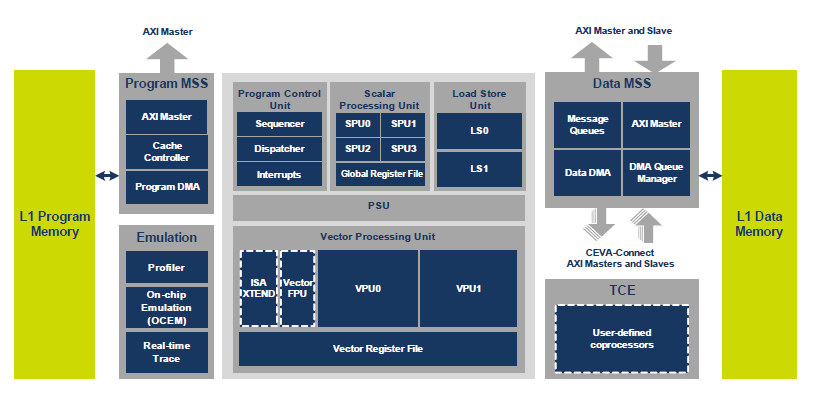
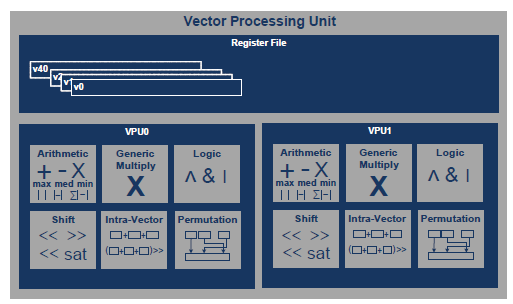
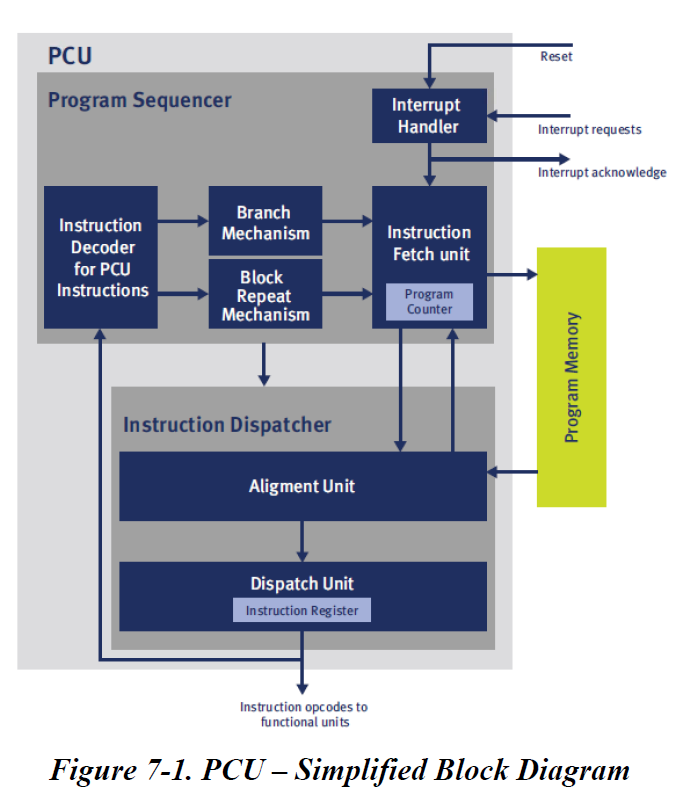
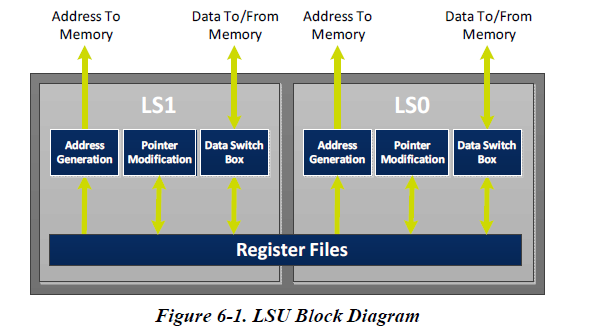
CEVA DSP Notes

