Zero Day Exploit

Processor Specifications

# Instructions

|  |  |  |
| --- | --- | --- |
| Instruction | Description | Privileged |
| DSTR | Moves data from memory to the disk | Yes |
| DLDR | Moves data from the disk to memory | Yes |
| CTD | Counts the number of attached disks | No |
| CPM | Changes the processor mode | Yes |
| HLT | Halts the processor | Yes |
| PUSH | Pushes a value to the stack | No |
| POP | Pops a value from the stack | No |
| B | Performs a far jump, relative to the current execution position | Yes |
| BL | In addition to a far jump, also sets r14 to the previous execution position | Yes |
| BX | Performs an absolute far jump, while changing the processor mode | Yes |
| STR | Moves data from a register to memory, in integer form | No |
| LDR | Moves data from memory to a register, in integer form | No |
| STRB | Moves the least significant byte of a register to memory | No |
| LDRB | Moves a byte from memory into a register | No |
| SWP | Swaps an integer between memory into a register | No |
| SWPB | Swaps a byte between memory and a register | No |
| MUL | Multiplies two operands | No |
| SVC | Executes an interrupt, elevating the execution state in the process | No |
| EOR | Performs an exclusive or on two operands | No |
| SUB | Subtracts two operands | No |
| RSB | Reverse-subtracts two operands | No |
| ADD | Adds two operands | No |
| TST | Sets status flags based on the result of ANDing two operands | No |
| TEQ | Sets status flags based on the result of XORing two operands | No |
| CMP | Sets status flags based on the result of subtracting two operands | No |
| CMN | Sets status flags based on the result of adding two operands | No |
| ORR | Performs an or on two operands | No |
| MOV | Moves a value between registers | No |
| BIC | Clears the specified bits of a register | No |
| MVN | Moves a value between registers, performing a logical NOT on the second | No |
| AND | Performs an AND on two operands | No |

# Instruction Arguments

|  |  |
| --- | --- |
| Instruction | Argument Scheme |
| DSTR | Rmem\_start, Rdisk\_start, Rlength, ARGid |
| DLDR | Rmem\_start, Rdisk\_start, Rlength, ARGid |
| CTD | Rto\_store |
| CPM | IMMmode |
| HLT | N/A |
| PUSH | Rto\_push |
| POP | Rto\_pop |
| B | IMMto\_jump |
| BL | IMMto\_jump |
| BX | Rto\_jump, Rmode |
| STR | Rbase, Rto\_store, IMMoffset |
| LDR | Rbase, Rto\_store, IMMoffset |
| STRB | Rbase, Rto\_store, IMMoffset |
| LDRB | Rbase, Rto\_store, IMMoffset |
| SWP | Rbase, Rto\_store, IMMoffset |
| SWPB | Rbase, Rto\_store, IMMoffset |
| MUL | Rdest, Rop1, ARGop2 |
| SVC | IMMinterrupt |
| EOR | Rdest, Rop1, ARGop2 |
| SUB | Rdest, Rop1, ARGop2 |
| RSB | Rdest, Rop1, ARGop2 |
| ADD | Rdest, Rop1, ARGop2 |
| TST | Rop1, ARGop2 |
| TEQ | Rop1, ARGop2 |
| CMP | Rop1, ARGop2 |
| CMN | Rop1, ARGop2 |
| ORR | Rdest, Rop1, ARGop2 |
| MOV | Rdest, ARGop |
| BIC | Rdest, Rto\_clear |
| MVN | Rdest, ARGop |
| AND | Rdest, Rop1, ARGop2 |

# What do R, ARG, and IMM mean?

R indicates that that argument is a register. The subscript denotes its purpose.

ARG indicates that either a register or constant can be used there.

IMM indicates that a constant must be used, prefaced with a # symbol.

