The PlayStation Program Analyzer

Basic Operation and Optimizations

Program Analyzer & Optimization Techniques

- What is the Program Analyzer?
- Using the Program Analyzer
- Interpreting the Results
- Optimization Techniques
- Having Your Programs Analyzed

What is the Program Analyzer?

- A Specialized Logic Analyzer
- Hand-Built
  - Only 5 in the world so far.
- Expen$ive
  - Would cost upwards of $100,000 if sold.
- Not a Hardware Profiler
- Updates are in the works

Using The Program Analyzer

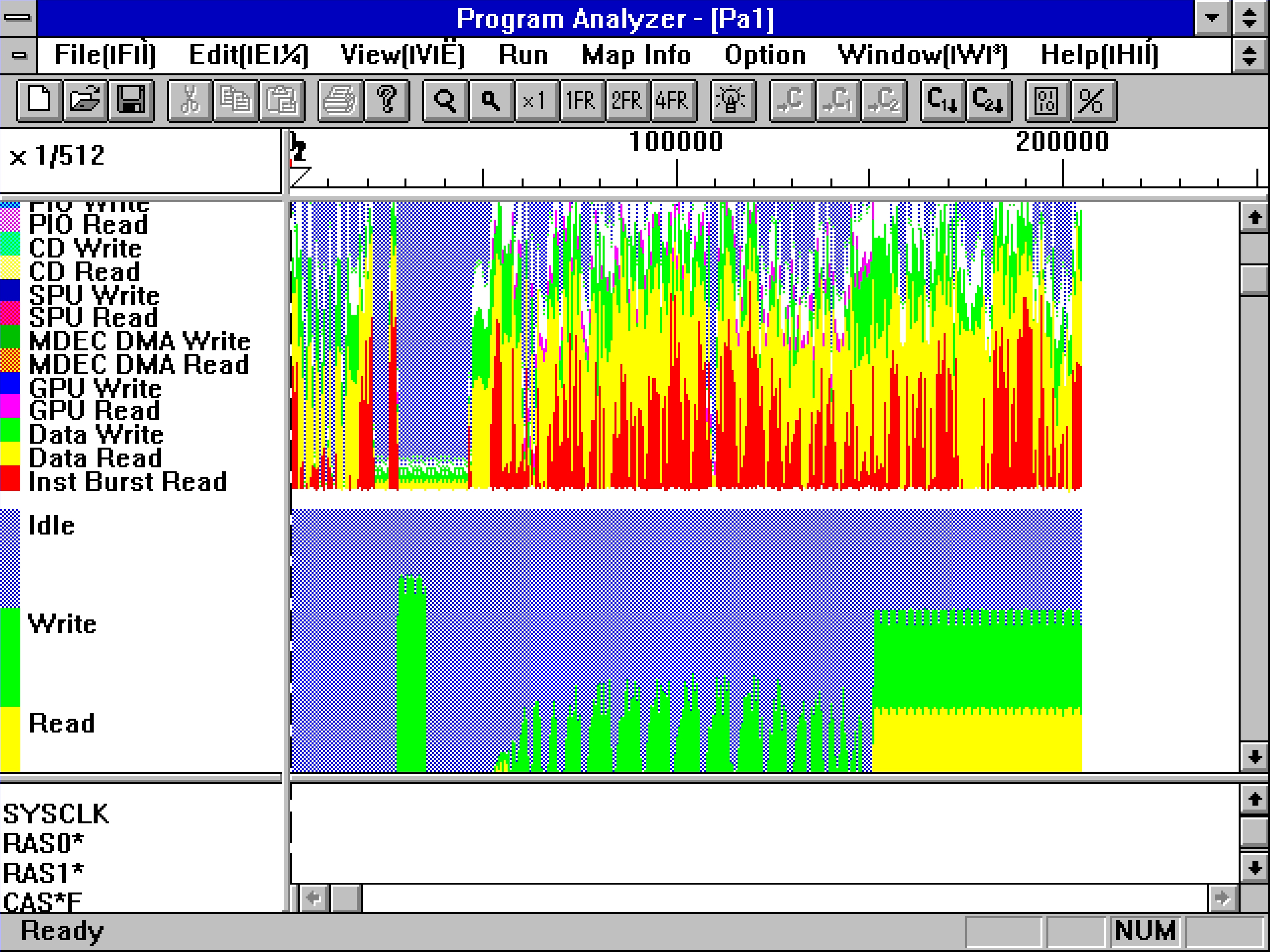
- Capturing Program Execution
- What Are We Looking For?
- Interpreting The Results

