line aligned, and there is only one linked list node per cache line. Cache-line aligned means that the first element of the array is at the very beginning of a cache line, and that value field for each linked list node is also at the very beginning of a cache line.

The data cache has a hit latency of one cycle and a miss penalty of four cycles. This means if an instruction stalls in M waiting for a cache miss, it will remain in the M stage for a total of five cycles (one cycle for the hit latency and four cycles for the miss penalty). The processor should stall for both read and write misses.

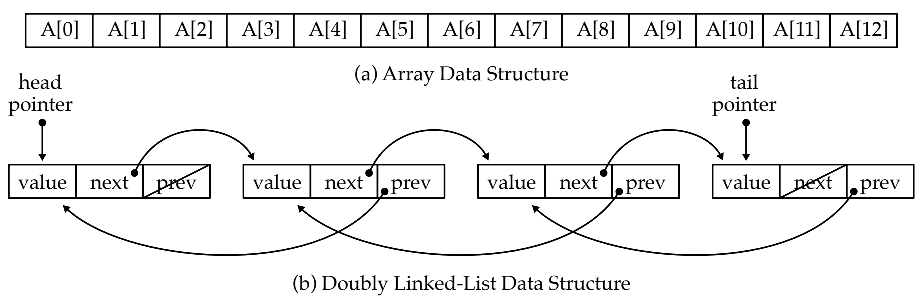

Figure 7: Linear Array and Doubly Linked-List Data Structures

Part	Number of Instructions	CPI	Execution Time (cyc)	CPI Breakdown: Useful Work	RAW Stalls	Control Squashes	Memory Stalls
Part 4.A							
Part 4.B							

Figure 8: Execution Time for Reverse Operation on Array and Linked List Data Structures

Part 4.A Analyzing Performance of an Array Data Structure

Figure 9 shows C-code which implements the reverse operation for an array data structure with an even number of elements. Figure 10 shows corresponding TinyRV1 assembly code for the function. Recall that arguments are stored in `x10` and `x11`. Note that we are organizing the loop a bit differently than some of the other loops we have studied in this course to better match the assembly code and also to better match the code used for the linked list. We use a few setup instructions to create a pointer to the last element in the array. We then iteratively move the forward and reverse pointers, swapping the data as we go along.

Draw a pipeline diagram illustrating at least one iteration of the loop show in Figure 10. Assume the cannonical five-stage fully bypassed TinyRV1 processor. You do not need to show the four setup instructions used to calculate the initial reverse pointer. You should draw as many iterations as you need in order to determine the steady-state behavior of the loop. Carefully consider which memory accesses hit or miss in the data cache. **Use your pipeline diagram to estimate the overall execution time in cycles for this operation. Fill in the corresponding row of Figure 8**. You will need to decompose the CPI into its various components based on what stalls and/or squashes occur in the execution. **You must show your work.**

```
void array_reverse( int* A, int n ) {

  # Code only works for even n
  assert( n % 2 == 0 )

  int* fwd_ptr = &A[0];
  int* rev_ptr = &A[n-1];
  int* rev_ptr_last = 0;

  do {
    # Swap values at fwd and rev pointers
    int temp = *fwd_ptr;
    *fwd_ptr = *rev_ptr;
    *rev_ptr = temp;

    # Save rev pointer for exit condition check
    rev_ptr_last = rev_ptr;

    # Update fwd and rev pointers
    fwd_ptr++;
    rev_ptr--;
  }
  while ( fwd_ptr != rev_ptr_last );

}
```

Figure 9: C-Code for Reverse on Array Data Structure

```
  # x10: A
  # x11: n

  addi x11, x11, -1
  addi x5,  x0, 4
  mul  x11, x11, x5
  add  x11, x11, x10

  # x10: fwd_ptr
  # x11: rev_ptr

loop:
  lw   x5,  0(x10)
  lw   x6,  0(x11)
  sw   x5,  0(x11)
  sw   x6,  0(x10)
  addi x7,  x11, 0
  addi x10, x10, 4
  addi x11, x11, -4
  bne  x10, x7, loop

  jr   x1
```

Figure 10: Assembly for Reverse on Array Data Structure

Part 4.B Analyzing Performance of a Linked-List Data Structure

Figure 11 shows C-code which implements the reverse operations for a linked-list data structure with an even number of elements. Figure 12 shows corresponding TinyRV1 assembly code for the function. Recall that arguments are stored in `x10` and `x11`. We need additional load instructions to retrieve the next and previous pointers for iterating through the list. Note that we use a non-zero offset when accessing these next and previous pointers since we know ahead of time where these fields are located relative to the value field.