# Mask / Data Pairs

|  |  |  |
| --- | --- | --- |
| Instruction | Unique Data | Mask |
| DSTR | 0000-1011-0000-0000-0000-0000-0000-0000 | 0000-1111-0000-0000-0000-0000-0000-0000 |
| DLDR | 0000-1001-0000-0000-0000-0000-0000-0000 | 0000-1111-0000-0000-0000-0000-0000-0000 |
| CTD | 0000-0101-1111-1111-1111-1111-1111-0000 | 0000-1111-1111-1111-1111-1111-1111-0000 |
| CPM | 0000-1111-1111-1111-1111-1111-1111-0000 | 0000-1111-1111-1111-1111-1111-1111-0000 |
| HLT | 0000-1111-1111-1111-1111-1111-1111-1111 | 0000-1111-1111-1111-1111-1111-1111-1111 |
| PUSH | 0000-1110-0111-1111-1111-1111-1111-0000 | 0000-1111-1111-1111-1111-1111-1111-0000 |
| POP | 0000-1111-1111-1111-1111-1111-1111-0000 | 0000-1111-1111-1111-1111-1111-1111-0000 |
| B | 0000-1010-0000-0000-0000-0000-0000-0000 | 0000-1111-0000-0000-0000-0000-0000-0000 |
| BL | 0000-1011-0000-0000-0000-0000-0000-0000 | 0000-1111-0000-0000-0000-0000-0000-0000 |
| BX | 0000-0001-0010-1111-1111-1111-0000-0000 | 0000-1111-1111-1111-1111-1111-0000-0000 |
| STR | 0000-0100-0000-0000-0000-0000-0000-0000 | 0000-1100-0101-0000-0000-0000-0000-0000 |
| LDR | 0000-0100-0001-0000-0000-0000-0000-0000 | 0000-1100-0101-0000-0000-0000-0000-0000 |
| STRB | 0000-0100-0100-0000-0000-0000-0000-0000 | 0000-1100-0101-0000-0000-0000-0000-0000 |
| LDRB | 0000-0100-0101-0000-0000-0000-0000-0000 | 0000-1100-0101-0000-0000-0000-0000-0000 |
| SWP | 0000-0001-0000-0000-0000-0000-1001-0000 | 0000-1111-1111-0000-0000-1111-1111-0000 |
| SWPB | 0000-0001-0100-0000-0000-0000-1001-0000 | 0000-1111-1111-0000-0000-1111-1111-0000 |
| MUL | 0000-0000-0000-0000-0000-0000-1001-0000 | 0000-0000-0010-0000-0000-0000-1111-0000 |
| SVC | 0000-1111-0000-0000-0000-0000-0000-0000 | 0000-1111-0000-0000-0000-0000-0000-0000 |
| EOR | 0000-0000-0010-0000-0000-0000-0000-0000 | 0000-1101-1110-0000-0000-0000-0000-0000 |
| SUB | 0000-0000-0100-0000-0000-0000-0000-0000 | 0000-1101-1110-0000-0000-0000-0000-0000 |
| RSB | 0000-0000-0110-0000-0000-0000-0000-0000 | 0000-1101-1110-0000-0000-0000-0000-0000 |
| ADD | 0000-0000-1000-0000-0000-0000-0000-0000 | 0000-1101-1110-0000-0000-0000-0000-0000 |
| TST | 0000-0001-0000-0000-0000-0000-0000-0000 | 0000-1101-1110-0000-0000-0000-0000-0000 |
| TEQ | 0000-0001-0010-0000-0000-0000-0000-0000 | 0000-1101-1110-0000-0000-0000-0000-0000 |
| CMP | 0000-0001-0100-0000-0000-0000-0000-0000 | 0000-1101-1110-0000-0000-0000-0000-0000 |
| CMN | 0000-0001-0110-0000-0000-0000-0000-0000 | 0000-1101-1110-0000-0000-0000-0000-0000 |
| ORR | 0000-0001-1000-0000-0000-0000-0000-0000 | 0000-1101-1110-0000-0000-0000-0000-0000 |
| MOV | 0000-0001-1010-0000-0000-0000-0000-0000 | 0000-1101-1110-0000-0000-0000-0000-0000 |
| BIC | 0000-0001-1100-0000-0000-0000-0000-0000 | 0000-1101-1110-0000-0000-0000-0000-0000 |
| MVN | 0000-0001-1110-0000-0000-0000-0000-0000 | 0000-1101-1110-0000-0000-0000-0000-0000 |
| AND | 0000-0000-0000-0000-0000-0000-0000-0000 | 0000-1101-1110-0000-0000-0000-0000-0000 |

# What Are These Used For?

These are used to determine the instruction presented during execution and disassembly. The mask is ANDed with the instruction, and the result is compared with the data segment. If they match, it is assumed that that instruction should be executed.