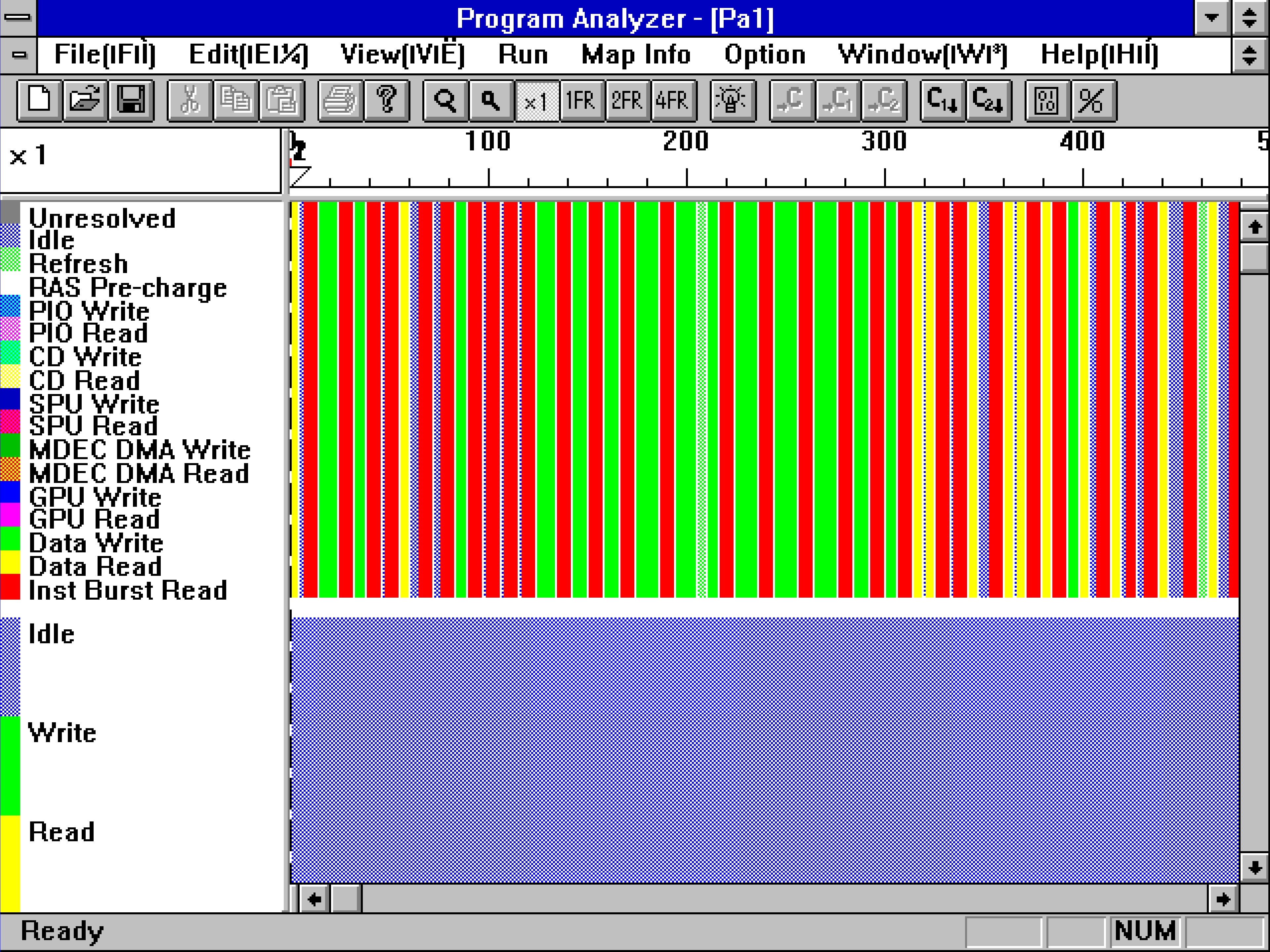
Using The Program Analyzer • Capturing Program Execution

