# Segmentation

Table888seg contains sixteen segment registers. A number of segment registers are dedicated to specific uses and take the place of dedicated registers found in other architectures. Segment registers are comprised of a program visible portion and an invisible portion. The visible portion of the segment register is a 24 bit selector. The selector is used to access a segment description entry in the global or local segment descriptor tables.

Selector Format:

|  |  |  |
| --- | --- | --- |
| RPL3 | TI | Index20 |

RPL: request privilege level

TI: table indicator 0=global,1=local

Index: index into the global or local descriptor table

Whenever the selector value is loaded into a segment register, the entire segment descriptor from the global or local table is loaded into hidden registers. The descriptor controls the accessibility and the location of the segment in memory. The format of a data descriptor is shown below:

## Data Segment Descriptor Format

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P | ~ | ~ | 1 | X | D | W | A | Upper Bound63..8 |
| DPL3 | | | ~5 | | | | | Lower Bound63..8 |

P: present bit

DPL: descriptor privilege level

D: expansion direction

W: Writeable

A: accessed

X: executable (0 for data)

Note that the descriptor format allows for memory resolution only down to a 256 byte page of memory. Also note that the memory region is identified by an upper and lower bound, and not a base and limit.

If an address generated is outside of the bounds specified by the segment descriptor, an out of bounds exception will occur.

Unlike the 80x86 series, there is no local descriptor table register. Instead the local descriptor table is represented by segment register eleven. In a similar vein the task register is simply segment register number twelve.

## Identifying the Segment in Use

A bit-field in the load / store instructions identifies which segment register to use during address formation for data addresses. For push / pop operations segment register #14 (the stack segment) is always used. For code addresses segment register #15 is always used.

* If segmentation is not desired then segmentation can effectively be ignored by setting all the segment lower bound registers to zero and upper bounds to 64’hFFFFFFFFFFFFFFFF.

## Segment Usage Conventions

Segment register #15 is the code segment (CS) register. All program counter addresses are formed with the code segment register unless the upper 48 bits of the address are zero in which case the code segment is ignored. If the program counter is used in a load / store instruction, then the code segment register is used as the base by default.

Segment register #14 is the stack segment (SS) register by convention. The stack segment is used for operations involving the stack, PUSH, POP, JSR, or RTS operations. If the base pointer or stack pointer registers are specified in a load / store instruction then the default is to use the stack segment as a base.

Segment register #1 is the data segment (DS) by convention.

Segment register #13 is the volatile data segment (VDS). Addresses formed using this segment register bypass the data cache.

Segment register #12 is task state segment register. This segment register points the task state save area. If the task register is used by a load / store instruction then the task state segment is used by default for address formation.

Segment register #11 is Table888’s equivalent to the LDT register in the 80x86 series processors.

#### Segment Registers

|  |  |  |  |
| --- | --- | --- | --- |
| Num |  | Long name | Comment |
| 0 | NS | NULL segment | by convention contains zero |
| 1 | DS | data segment | by convention |
| 2 | TS | thread storage | by convention |
| 3 | BSS | BSS segment | by convention |
| 4 | RS | read only segment | by convention |
| 5 | ES | extra segment | by convention |
| … |  |  |  |
| 11 | LDT | local descriptor table | used by hardware to look up local descriptors |
| 12 | TSS | task state segment | associated with tr |
| 13 | VDS | volatile data segment | bypasses the cache |
| 14 | SS | Stack segment | by convention (associated with sp, bp) |
| 15 | CS | Code segment | always used for code addressing |

## Software Support

Segment registers may only be transferred to or from one of the general purpose registers. The mtseg and mfseg instructions can be used to perform the move. When the mtseg instruction is executed, the segment description is loaded from the descriptor table.

## Resolution

Segments are resolved down to 256 byte page in size.

## Address Formation:

Non-segmented address bits 0 to 7 pass through the segmentation module unchanged. Address bits 63 to 8 are added to the contents of the segment register to form the final segmented address. Note that there is no shift associated with the segment addition.

|  |  |
| --- | --- |
| Address[63:8] | Address[7:0] |
| + | + |
| lower bounds value[63:8] | 008 |
| = | |
| Segmented address[63:0] | |

## Selecting a segment register

A bit-field in the instruction identifies which segment register to use. This selection applies to data addresses only. Code addresses always use segment register #15 – the code segment. The segment register field is limited to two bits so that space in the instruction isn’t wasted. Normally for data access one of three segments is desired: the data segment, the bss segment or the stack segment. If another segment register is needed it may specified with a prefix instruction.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Displacement14 | Sg2 | Rt8 | Ra8 | 828h | LC Rt,d16(Rn) |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Of4 | Sg2 | Sc2 | Rt8 | Rb8 | Ra8 | 898h | LBU Rt,o6(Ra+Rb\*sc) |

|  |  |  |
| --- | --- | --- |
| Sg2 Field | Seg Reg # |  |
| 0 | 1 | default segment (Data segment) |
| 1 | 3 | BSS segment |
| 2 | 5 | Extra segment |
| 3 | 14 | Stack Segment |

This field may be overridden using an immediate prefix instruction.