Draw a pipeline diagram illustrating at least one iteration of the loop show in Figure 12. Assume the cannonical five-stage fully bypassed TinyRV1 processor. You should draw as many iterations as you need in order to determine the steady-state behavior of the loop. Carefully consider which memory accesses hit or miss in the data cache. **Use your pipeline diagram to estimate the overall execution time in cycles for this operation. Fill in the corresponding row of Figure 8**. You will need to decompose the CPI into its various components based on what stalls and/or squashes occur in the execution. **You must show your work.**

```
void list_reverse( node* head, node* tail ) {

  # Code only works for even n
  assert( n % 2 == 0 )

  int* fwd_ptr = head;
  int* rev_ptr = tail;
  int* rev_ptr_last = 0;

  do {

    # Swap values at fwd and rev pointers
    int temp = fwd_ptr->value;
    fwd_ptr->value = rev_ptr->value;
    rev_ptr->value = temp;

    # Save rev pointer for exit condition check
    rev_ptr_last = rev_ptr;

    # Update fwd and rev pointers
    fwd_ptr = fwd_ptr->next;
    rev_ptr = rev_ptr->prev;

  }
  while ( fwd_ptr != rev_ptr_last );

}
```

Figure 11: C-Code for Reverse on Linked-List Data Structure

```
  # x10: fwd_ptr
  # x11: rev_ptr

loop:
  lw   x5,  0(x10)
  lw   x6,  0(x11)
  sw   x5,  0(x11)
  sw   x6,  0(x10)
  addi x7,  x11, 0
  lw   x10, 4(x10)
  lw   x11, 8(x11)
  bne  x10, x7, loop

  jr   x1
```

Figure 12: Assembly for Reverse on Linked-List Data Structure

Part 4.C Comparison of Data Structures

Compare the performance of the two data structures and generalize your results by answering the following questions.

- Which data structure performs better in this specific example and why?
- How would the execution time change as a function of cache-line size assuming we always only allocate one linked-list node per cache line?
- How would the execution time change if we assumed larger cache lines and that the memory allocator organizes multiple linked-list nodes on the same cache line?
- How does the execution time change as the number of elements in the data structure grows asymptotically large? How does this relate to the theoretical asymptotic behavior of $O(n)$?
- How would the execution time change if both data structures were already present in the cache such that there were no cache misses?
- Can we draw any broad conclusions about the cache behavior of more regular array- or matrix-based data structures vs. more irregular list-, tree-, or graph-based data-structures that make extensive use of dynamic memory allocation and pointers?

Problem 5. Multicore Two-Level Data Cache

In this problem, you will explore the two different multicore cache microarchitectures shown abstractly in Figures 13 and 14. The microarchitecture in Figure 13 includes private L1 data caches. In lecture, we also discussed how multi-level cache hierarchies can potentially reduce the average memory access latency. The microarchitecture in Figure 14 adds a shared L2 data cache. For this problem ignore the instruction cache, and assume that all caches are blocking, write-back, and write-allocate.

We will examine how these two microarchitectures execute the following transaction sequence. All caches initially start with every cache line invalidated, and each access reads four bytes at the given address. There are no write accesses, so there are no evictions. Assume that the transaction pattern repeats such that each processor executes a total of 256 transactions. To simplify our analysis, assume that processor 0 completely executes its first four transactions, then processor 1 completely executes its first four transactions, then processor 0 completely executes the next four transactions, and so on. This means there is never any contention between processors in the memory system because we are only ever executing one transaction at a time.

Processor 0: `0x004, 0x020, 0x018, 0x03c, 0x004, 0x020, ...` (all requests read four bytes)

Processor 1: `0x404, 0x430, 0x428, 0x41c, 0x404, 0x430, ...` (all requests read four bytes)