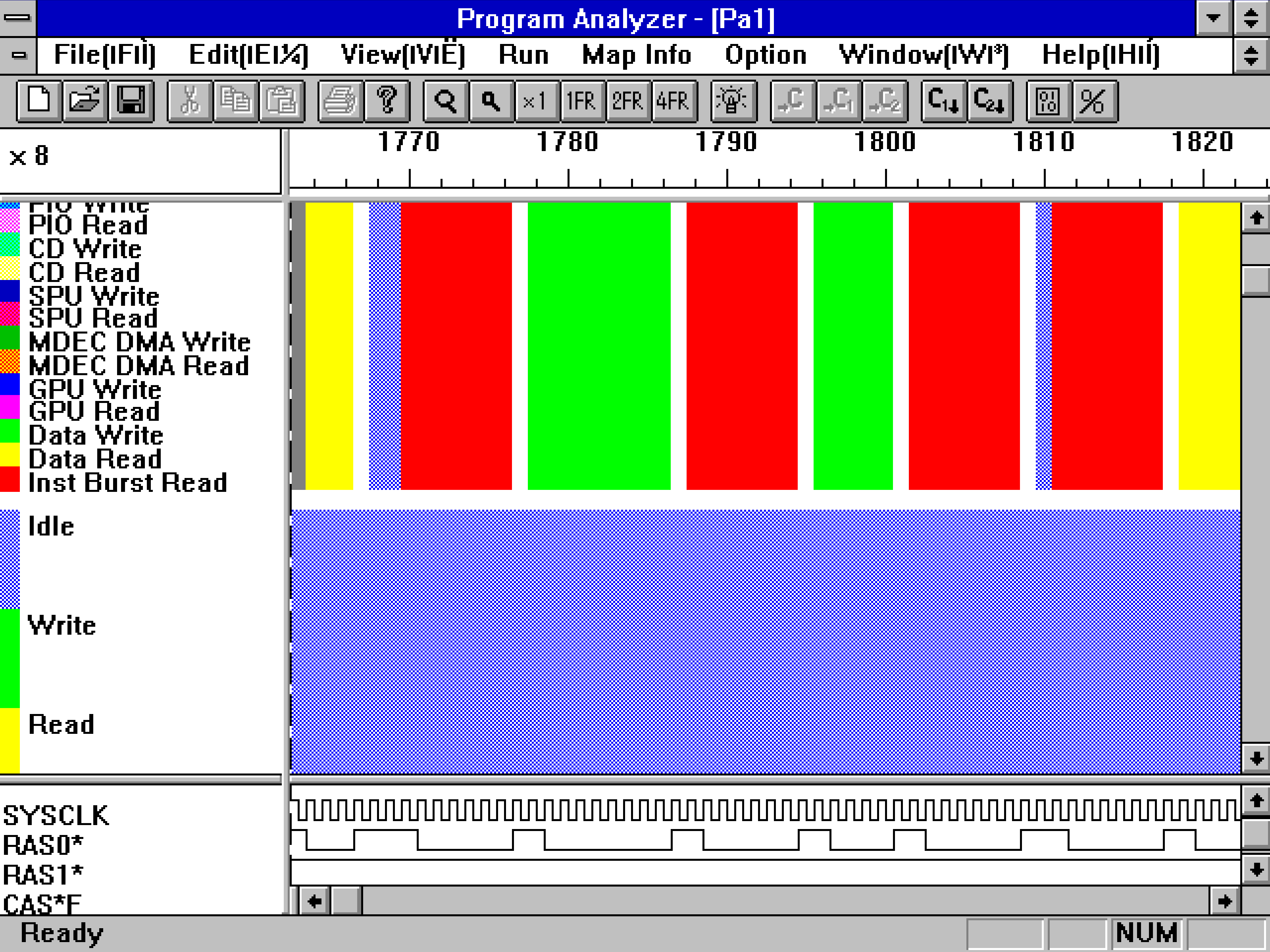
- Capture initiated by trigger button
  - Future versions may support capture under program control.
- Stores a capture of all bus access
  - All bus signals (read / write enable lines, etc.)
- Captures up to about 3.7 frames
  - Requires nearly 50 mb of storage.