# Overview









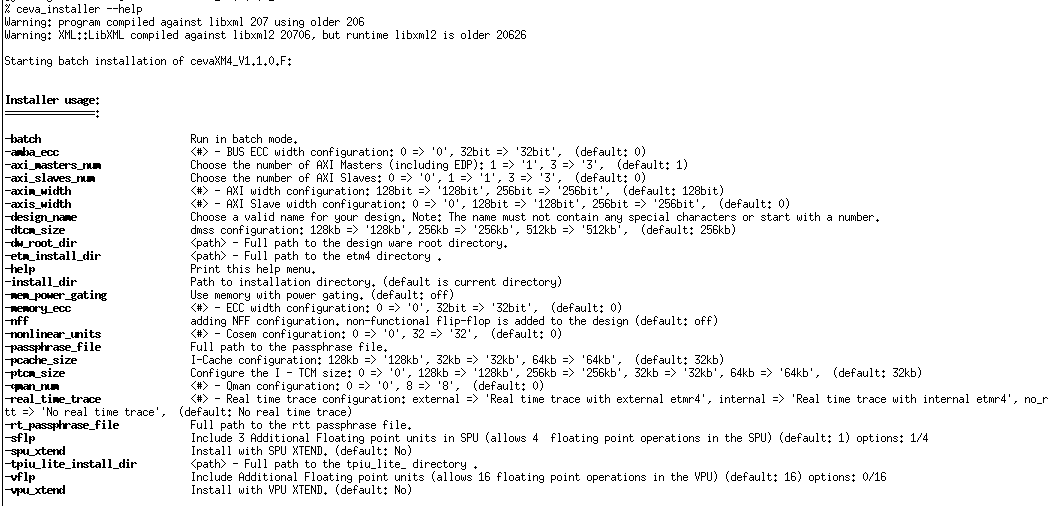
# 安装指南

% mkdir work; chdir work

% copy /LnxShare/eda/Ceva/update\_v1.1.1.F/cevaXM4\_V1.1.1.F\_passphrase

% tar zxfv /LnxShare/eda/Ceva/update\_v1.1.1.F/cevaXM4\_V1.1.1.F.tgz

帮助中列出的所有的可以config的部分。



范例：

% ceva\_installer \

-passphrase\_file ../cevaXM4\_V1.1.0.F\_passphrase \

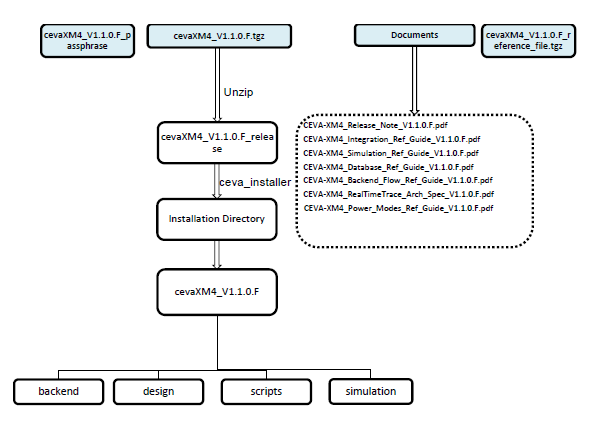
-dw\_root\_dir /tools/linux/synopsys/2015.06-SP3-1/syn/dw/sim\_ver \

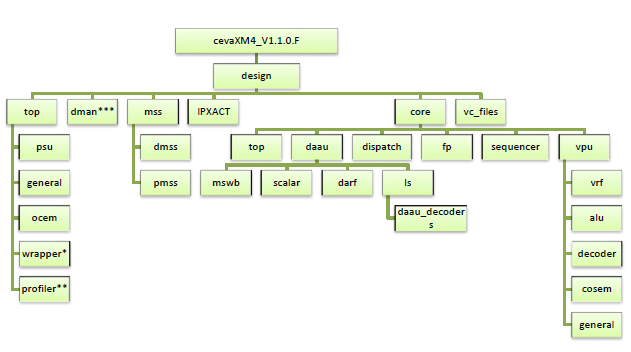
-dtcm\_size 128kb \

-memory\_ecc 32bit

产生的top level名为<design\_name>\_cevaxm4

目录结构如下：





% cd cevaXM4\_V1.1.0.F\_release

下面用$CEVAXM4表示config好的一个工作目录

% module load vcs verdi ceva

% cd $CEVAXM4

% source ./scripts/setenv.csh

% cd simulation/asm/work

如果vcs运行不能checkout license，试试

% setenv LM\_LICENSE\_FILE 27000@eda-11:$LM\_LICENSE\_FILE

% ceva\_sim –sim vcs –t ceva4\_vpu0\_bypss –debug

完整的命令行参数可以用 –help 获得

查看所有release的test可以用 –show tests

开启waveform dump使用-verdi选项

最终调用./simv –l vcs.log –ucli –I vcs.cmd

在vcs.cmd中开启dump

fsdbDumpfile Verilog.fsdb

fsdbDumpvars 0 cevaxm4\_sim\_top

run

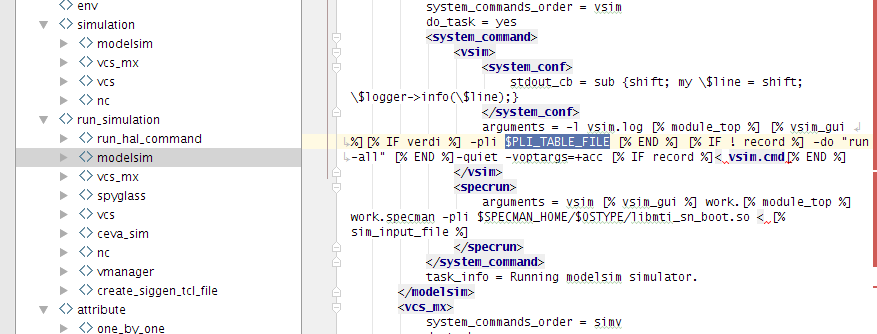
for modelsim

% setenv LD\_LIBRARY\_PATH $NOVAS\_HOME/share/PLI/MODELSIM/LINUX64:$LD\_LIBRARY\_PATH

% setenv PLI\_TABLE\_FILE “novas\_fli.so”