# Condition Codes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Binary Value | Flags | Description |
| EQ | 0000 | Z set | equal |
| NE | 0001 | Z clear | not equal |
| CS | 0010 | C set | unsigned higher or same |
| CC | 0011 | C clear | unsigned lower |
| MI | 0100 | N set | negative |
| PL | 0101 | N clear | positive or zero |
| VS | 0110 | V set | overflow |
| VC | 0111 | V clear | no overflow |
| HI | 1000 | C set and Z clear | unsigned higher |
| LS | 1001 | C clear or Z set | unsigned lower or same |
| GE | 1010 | N equals V | greater or equal |
| LT | 1011 | N not equal to V | less than |
| GT | 1100 | Z clear AND (N equals V) | greater than |
| LE | 1101 | Z set OR (N not equal to V) | less than or equal |
| AL | 1110 | (ignored) | always |

# What are condition codes?

They can follow any instruction, and allow them to be executed conditionally.

# Processor Modes

|  |  |  |
| --- | --- | --- |
| Name | Binary Value | Description |
| User (USR) | 1000 | Unprivileged mode, programs should run here |
| FIQ | 0100 | Fast interrupt |
| Abort (ABT) | 0010 | Entered after an exception |
| IRQ | 0001 | Normal interrupt |
| Undefined (UNDEF) | 1001 | Processor boots here |
| System (SYS) | 0011 | Privileged mode using user mode’s register set |

# Assembler Features

|  |  |  |
| --- | --- | --- |
| Name | Usage | Description |
| Labels | :<NAME>: | All instances of :<NAME>: replaced with the distance from the current location |
| Constants | ? <NAME> = <VALUE> ? | All instances of <NAME> replaced with <VALUE> |
| Comments | ;<COMMENT> | All lines prefaced with a semicolon are ignored upon assembly |
| Hex Data | 0x <DATA> x0 | Injects data during assembly |
| Binary | # <FILE> # | Injects a file or additional ASM code during assembly |

# Graphics Instructions

|  |  |  |
| --- | --- | --- |
| Instruction | Description | Arguments |
| CLR | Queues a screen clear | N/A |
| DRAW | All primitives further are outlined | N/A |
| FILL | All primitives further are filled | N/A |
| RECT | Draws a rectangle | Rx, Ry, Rw, Rh |
| TRI | Draws a triangle | Rx, Ry, Rl, Rh |
| OVL | Draws an oval | Rx, Ry, Rw, Rh |
| COLOR | Sets the current drawing color | Rrgba |
| GCOLOR | Stores the current color into a register | Rrgba |
| TEXT | Draws a string to a specific location | Rx, Ry, Raddr |
| MOUSE | Stores the mouse’s location into two registers | Rx, Ry |
| ROT | Applies a rotation to the next render object | Ro |
| SCR | Stores the screen size to two registers | Rx, Ry |
| BLK | Blocks screen updates | N/A |
| UBLK | Unblocks screen updates | N/A |