Note that the default segment varies depending on the Ra register used in address formation. . The register and associated segment register are outlined in the table below.

### Default Segment Register Used in Address Formation

|  |  |  |  |
| --- | --- | --- | --- |
| Reg |  | Associated Segment Register |  |
| r0 |  | ds |  |
| r1 |  | ds |  |
| … |  | ds |  |
| r251 |  | ds |  |
| r252 | tr | tss |  |
| r253 | bp | ss |  |
| r254 | pc | cs |  |
| r255 | sp | ss |  |

For example: LW R1,64[pc] will use the code segment as a default in generating the address because the program counter register is specified for the address formation.

The default segment register used in address formation changes based on the Ra register field

For load / store instructions immediate prefixes specify a segment register, overriding the segment register specified in the following instruction.

|  |  |  |  |
| --- | --- | --- | --- |
| Sg4/Cn4 | Constant28 | FDh | IMM1 |
| Segreg4 | Constant28 | FEh | IMM2 |

Example 1:

The IMM1 prefix specifies a segment register overriding the specification in the following instruction. The constant field is extended by 28 bits.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Segreg4 | Constant28 | | | | FDh | IMM1 |
| Displacement14 | | Sg2 | Rt8 | Ra8 | 828h | LC Rt,d16(Rn) |

Example 2:

The IMM2 prefix specifies a segment register overriding the specification in the following instruction. The IMM1 prefix adds 32 bits to the constant. The constant field is extended by 60 bits.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Constant32 | | | | | | | FDh | IMM1 |
| Segreg4 | | Constant28 | | | | | FEh | IMM2 |
| Of4 | Sg2 | | Sc2 | Rt8 | Rb8 | Ra8 | 898h | LBU Rt,o6(Ra+Rb\*sc) |

## Power-up State

On reset the value in the segment registers are undefined; the reason why the boot area is required because the code segment may contain random data. Note also that memory data cannot be properly accessed until the data segment is setup. Note that the processor begins executing instructions out of the non-segmented code area as the reset address is 64’h000000000000FFF0. One of the first tasks of the boot program would be to initialize the segment registers to known values. The segment register must be setup to perform data accesses properly.

## Non-Segmented Code Area

The address range defined as 64’h00000000000xxxx is a non-segmented code area. This area allows the operating system to work without paying attention to the code segment, so that the operating system may boot.

## Changing the Code Segment

The code segment may be changed by a JMP or JSR instruction that uses a jump prefix instruction. The jump prefix specifies the selector for the target code segment, plus eight additional target address bits.

Instruction Format with Selector Prefix:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| RPL3 | TI | Index20 | Addr8 | 618h | JSP |
| Address32 | | | | 518h | JSR address |

Note that the JMP or JSR instructions can only transfer to code at the same privilege level. In order to call code at a higher privilege level a jump gate must be used. A jump gate routine may be called using the JGR (jump gated routine) instruction. The JGR instruction specifies only the selector of the gate to use.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| RPL3 | TI | Index20 | ~8 | 578h | JGR |

# Protection

Table888’s protection mechanism is similar to that of the 80x86 series. If you are comfortable with the 80x86 protection mechanism then Table888’s mechanism will seem familiar. The format of descriptors is different but the fields are similar. Table888 has eight protection ring levels.

## Protection Rules

Code cannot access data at more protected level. This is checked by comparing the processor’s current privilege level to the privilege level of the segment that is pending access. The check is performed when a segment register is loaded using the mtseg instruction.

Code at a high protection level cannot call code at a lower level. This is checked when a jump to subroutine or jump instruction is executed with a jump selector prefix.

The code segment may only be loaded with code type descriptors. If an attempt is made to load a data descriptor into a code segment a segment type violation exception occurs.

The data segments may only be loaded with data type descriptors. If an attempt is made to load a code segment into a data segment a segment type violation exception occurs.

Interrupts

|  |  |  |  |
| --- | --- | --- | --- |
| Vecno |  | Description |  |
| 0 |  |  |  |
| 1 |  |  |  |
| 2 |  | FMTK Scheduler |  |
| 449 | KRST | Keyboard reset interrupt |  |
| 450 | MSI | Millisecond Interrupt |  |
| 451 | TICK | FMTK Tick Interrupt |  |
| 463 | KBD | Keyboard interrupt |  |
| 500 | SBV | segment bounds violation |  |
| 501 | PRIV | privilege level violation |  |
| 502 | STV | segment type violation |  |
| 503 | SNP | segment not present |  |
| 508 | DBE | data bus error |  |
|  |  |  |  |
| 510 | NMI | Non-maskable interrupt |  |
|  |  |  |  |

The program counter in Table888 was limited to 32 bits; all the bits the external bus implemented. In Table888seg the program counter is 40 bits. When combined with the selector addressing up of up to 61 bits is possible. Program addresses are formed from the addition of the code segment register to the program counter.