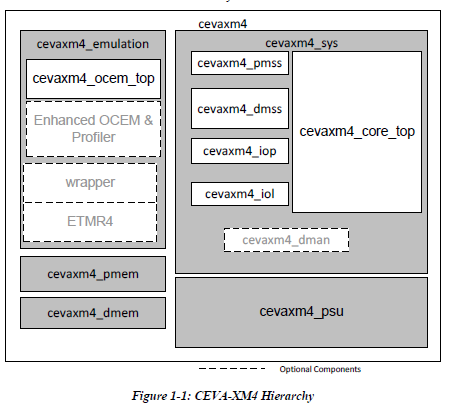
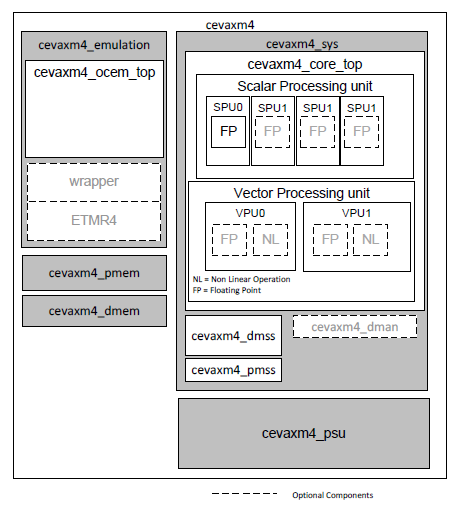
仿真的命令行参数在<ceva\_root>/script/conf/cevaXM4.cf

Xml file中寻找modelsim





# RTL tree

cevaxm4\_sim\_top

--<design>\_cevaxm4

-- cevaxm4\_psu

?? clock gater and PG …

-- cevaxm4\_sys

--dman DMA Manager

--qman

--bman

--core\_top (cevaxm4\_core\_top)

--daau

--arf

--daau\_to\_vpu0|1\_bypass

--ls0|1

--Is

--dmswb: Memory Switch Box

--scalar0 (cevaxm4\_spu)

--scalar123

--scalar\_fp

--scalar: decode+execute

--src0|1\_sops

--pdispatch

--psequencer

--vpu\_top

--vld

--vpu0|1 (cevaxm4\_vpu)

--vpu\_dls\_en\_sel\_gen

--vpu\_modv\_reg

--vpu\_vrf

--vst

--dmss Data Memory Sub-System

--4xx\_rab\_ls0|1 cache Read Address Buffer

--adec\_ls0|1 Address DECoder

--cpm

--dacu\_ls0|1 Data Access & Control Unit

--ddma DDMA TCM write module

--ddma\_debug

--edap External Device Access Port (as AXI slave)

--edp External Port

--eos External Write Output Stage

--hist HISTogram, VPU VHIST RMW access to TCM

--mcci Multi-Core Command Interface

--rd\_vu\_ls0|1 byte read buffer, VU aligner

--sys\_wdog Watch Dog

--wb Write Buffer

--iol IO Logic Block

--iop APB3 Interface

--pmss Program Memory Sub-System

--pmc Program Memory Controller

--iarb Internal Arbiter

--earb External Arbiter

--epp Program External Port

--iacu Instruction Access and Control Unit

--l1pc L1 Program Cache

--pipe

--rab Read Address Buffer

--cpm

-- cevaxm4\_emulation

--trace: cevaxm4\_wrapper

--[ETMR4]: 仅当设置为external时才会有instance

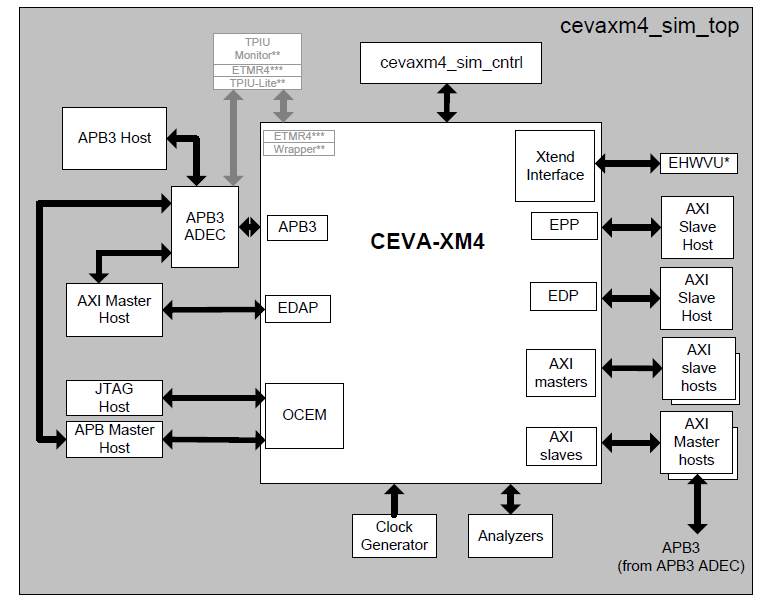
--profiler: addr\_[high|low]\_[comp|gater]0~7, prf\_counter0~7, clock counter.