# Mask / Data Pairs

|  |  |  |
| --- | --- | --- |
| Instruction | Unique Data | Mask |
| CLR | 0000-0000-1111-1001-0000-0000-0000-0000 | 0000-1111-1111-1111-0000-0000-0000-0000 |
| DRAW | 0000-0000-0111-1001-0000-0000-0000-0000 | 0000-1111-1111-1111-0000-0000-0000-0000 |
| FILL | 0000-0000-1011-1001-0000-0000-0000-0000 | 0000-1111-1111-1111-0000-0000-0000-0000 |
| RECT | 0000-0000-1101-1001-0000-0000-0000-0000 | 0000-1111-1111-1111-0000-0000-0000-0000 |
| TRI | 0000-0000-1110-1001-0000-0000-0000-0000 | 0000-1111-1111-1111-0000-0000-0000-0000 |
| OVL | 0000-0000-0110-1001-0000-0000-0000-0000 | 0000-1111-1111-1111-0000-0000-0000-0000 |
| COLOR | 0000-0000-1010-1001-0000-0000-0000-0000 | 0000-1111-1111-1111-0000-0000-0000-0000 |
| GCOLOR | 0000-0000-1100-1001-0000-0000-0000-0000 | 0000-1111-1111-1111-0000-0000-0000-0000 |
| TEXT | 0000-0000-0100-1001-0000-0000-0000-0000 | 0000-1111-1111-1111-0000-0000-0000-0000 |
| MOUSE | 0000-0000-1000-1001-0000-0000-0000-0000 | 0000-1111-1111-1111-0000-0000-0000-0000 |
| ROT | 0000-0000-0100-1001-0000-0000-0000-0000 | 0000-1111-1111-1111-0000-0000-0000-0000 |
| SCR | 0000-0000-1001-1001-0000-0000-0000-0000 | 0000-1111-1111-1111-0000-0000-0000-0000 |
| BLK | 0000-0000-0001-1001-0000-0000-0000-0000 | 0000-1111-1111-1111-0000-0000-0000-0000 |
| UBLK | 0000-0000-0101-1001-0000-0000-0000-0000 | 0000-1111-1111-1111-0000-0000-0000-0000 |

# How are these different from normal instructions?

They allow for graphics manipulation, rather than data operations. In addition, ALL graphics operations must be executed in a privileged mode.

# Network Instructions

|  |  |  |
| --- | --- | --- |
| Instruction | Description | Arguments |
| OPN | Opens a connection | RID, ARGIP |
| CLS | Closes a connection | RID |
| SND | Sends data along a connection | RID, Raddr, Rlength |
| RCV | Reads data from a connection | RID, Raddr, Rlength |
| SIZ | Stores the size of the receive buffer | RtoStore |
| DMP | Clears the receive buffer | N/A |

# Data Structures

## Interrupt Vector Table (IVT)

The IVT specifies where the processor should jump after and interrupt is called. The IVT’s base address is defined in R12, whose value is not linked to the processor mode. This register, like R15, can only be modified from a privileged mode.

The IVT is simply a set of 32 bit absolute addresses. To calculate the jump point location, consider this expression:

FIQ\_R14 = GLOB\_r15

GLOB\_R15 = [GLOB\_R12 + (<INTERRUPT\_ID> \* 4)]

PROCESSOR\_MODE = FIQ

In memory, an IVT containing jump points {2, 5, 8, and 56} would look like this:

0x02\_05\_08\_38

## Stack

The stack is a table of values to which values can be pushed and popped (controlled by the instructions PUSH and POP, respectively), adding or removing them from the top of the stack. The stack consists of a 32 bit offset value, followed by the table itself. To push a value to the stack, consider this expression:

[R13] += 4

[[R13] + R13] = <ARG>

The offset value is increased by 4, and the argument is placed at the end of the table. To pop a value from the stack, consider this expression:

<R\_ARG> = [[R13] + R13]

[R13] -= 4

In memory, a stack containing the values {2, 5, 8, and 56} would look like this:

0x0C\_02\_05\_08\_38

# Code Segments

## Obtaining an absolute address

Labels were designed with the branch instruction in mind. As such, their occurrences are replaced with a relative address rather than an absolute one. However, conditions may arise where a developer needs to obtain an absolute address from a label. This is possible, through the use of R15, in only 1 instruction. Consider the following code segment:

**ADD** r1, r15, #:label:

After this instruction executes, the absolute address of :label: will be stored into memory. The artifact “#:label:” will be replaced by the distance to the label during assembly. By adding the current position, you can apply the offset obtain its true location.