Virtual addresses are composed of a selector and an offset that when combined together form a 64 bit pointer. The index portion of the selector ‘indexes’ into one of two tables, the GDT (global descriptor table) or the LDT (local descriptor table). The index part of the selector plus the table indicator can specify up to two million entries. When combined with a 40 bit offset, addressing of up to 61 bits is possible.

Selector Format:

|  |  |  |
| --- | --- | --- |
| RPL4 | TI | Index19 |

Sixty-four bit pointer:

|  |  |
| --- | --- |
| Selector24 | Offset40 |

For code addresses, the selector may be specified directly before a jump or jump-to-subroutine instruction via a selector prefix instruction.

Segment descriptors contain lower and upper bounds. The resolution of segments is 256 byte pages.

Data Segment Descriptor

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P | ~ | ~ | 1 | X | D | W | A | Limit55..0 |
| DPL4 | | | | Base59..0 | | | | |

P: present bit

DPL: descriptor privilege level

D: expansion direction

W: Writeable

A: accessed

X: executable (0 for data)

Code Segment Descriptor

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P | ~ | ~ | 1 | X | C | R | A | Limit55..0 |
| DPL4 | | | | Base59..0 | | | | |

P: present bit

DPL: descriptor privilege level

C: conforming

R: Readable

A: accessed

X: executable = 1

System Segment Descriptor

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P | ~ | ~ | 0 | 0 | | 0 | T | T | Limit55..0 |
| DPL4 | | | | | Base59..0 | | | | |

P: present bit

DPL: descriptor privilege level

TT:

1=inactive TSS

2=LDT

3=active TSS

## Call Gates

Call gates are used to switch to a higher level of privilege. Call gates in Table888 does not touch the stack. No parameters are copied between stacks. Parameters are passed through the gate entirely in registers. It is left up to software to switch stacks if required. Stack switching can be done relatively fast by software using existing instructions as the following example shows.

|  |
| --- |
| SomeCallgateFn:  lw r251,[sp] ; get return address  ; First, save the current stack pointer and segment  shr r250,r251,#61 ; get originating privilege level  ; The stack pointer will be invalid for a little bit, so disable interrupts  sei  sw sp,TCB\_SP0Save[tr+r250\*8] ; save original stack pointer  mfseg sp,ss  sw sp,TCB\_SS0Save[tr+r250\*8]  ; Second, setup a stack at this privilege level  mfseg r250,cs ; get the current privilege level from the cs selector  shr r250,r250,#61  lw sp,TCB\_SS0Save[tr+r250\*8]  mtseg ss,sp  lw sp,TCB\_SP0Save[tr+r250\*8] ; get stack pointer according to privilege level  cli  push r251  ;...  ;now use the stack  ;...  ; Save the current stack back, if needed (may not be necessary)  mfseg r250,cs ; get current privilege level  shr r250,r250,#61  sw sp,TCB\_SP0Save[tr+r250\*8] ; store the stack pointer in the tss.  mfseg r249,ss  sw r249,TCB\_SS0Save[tr+r250\*8] ; store stack segment in tss  ; load back the original stack  pop r251  shr r250,r251,#61 ; get originating privilege level  sei  lw sp,TCB\_SS0Save[tr+r250\*8] ; get back segment register  mtseg ss,sp  lw sp,TCB\_SP0Save[tr+r250\*8] ; get appropriate stack pointer from tss  cli  rts |

Call Gate Descriptor

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P | ~ | ~ | 1 | | X | C | R | A | ~27 | ~5 | Selector23..0 |
| DPL4 | | | | Offset59..0 | | | | | | | |

## Task Gates

Where are the task gates ? Task gates on the 80x86 series have proven to be too much of a solution and too inflexible for practical usage. There are no operating systems that make use of them. I tried for a while a number of years ago to implement a simple task switcher using task gates. I always found myself wanting to ‘wedge into’ the task switch mechanism. Hopefully learning from the experience of others, Table888 does not implement task gates.

Table888 provides a fast way to save groups of registers via the store multiple registers and load multiple registers instructions. These instructions are components that could support an operating system.

Interrupt Descriptor Table Entries

The interrupt descriptor table contains only two types of gates (interrupt, or trap). Interrupt descriptor table entries are eight bytes in size with the following format:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P | ~ | ~ | 0 | 0 | 1 | 1 | T | Interrupt Code Segment Selector24 |
| DPL4 | | | | Offset28 | | | | |

T = 1: Trap Gate

T = 0: Interrupt Gate

The difference between a trap and an interrupt is that an interrupt automatically masks further interrupts from happening. A trap does not affect the interrupt flag.

Note that the interrupt routine must be located within the first 256MB region of the segment because only a 28 bit offset is allowed for.

IDT/GDT Address Record

The IDT/GDT address record is consists of four 32 bit words, and is the load / store format of the IDT/GDT register.

|  |  |  |
| --- | --- | --- |
| Upper Bound63..32 | | C |
| Upper Bound31..8 | ~8 | 8 |
| Lower Bound63..32 | | 4 |
| Lower Bound31..8 | ~8 | 0 |

The interrupt and global descriptor tables should be loaded using the null segment (NS) in order to use a physical address rather than a segmented one. These tables need to be setup prior to the segment register loading.