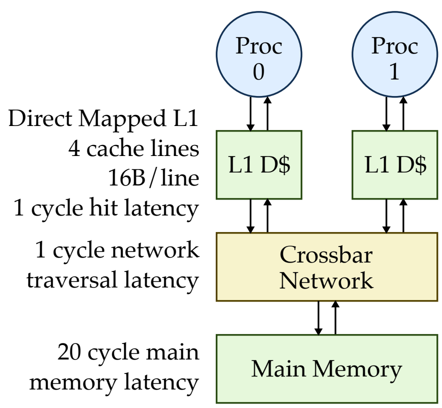

Figure 13: Multicore Single-Level Data Cache

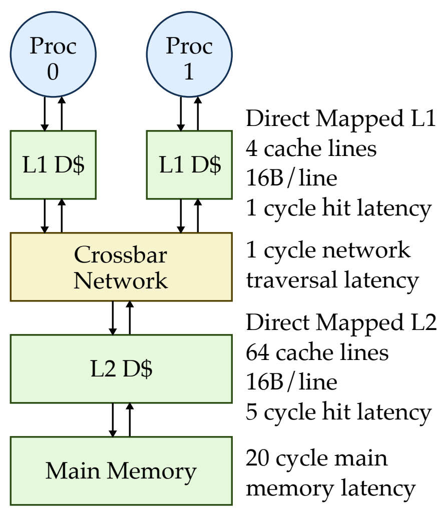

Figure 14: Multicore Two-Level Data Cache

Part	Microarchitecture	L1 Hit Latency (cycles)	L1 Miss Rate	L1 Miss Penalty (cycles)	AMAL (cycles)
3.A	**One-Level**	1			
3.B	**Two-Level**	1			

Figure 15: Memory Performance for Entire Loop on Two Different Microarchitectures

Part 5.A Performance of Multicore Single-Level Data Cache

In this part, we will calculate the average memory access latency for the transactions shown above executing on the multicore single-level microarchitecture shown in Figure 13. Each L1 data cache is direct mapped with four cache lines. Each cache line contains 16 bytes. The L1 data cache hit latency is one cycle. A miss refill must go across the crossbar, access main memory, and then go back across the crossbar. Assume there is no serialization latency. Each crossbar traversal has a latency of one cycle, and main memory has a total access latency of 20 cycles.

Fill in the following table to illustrate where the cache lines are placed in the L1 data caches. Use a dash symbol (–) to indicate if the corresponding cache line in the cache is invalid or invalidated. Use a (*) to indicate a hit. If the state of a cache line does not change, then you can leave the corresponding entry blank. **Fill in the appropriate row of the table in Figure 15 considering all 512 transactions. Please clearly explain how you calculate the miss rate and the miss penalty. You must show your work and state any assumptions.**

Byte Address	Proc 0 L1 D$ L0	Proc 0 L1 D$ L1	Proc 0 L1 D$ L2	Proc 0 L1 D$ L3	Proc 1 L1 D$ L0	Proc 1 L1 D$ L1	Proc 1 L1 D$ L2	Proc 1 L1 D$ L3	miss?
P0: 0x004	00	--	--	--	--	--	--	--	Y
P0: 0x020									
P0: 0x018									
P0: 0x03c									
P1: 0x404									
P1: 0x430									
P1: 0x428									
P1: 0x41c									
P0: 0x004									
P0: 0x020									
...									

Part 5.B Performance of Multicore Two-Level Data Cache

In this part, we will calculate the average memory access latency for the transactions shown above executing on the multicore two-level microarchitecture shown in Figure 14. The L1 data caches are the same as in the previous part. The shared L2 data cache is direct mapped with 64 cache lines. Each cache line contains 16 bytes. The hit latency for the L2 cache is 5 cycles. In this design, the L2 cache is inclusive of the L1 caches. This means that every line that is in an L1 cache must also be in the L2 cache. If we have to replace a line in the L2 cache, then we must also invalidate the corresponding line if it is present in the other L1 cache.

Fill in the following table to illustrate where the cache lines are placed in the L1 and L2 data caches. Since there are 64 lines in the L2 data cache, you must specify the appropriate line index at the top of each column for the L2 data cache. Use a dash symbol (–) to indicate if the corresponding cache line in the cache is invalid or invalidated. Use a (*) to indicate a hit. If the state of a cache line does not change then you can leave the corresponding entry blank. **Fill in the appropriate row of the table in Figure 15 considering all 512 transactions. Please clearly explain how you calculate the miss rate and the miss penalty. You must show your work and state any assumptions.**

Byte	**Proc 0 L1 D$**				**Proc 1 L1 D$**				**L1**	**L2 D$**								**L2**
Address	**L0**	**L1**	**L2**	**L3**	**L0**	**L1**	**L2**	**L3**	**miss?**	**L0**								**miss?**
`P0: 0x004`	00	--	--	--	--	--	--	--	Y	00	--	--	--	--	--	--	--	Y
`P0: 0x020`																		
`P0: 0x018`																		
`P0: 0x03c`																		
`P1: 0x404`																		
`P1: 0x430`																		
`P1: 0x428`																		
`P1: 0x41c`																		
`P0: 0x004`																		
`P0: 0x020`																		
...																		