Using The Program Analyzer • What Are We Looking For?

- Idle time
  - Bus not being utilized to full potential.
- Inefficient Memory Accesses
  - Poor I-cache usage.
  - Poor D-cache usage.
  - Memory Page Faults.


Program Analyzer - [Pa1]
File(IFIÌ) Edit(IEI¼) View(IVIË) Run Map Info Option Window(IWI³) Help(IHIÍ)
×1 1FR 2FR 4FR
× 1/512
100000
200000
PIO Write
PIO Read
CD Write
CD Read
SPU Write
SPU Read
MDEC DMA Write
MDEC DMA Read
GPU Write
GPU Read
Data Write
Data Read
Inst Burst Read
Idle
Write
Read
SYSCLK
RAS0*
RAS1*
CAS*F
Ready
NUM


Program Analyzer - [Pa1]
File(IFIÌ)
Edit(IEI¼)
View(IVIË)
Run
Map Info
Option
Window(IWI³)
Help(IHIÍ)
×1
1FR
2FR
4FR
×1
100
200
300
400
Unresolved
Idle
Refresh
RAS Pre-charge
PIO Write
PIO Read
CD Write
CD Read
SPU Write
SPU Read
MDEC DMA Write
MDEC DMA Read
GPU Write
GPU Read
Data Write
Data Read
Inst Burst Read
Idle
Write
Read
Ready
NUM

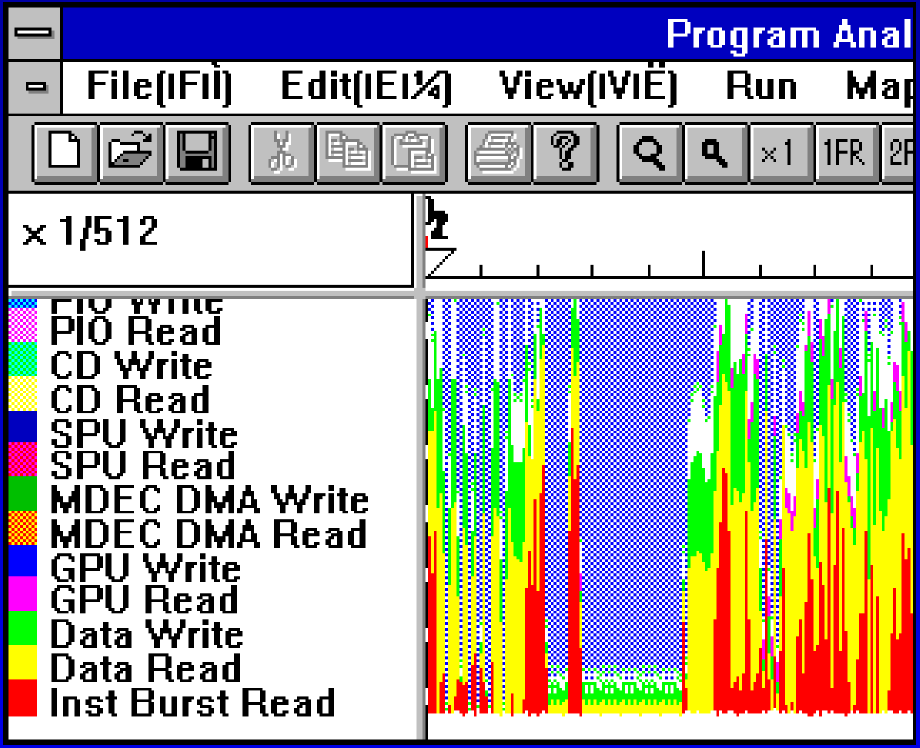

Program Analyzer - [Pa1]
File(|F|)
Edit(|E|¼)
View(|V|Ë)
Run
Map Info
Option
Window(|W|³)
Help(|H|Í)
×1
1FR
2FR
4FR
× 8
1770
1780
1790
1800
1810
1820
PIO Write
PIO Read
CD Write
CD Read
SPU Write
SPU Read
MDEC DMA Write
MDEC DMA Read
GPU Write
GPU Read
Data Write
Data Read
Inst Burst Read
Idle
Write
Read
SYSCLK
RAS0*
RAS1*
CAS*F
Ready
NUM