--ocem: apbs/break/control/jtag

-- cevaxm4\_pmem

-- cevaxm4\_dmem

# Interfaces



除了EPP/EDP是必须的，可以选用最多两个AXI Master，而且配置是两个都有或都没有。

软件可以选择去哪个端口，各端口之间是没有顺序的。

除了必须得EDAP slave，可以选用最多3个AXI Slave。多半是为了外部DMA使用。如果启用内部DMA，那么只有Safety原因才会用。

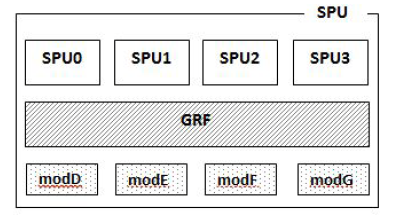
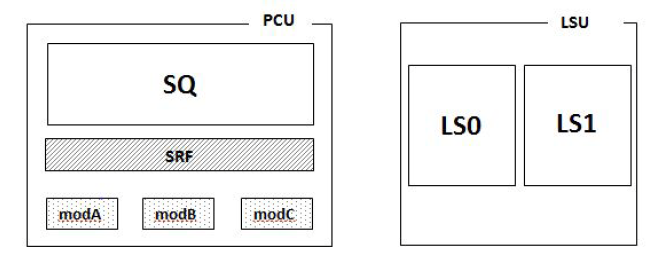
EPP在正常的操作中是只读端口，但是在OCEM下允许写入外部程序区。所以是完整的R/W AXI4。仿真中也通过该端口装载程序。（不可以只通过data端口写吗？调试中不可以改动data端口配置的现状）

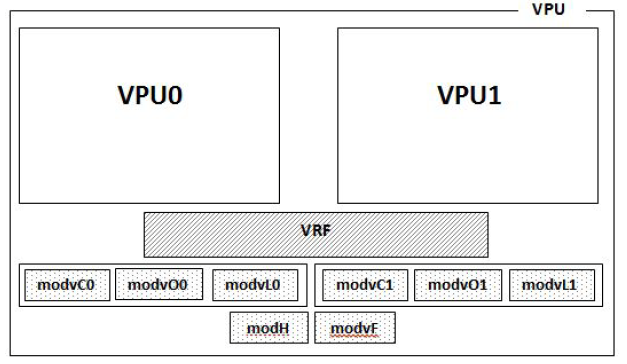
DSP编程自己的CPM寄存器是通过IO指令，带上CPM标志。访问别人的CPM寄存器只能通过EDAP/AXI Slave

|  |  |  |  |
| --- | --- | --- | --- |
| Name | D | W | Comments |
| EPP AXI | M | 128 | Program Fetch |
| div\_en\_epp | I | 1 | slow clock enable for EPP AXI interface |
| EDP AXI | M | 128/256 | Data Access |
| div\_en | I | 1 | Slow clock enable for AXI interface: EDP, EDAP |
| AXIM0可选 | M | 128 |  |
| div\_en\_axim0可选 | I | 1 | slow clock enable |
| AXIM1可选 | M | 128 |  |
| div\_en\_axim1可选 | I | 1 | slow clock enable |
|  |  |  |  |
| EDAP AXI | S | 128 |  |
| div\_en | I | 1 | Slow clock enable for AXI interface: EDP, EDAP |
| AXIS0可选 | S | 128/256 |  |
| div\_en\_slv0可选 | I | 1 | slow clock enable |
| AXIS1可选 | S | 128/256 |  |
| div\_en\_slv1可选 | I | 1 | slow clock enable |
| AXIS2可选 | S | 128/256 |  |
| div\_en\_slv2可选 | I | 1 | slow clock enable |
|  |  |  |  |
| IOP APB3 | M | 32 | I/O 指令访问 |
| div\_en\_iop | I | 1 | slow clock enable |
| OCEM APB | S | 32 | Debug Slave Port |
|  |  |  |  |
| MCCI & Snoop |  |  |  |
| mcci\_mes\_int | O | 1 | MCCI interrupt request |
| mcci\_rd\_ind\_r | O | 32 | MCCI read indicator |
| snoop\_sn\_int | O | 1 | Snoop interrupt request |
|  |  |  |  |
| DMA Manager可选 |  |  |  |
| qman\_irq | O | 1 | Violation indications interrupt |
| qman\_semaphore\_req | O | 16 |  |
| qman\_semaphore\_grant | I | 16 |  |
| qn\_desc\_en | I | 16 | QMAN enable descriptor increment signal |
|  |  |  |  |
| XTEND SPU |  |  |  |
| scalar\_dst\_data | I | 32 | Result at stage V1 |
| xh\_src0|1\_m\_r | O | 32 | Src0|1 data at stage E1 |
| xh\_pr\_en\_e1 | O | 1 | Predicate valid indication |
| xh\_pir\_a1\_r | O | 23 | Instruction Register at Stage A2 |
| xh\_pir\_valid\_a1\_r | O | 1 | Instruction Valid at stage A2 |
| xh\_pext\_a1\_cr | O | 26 | Extension Word at stage A2 |
| xh\_pext10\_valid\_a1\_r | O | 1 | Dispatcher 10 extension valid indication |
| xh\_pext26\_valid\_a1\_r | O | 1 | Dispatcher 26 extension valid indication |
|  |  |  |  |
| XTEND VPU |  |  |  |
| dest\_v4\_r | I | 256 | Destination Result |
| en\_v4\_r | I | 32 | Destination Enable |
| vpu\_xtend\_src0|1\_e2 | O | 256 | Src at Stage E2 |
| vpu\_xtend\_vpr\_e2 | O | 32 | Predicate vector indication |
| vpu\_xtend\_pir\_e1\_r | O | 23 | Instruction Register at Stage E1 |
| vpu\_xtend\_pext\_e1\_r | O | 26 | Extension Word at stage E1 |
| vpu\_xtend\_pir\_valid\_e1\_r | O | 1 | Instruction Valid at stage E1 |
| vpu\_xtend\_pir10\_valid\_e1\_r | O | 1 | Dispatcher 10 extension valid indication |
| vpu\_xtend\_pir26\_valid\_e1\_r | O | 1 | Dispatcher 26 extension valid indication |
|  |  |  |  |
| 中断源 |  |  |  |
| Int0-2 | I | 1 | Maskable interrupt 0-2 |
| nmi | I | 1 | Non-maskable interrupt |
| vint | I | 1 | Request signal for vector interrupt |
| vector | I | 32 | Vectored interrupt |
| uop\_int\_r | O | 1 | Undefined Operation |
| seq\_int[0-4]\_ack\_n\_r | O | 1 | Maskable interrupt #, stretched ack low |
| seq\_bp\_ack\_n\_r | O | 1 | BP |
| seq\_nmi\_ack\_n\_r | O | 1 | Non maskable |
| seq\_vint\_ack\_n\_r | O | 1 | Vector interrupt |
| epp\_wdog\_viol\_r | O | 1 | Watch Dog Timeout |
| edp\_wdog\_viol\_r | O | 1 | Watch Dog Timeout |
| iop\_wdog\_viol\_r | O | 1 | Watch Dog Timeout |
| axim#\_wdog\_viol\_r | O | 1 | Watch Dog Timeout |
| [Optional ECC interrupt] | O | 1 |  |
| code\_ecc\_int\_r | O | 1 | One ECC error |
| code\_ecc\_2err\_int\_r | O | 1 | Two ECC error |
| code\_tag\_ecc\_int\_r | O | 1 | ECC stretched error interrupt for tag array |
| eccerr\_int\_r | O | 1 | Double Error detected on L1DM ECC |
| ecccor\_int\_r | O | 1 | Correctable error detected on L1DM ECC |
|  |  |  |  |
| Semaphore Interface |  |  |  |
| seq\_trp\_srv\_r | O | 1 | Trap service routine indication, set when core is in a trap |
| OCEM |  |  |  |
| ext\_bp1\_req | I | 1 | External Breakpoint request #1 |
| ext\_bp2\_req | I | 1 | External Breakpoint request #2 |
| bs\_reg\_tdo | I | 1 | Boundary Scan register TDO |
| tck/tdi/tms | I | 1 | JTAG |
| jt\_ap | I | 1 | OCEM scan chains access type JTAP/APB |
| ocm\_tdo\_r | O | 1 | JTAG TDI |
| ocm\_tdo\_oen\_r | O | 1 | JTAG TDO enable |
| ocm\_jtag\_state\_r | O | 4 | JTAG state |
| ocm\_gp\_out\_r | O | 4 |  |
| ocm\_ext1\_ack\_r | O | 1 |  |
| ocm\_ext2\_ack\_r | O | 1 |  |
| ocm\_core\_rst\_r | O | 1 |  |
| ocm\_debug\_r | O | 1 |  |
|  |  |  |  |
| PSU and AXI Low Power |  |  |  |
| csysreq | I | 1 | Request to switch to Light-sleep and Standby |
| core\_rcvr | I | 1 | Recover from Light-sleep and Standby |
| stop\_sd | I | I | Stop core from shutdown |
| psu\_cactive\_r | O | 1 |  |
| psu\_csysack\_r | O | 1 |  |
| psu\_rtck\_r | O | 1 | Return test clock (tck synced to free clk) |
| psu\_dsp\_idle\_r | O | 1 | clock and MSS clock可关 |
| psu\_core\_idle\_r | O | 1 | Clock from PSU to core shut down |
| psu\_pshtdwn\_\_r | O | #PD | Power shutdown request per unit domain  Core+MSS/Emulation/DTCM(4)/PTCM/P$ |
| psu\_sys\_pshtdwn\_r | O | #BK | Memory retention mode, indication for Deepsleep (DTCM(4)/PTCM/P$) |
|  |  |  |  |
| DFT |  |  |  |
| testmodep | I |  |  |
| tst\_gatedclock | I |  |  |
| tst\_mem\_gatedclock | I |  |  |
| bist\_prog\_in | I | # |  |
| bist\_data\_in | I | # |  |
| bist\_prog\_out | O | # |  |
| bist\_data\_out | O | # |  |
| bist\_dcdata\_out | O | # |  |
|  |  |  |  |
| DDMA Debug |  |  |  |
| ext\_ddma\_dbg\_match\_ack | I | 1 | External ack for DDMA debug match |
| next\_ddma | I | 1 | External control over DDMQ |
| ddma\_dbg\_match\_r | O | 1 | DMSS DMA addr/data value match |
| gvi\_r | O | 1 | General Violation |
|  |  |  |  |
| Operation Mode Support |  |  |  |
| ext\_pv | I | 1 | External Permission Violation |
| ext\_vom | I | 2 | External Permission Violation operation mode |
| seq\_om\_r | O | 2 | Operation Mode |
| seq\_pi\_out\_r | O | 1 | Permission Interrupt output |
|  |  |  |  |
| General |  |  |  |
| ceva\_free\_clk | I | 1 | Root clock |
| ceva\_###\_rst\_n | I | 1 | core/sys/global/ocem |
| ceva\_###\_wdog\_clk | I | 1 | sys/epp/edp/iop/am0/am1 |
| boot | I | 1 | Boot request, set only during reset |
| external\_wait | I | 1 |  |
| mcache\_invalidate\_strap | I | 1 | Memory cache invalidate strap |
| gp\_in | I | 32 | GPIO addr=0x34 MSS\_GPIN |
| gpout | O | 32 | GPIO addr=0x38 MSS\_GPOUT |
| core\_id | I | 32 |  |
| psu\_core\_wait\_r | O | 1 | Wait indication from core |
| psu\_mapv\_r | O | 1 | Access protection violation |
| bus\_parity [ECC] | I | 1 |  |
| acu\_lock | I | 1 | DACU and IACU lock indication. Cannot change configuration of DACU and IACU |
| acu\_slv\_acc | I | 1 | 1: only external can change config  0: DSP in Supervisor mode can change |
|  |  |  |  |
| Verification Indication |  |  |  |
| cverbit | O | 1 | Error bit summary |
| seq\_eotbit\_r | O | 1 | End of Test bit |