Interpreting the Results • Idle Time

- Small amounts of idle time are OK
  - Stuff is happening in registers. RAM, ROM, and hardware registers not being accessed.
  - R3000 operations 1 to 6 ticks for most cases.
  - GTE operations up to 38 ticks.

Interpreting the Results • Idle Time

- Large amounts of idle time usually BAD
  - Looping while polling for a hardware operation to complete.
  - OK if executing code out of cache & using registers.

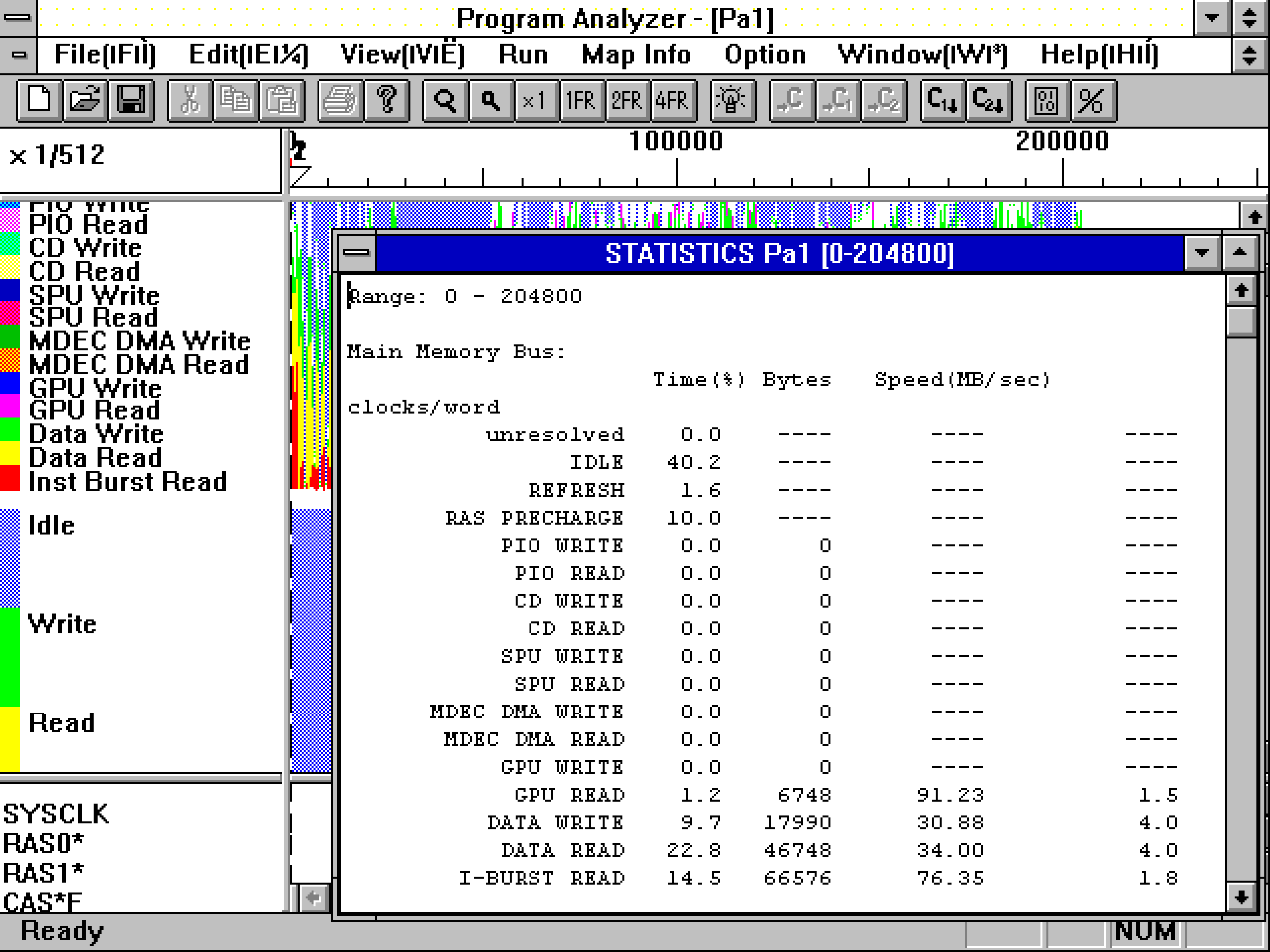


Interpreting The Results • Inefficient Memory Accesses

- Lots of I-cache burst reads
  - Means code not running from cache, has to fill cache 16 bytes at a time.
  - Potential for causing memory page faults.
- Lots of bus time spent on data reads & writes
  - Suggests need for more efficient usage of the D-cache (aka “Scratchpad”).
  - Suggests lots of memory page faults.

Interpreting The Results • Statistics

- Show percentage of time the bus is used for various types of operation.
  - Shows selected range of clock ticks.
  - Shows memory transfer amount.
  - Shows memory transfer rates.



```
Range: 0 - 204800

Main Memory Bus:
                                           Time(%) Bytes   Speed(MB/sec)
clocks/word
                    unresolved     0.0     ----       ----          ----
                          IDLE    40.2     ----       ----          ----
                       REFRESH     1.6     ----       ----          ----
                 RAS PRECHARGE    10.0     ----       ----          ----
                     PIO WRITE     0.0        0       ----          ----
                      PIO READ     0.0        0       ----          ----
                      CD WRITE     0.0        0       ----          ----
                       CD READ     0.0        0       ----          ----
                     SPU WRITE     0.0        0       ----          ----
                      SPU READ     0.0        0       ----          ----
               MDEC DMA WRITE      0.0        0       ----          ----
                MDEC DMA READ      0.0        0       ----          ----
                     GPU WRITE     0.0        0       ----          ----
                      GPU READ     1.2     6748      91.23           1.5
                    DATA WRITE     9.7    17990      30.88           4.0
                     DATA READ    22.8    46748      34.00           4.0
                  I-BURST READ    14.5    66576      76.35           1.8
```

Optimization Techniques

- I-Cache Usage
- D-Cache Usage
- Avoiding Memory Page Faults
- Avoiding Idle Time

Optimization Techniques • I-Cache

- I-Cache is 4k long.
  - Divided into 16-byte “lines”.
    - 4 instructions per line.
    - Each line has a “tag” indicating which physical address is cached.
      - Instruction reads check tag to see if code is in cache.

Optimization Techniques • I-Cache

- Not like a typical PC cache.
  - Code position in I-cache depends on low-order 12-bits of address.
    - Code may wrap from end of cache to beginning.
  - High nybble of instruction address controls if code is cacheable .

Optimization Techniques • I-Cache

- Instruction reads from cache take 1 tick.
- If instruction not in cache, the appropriate cache line is burst filled from RAM.
  - Takes 4-7 ticks to read from RAM.
    - Plus 1 tick to transfer from cache to R3000.
  - Reads entire line in 1 burst.
    - Doesn’t read portion of line before instruction address.

Optimization Techniques • I-Cache

- Code position in I-cache depends on low-order 12-bits of address.

Function Name	Memory Addresses	Cache Addresses
move_loop()	0x800F2346 to 0x800F24F1	0x346 to 0x4F1
move_ship()	0x80051534 to 0x80051599	0x534 to 0x599
move_missiles()	0x80051F34 to 0x80052233	0xF34 to 0xFFF and 0x000 to 0x233
move_enemies()	0x801D34F2 to 0x801D368B	0x4F2 to 0x68B
calc_score()	0x8013563E to 0x801356F1	0x63E to 0x6F1

Optimization Techniques • I-Cache

- Code position in I-cache depends on low-order 12-bits of address.