# ISA

## Overview





变长指令集16/32/48/64，VLIW，package

每条都带有predicate (IA-32)

Register ID 5bit

System Encoding 8bit

ISA 文档描述, unit.instruction args …

* {指令的开关}
* [可选项]
* arA :Ａ是合法的寄存器标号（０－３１）
* 假操作数: predicate, post-modification …

## Register file

* GRF General Register File
* ARF Address Register File
* SRF System Register File
* SPR Special Purpose Register
* VRF Vector Register File
* PRF Predicate Register File

GRF: 32b

* r0~31

VRF: 256b，可以看作32c(32x8b), 16s(16x16b), 8i(8x32b)

* Accumulator: 两个VRF并成一个512b，可以看作32s(32x16b), 16i(16x32b)
* vr0~vr39
* vl: 0~23
* v: 0~31
* vacc: 24~39 8个
* vacch: 32~39 4个
* @vpu\_top/vpu\_vrf/vrf0\_23
* @vpu\_top/vpu\_vrf/vrf24\_31
* @vpu\_top/vpu\_vrf/vrf32\_39
* @vpu\_top/vpu\_modv\_reg/modvl0~1(32)
* @vpu\_top/vpu\_modvfpr(32)

PRF：

* Scalar 1b pr0~14 d####@daau/arf
  + pr15 is constant 1, always true
* Vector 32b vpr0~6 @vpu\_top/vpu\_vrf/
  + vpr7 is constant 0xffff\_ffff

ARF(32b) d####@daau/arf

* sp
* step0~7
* modeD/E/F/G
* modu0~3 地址的wrap
* modv0~3 2D地址的配置

SRF(32b) p\_##### @ psequencer

* pc where?
* retreg @pbranch
* retregi interrupt, maskable
* retregn non-maskable interrupt
* retregb trap(sw) or BI(HW breakpoint)
* lc0~3 loop counter @pblock\_repeat
* lci0~3 loop counter init
* lcstep0~1 post modify