Function Name	Memory Addresses	Cache Addresses
`move_loop()`	`0x800F2346 to 0x800F24F1`	`0x346 to 0x4F1`
`move_enemies()`	`0x801F24f2 to 0x801F268B`	`0x4F2 to 0x68B`
`move_ship()`	`0x801F268C to 0x801F26F1`	`0x68C to 0x6F1`
`move_missiles()`	`0x801F26F2 to 0x801F29F1`	`0x6F2 to 0x9F1`
`calc_score()`	`0x801F29F2 to 0x801F2AA5`	`0x9F2 to 0xAA5`

Optimization Techniques • I-Cache

- Code may wrap from end of cache to beginning without penalty.
  - Execution continues at start of cache.

```
Function Name           Memory Addresses             Cache Addresses
--------------------------------------------------------------------
move_missile()          0x80051F34 to 0x80052233     0xF34 to 0xFFF
                                                 and 0x000 to 0x233
```

Optimization Techniques • I-Cache

- High nybble of instruction address controls if code is cacheable.
  - 0x8 & 0x0 = Cacheable
  - 0xA = Not Cacheable

```
Function Name          Memory Addresses              Cache Addresses
--------------------------------------------------------------------
move_ship()            0x801F268C to 0x801F26F1      0x68C to 0x6F1
move_ship()            0xA01F268C to 0xA01F26F1      code not cached
```

Optimization Techniques • I-Cache

- Control cache usage by using function pointers to call routines.
  - Manipulate pointers at runtime to dynamically control cache usage.

```
If the function pointer ‘move’ is pointing at a routine that
is located at position 0x001E8584 in physical memory, then
you can control the cache usage by adjusting the high nybble
of the address:

int (move *)(int x) = 0x001E8584;    // Default = cacheable
(long)move |= 0xAFFFFFFF;            // make non-cacheable
(long)move &= 0x8FFFFFFF;            // make cacheable
```

Optimization Techniques • D-Cache

- D-Cache (aka "Scratchpad") is 1K long.
  - Memory-mapped.
    - Address range 0x1F800000 to 0x1F8003FF.
  - 1 tick read / write access.
    - RAM writes normally 3 to 6 ticks.
    - RAM reads normally 2 to 6 ticks.
  - Cannot be used for DMA transfers.

Optimization Techniques • D-Cache

- Use for commonly accessed globals.
- Use as work area for polygon subdivision.
- Place stack in D-Cache.
  - Do on temporary basis for functions with many arguments or lots of local variables.

```
SetStack( 0x1f8001FC );
call_function_with_scratchpad_used_for_stack();
ResetStack();
```

Optimization Techniques • D-Cache

- Placing the stack in the D-Cache.

```
#define SetStackAddr(addr) { \
__asm__ volatile ("move $8,%0"     ::"r"(addr):"$8","memory"); \
__asm__ volatile ("sw $29,0($8)"   ::          :"$8","memory"); \
__asm__ volatile ("addiu $8,$8,-4" ::          :"$8","memory"); \
__asm__ volatile ("move $29,$8"    ::          :"$8","memory");  }

#define ResetStack() {\
__asm__ volatile ("addiu $29,$29,4":::"$29","memory"); \
__asm__ volatile ("lw $29,0($29)"  :::"$29","memory"); }

#define GetStackAddr(addr) { \
__asm__ volatile ("move $8,%0"     ::"r"(addr):"$8","memory"); \
__asm__ volatile ("sw $29,0($8)"   ::          :"$8","memory"); }
```

Optimization Techniques • D-Cache

- If you need more space, try this:
  - Save part of the D-cache to DRAM.
  - Use saved area temporarily.
    - Do not call routines that use saved portions.
  - Restore D-cache from DRAM.
  - If done properly, save & restore can take fewer cycles than not using D-cache at all.