SPR Special Purpose Register

* bknest0~2
* moda
* modb
* modc
* modh @pexcept\_handle

## SPU

**Logic**:

and/nand/or/nor/andnot/not/xor/xnor

**Bit**:

bitdup: 每个bit扩张成两比特，8/16 => 16/32

cntbits: 计数0或1的个数

extract: 提取指定位置的指定宽度的位到dst

insert: 在dst指定位置放入指定宽度的位

ffb: find first bit，可指定找0/1，从MSB/LSB开始，找连续的符号位。

tst: 测试bit是否为0/1，可以选择某一比特，或是选择一个mask，结果可以写入互补的两个寄存器

lbf: 立即数写入flag

flcopy: flag拷贝到predicate

prm: Predicate Register Manipulation, 对最多3个predicate做逻辑操作，结果写入另外一个（或加一个互补）的predicate寄存器。

shiftl:

shiftr:

**Alu**:

Add {sat, cf}

Sub {sat, cf}

Shift\_add a<<b+c

Shift\_sub c-a<<b

Abs

Neg

Inc

Dec

Cmp {lt|le|gt|ge|eq|neq}

Max

Min

Mpy

Mac

**Float**:

fpadd

fpsub

fpcmp

fpmpy 无乘加

fpcombine 2register + 1predicate => 拼接出一个float

fpextract

fp2int

int2fp

fp2usint

usint2fp

mov

pcu register/vflag/modDEFG/pc/divtemp

r8to15: r8~r15 放入v39[0]~[7]

r16to23: r16~r23放入v38[0]~[7]

step0to7: step0~7放入v37[0]~[7]