Optimization Techniques • Avoiding Memory Page Faults

- Memory Page Characteristics
  - Memory pages are 1k long, on 1k boundaries.
    - 0x-----000 to 0x-----3FF, 0x-----400 to 0x-----7FF, 0x-----800 to 0x-----BFF, 0x-----C00 to 0x-----FFF
  - Page faults occur when you read or write a different page from the previous read / or write operation.
  - Burst reads to I-cache can cause page fault.
    - Make sure your code is running from the I-cache.
  - D-cache reads & writes don’t cause page fault.

Optimization Techniques • Avoiding Memory Page Faults

- Why worry about page faults?
  - Page faults slow down RAM Reading access.
    - 1st read from a particular page in RAM takes 5 ticks.
      - 4 ticks for RAM to R3000 R-buffer, 1 tick R-buffer to CPU.
    - 2nd & subsequent reads take just 2 ticks.
      - 1 tick for RAM to R3000 R-buffer, 1 tick R-buffer to CPU.
    - True for both data reads and I-cache burst read.
      - Keep code execution in I-cache to avoid page faults.

Optimization Techniques • Avoiding Memory Page Faults

- Why worry about page faults?
  - Page faults slow down RAM Writing access.
    - 1st write to a particular page in RAM takes 5 ticks.
      - 1 tick for CPU to R3000 W-buffer, 4 ticks W-buffer to RAM.
    - 2nd & subsequent writes take just 3 ticks.
      - 1 tick for CPU to R3000 W-buffer, 2 ticks W-buffer to RAM.

Optimization Techniques • Avoiding Memory Page Faults

- Tips For Avoiding Memory Page Faults
  - Registers, Registers, Registers!
    - Read as much data as possible into registers from source, then write from registers.
  - Use scratchpad.
    - Store frequently accessed items on scratchpad.
    - When copying data or doing calculations, read data to scratchpad, then write out from there.

Optimization Techniques • Avoiding Idle Time

- Interleave R3000 & GTE instructions.
- Use double or triple buffer schemes.
- Use threads

Optimization Techniques • Avoiding Idle Time

- Avoid using blocking mode of hardware sync functions.
  - ***DrawSync***, ***VSync***, ***CdReadSync***, etc.
    - Use callback routines whenever possible.
    - When you must use them, whenever possible, use these functions BEFORE an operation, not after.

```
/* BAD */
DrawOTag( ot );
DrawSync(0);
…
…
DrawOTag( ot );
```

```
/* GOOD */
DrawOTag( ot );
…
…
DrawSync(0);
DrawOTag( ot );
```

Having Your Programs Analyzed

- Technical Requirements
- When To Do It
- Materials Required

Having Your Programs Analyzed • Technical Requirements

- Program runs on standard blue test PlayStation
  - Runs in 2mb of system RAM.
  - Bootable gold PlayStation CD-ROM disc.
  - Program does not call PollHost, PCOpen, PCRead, PCWrite, etc.

Having Your Programs Analyzed • When to do it

- When you are not achieving your target frame rate by a significant margin.
  - If you’re already running at 60 fps, don’t bother.
  - If you’re running at 10fps when you need 30, call us!
- When your frame rate is widely variable.
  - 30 fps during some portions, 10 fps in others.

Having Your Programs Analyzed • When to do it

- Program is more than 50% done, but less than 90%.
  - Refers to code, not artwork, sound effects, music scores, level data, etc.
  - Do it before it’s too late to make significant changes.

Having Your Programs Analyzed • When to do it

- Before alpha-test or beta-test stage
  - If you need to make changes, do it before going to test.
- At least a month or two before you need to make your final submission.
  - Allow enough time to make significant changes if necessary.

Having Your Programs Analyzed • Materials Required

- Make Appointment
  - Contact your account executive or developer support
    - Send Email to DevTech_Support@interactive.sony.com
- Bring Bootable gold CD
  - You may want to bring 2 or 3… just in case.
- Bring MAP file created by linker, with symbol information matching the executable on your gold disc.

Having Your Programs Analyzed • Materials Required

- Programmer
  - We can analyze your program for you, but the interpretation of the results will be more meaningful if one or more of the main programmers is on hand.
    - The programmer should bring along a complete copy of the source code and any data required to rebuild a bootable disc.

The End