movselect: z = c?a:b

verifeqs 比较决定是否设置verbit，协助仿真。

## VPU

|  |  |
| --- | --- |
| Opcode | Function |
| Logic |  |
| vand |  |
| vnand |  |
| vor |  |
| vnor |  |
| vxor | 没有nxor |
| vnot |  |
|  |  |
| Bit manip |  |
| vffb(a) | 可指定找0/1，从MSB/LSB开始，找连续的符号位。 |
| vcntbits(a) | 计数0或1的个数 |
| vsplit(a,b) | 将a分解为低b位和剩余高位部分，结果写入两个vreg |
| vshiftl(a, b) | b若为负值就变成右移 |
| vshiftr(a, b) | a的值决定是逻辑或是算术的移位 |
| vaccshiftr(a,b) | a/z是vacc, 有两个，没有左移 |
|  |  |
| Misc |  |
| vmov | 非常丰富的变种，除了基本的vr之间的move，还有   * 立即数/scalar reg 付给所有vector分量，且做格式转换 * 将scalar reg付给vector指定（vB，imm）分量或者反之 * 初始化为连续整数 * 初始化为A 的4bit，初始化8个，1xxx=>10xxx (perm) * 初始化为A,B的4bit，初始化16个，1xxx=>10xxx (perm) * 写入flag * r8to15, v39写入r8~15 * r16to23, v38写入r16~23 * s0to7, v37写入step0~7 * vz[i] = rc ? rb : va[i] |
| vaccmov | 写入acc |
| vmovselect | z[i] = c[i] ? a[i] : b[i], vprh(使用c的高位) |
| vperm | {b, a}[c[i]] = z[i], 任意的swizzle。c是uc32且必须在规定范围，mask掉超出的 |
| vlut | {b, a}[c[i]] = z[i], 对c没有约束，超出就不对结果赋值 |
| vcast | A or {A, B} 格式转换=> Z, 相同格式就用vmov |
| vacccast | 同上，Z是acc |
| vinterleave | 将a，b的分量交织放入z，a b只能用一半的分量 |
| vinterleave3 | RGB合并的vector，分解出R vector，G vector，B vector |
| vdeinterleave | 取出输入vector的奇或是偶位置上的数给Z  需要调用两次完成interleave的逆操作 |
| vdeinterleave3 | R vector，G vector，B vector 交织成RGB合并的3vector长的数据付给3个vector  文档定义没有搞反吗？ |
| vhist | Histogram, 可以提供一个vector作为weight加在对应的bin上，否则只是加1 |
|  |  |
| ALU |  |
| vadd | A+B |
| vsub | A-B |
| vaccadd | A+{B,C} |
| vaccsub | A-{B,C} |
| vaddsat | A+B With saturate |
| vsubsat | A-B With saturate |
| vabssub | Abs(a-b) |
| vabssubacc | w = w + abs(a-b) |
| vneg | z=-a |
| vintrasum | z=sum(a[i]) (32b)  z[i]=sum(A[(i\*4+3):(i\*4)]) ; //a[i]为8b，4个加起来写入z(32b)  z[i]=sum(A[(i\*2+1):(i\*2)]) ; //a[i]为16b，2个加起来写入z(32b)  z[i]=sum(A[(i\*4+3):(i\*4)]) + sum(B[(i\*4+3):(i\*4)]); //a[i] b[i]为8b，4个加起来写入z(32b)  z[i]=sum(A[(i\*2+1):(i\*2)]) + sum(B[(i\*2+1):(i\*2)]); //a[i] b[i]为16b，2个加起来写入z(32b) |
|  |  |
| 整数乘法 |  |
| vmpy | 32x32 写入32，可选择结果的高或低的32  32x16 可选择是否右移16  8x8  16x8  B可以是reg  可以提供一个round点{rnd}, rD  同时可以要求做算术右移或逻辑右移{psl} post shift logic  {splitsrc} 低一半结果做A\*B，高一半结果是A\*C |
| vmpynorm | Z = ( (a\*b) + rnd) >> C |
| vmpyadd | Z = a\*b + c  Z = (a\*b + c) >> rd |
| vmac | w = w + a \* b  z = （c + a \* b）>> rd |
| vmac3 | w = w + a \* b + c \* d  z = e + a \* b + c \* d  z = (e + a \* b + c \* d) >> f  w = w + a \* cl + b \* ch  z = (e + a \* cl + b \* ch) >> f  w = w + a \* a + b \* ch  z = (e + a \* a + b \* ch) >> f |
| vmad | z = a \* b + c \* d  z = (a \* b + c \* d) >> f  z = a \* cl + b \* ch  z = (a \* cl + b \* ch) >> f  z = a \* a + b \* ch  z = (a \* a + b \* ch) >> f |
| vmpymat4x4 | 16bit A/B, 16=4x4 matrix, 结果是32bit，一次只能做8个结果  {high}决定作出高8个结果  {transpose}变换位置 |
| vaddmpy | (a+b)\*c  ((a+b)\*c) >> D |
| vsubmpy | (a-b)\*c  ((a-b)\*c) >> D |
| vaddmac | w = (w + (a+b)\*c)  z = (d + (a+b)\*c) >> e |
| vsubmac | w = (w + (a-b)\*c)  z = (d + (a-b)\*c) >> e |
| vgenmpys | Vector GENeric Multiply by Scalar E/F  w = w + (a+b)\*E[15:8] + (c+d) \* E[7:0]  z = (w + (a+b)\*E[15:8] + (c+d) \* E[7:0]) >> E[8:4]  w = F + (a+b)\*E[15:8] + (c+d) \* E[7:0]  z = (F + (a+b)\*E[15:8] + (c+d) \* E[7:0]) >> E[8:4] |
| vgenmpyv | Vector GENeric Multiply by vector, scalar E/F  w = w + (a+b)\*(c+d)  z = (w + (a+b)\*(c+d)) >> E[8:4]  w = F + (a+b)\*(c+d)  z = (F + (a+b)\*(c+d)) >> E[8:4] |
| 除法/开方 |  |
| vinv | 1/A = VrZ \* (2^(VaccZ-16))  16或32位无符号整数求倒数，结果的exp/mantissa写入两个寄存器。 |
| vsqrt | sqrt(A) = Z0 \* 2^(Z1-8) |
| vsqrti | 1/sqrt(A) = VrZ \* (2^(VaccZ-16)) |
|  |  |
| Compare |  |
| vclip(a,b,c) | 将a的值clip到[b，c]范围内 |
| vcmp | op : lt|le|gt|ge|eq|neq  vorop: and|or|xor  两操作数a op b  三操作数 a>=b, a<=c  两操作数+vpr, (a op b) vprop vpr  结果写入 vpr |
| vcmpacc | A, B(B可以是r或vr)比较结果写入vpr  成功比较  w[i]=w[i]+c[i]  w[i]=w[i]+rC |
| vcmpmov | A, B(B可以是r或vr)比较结果写入vpr  Z[i]=vpr[i] ? A[i] : B[i]  Z[i]=vpr[i] ? A[i] : B  Z[i]=vpr[i] ? C[i] : D[i]  Z[i]=vpr[i] ? C[i] : D |
| vabscmp | vcmp(abs(a), abs(b) =>vpr |
| vabscmpmov | vcmp(abs(a), abs(b) =>vpr  Z0[i]=vpr[i] ? A[i] : B[i]  Z1[i]=vpr[i] ? B[i] : A[i] |
| vmin | 2或3个操作数vr，per component 做min，写出到z的对应的component |
| vmax | 2或3个操作数vr，per component 做max，写出到z的对应的component |
| vintramin | 对操作数所有的component求min，写出到rZ，并在对应的vpr上置位 |
| vintramax | 对操作数所有的component求max，写出到rZ，并在对应的vpr上置位 |
| vmed | 3vector，per component求中值 |
|  |  |
| Float | |
| vfp2int |  |
| vfp2usint |  |
| int2vfp |  |
| usint2vfp |  |
| vfpcombin | vA , vB, vprC => vFloatZ |
| vfpextract | vFloatZ=> vA , vB, vprC |
| vfpcmp | {lt,le,gt,ge,eq,neq} vA, vB => vprZ （& vprC） |
| vfpadd |  |
| vfpsub |  |
| vfpmpy |  |
| vfpinv |  |
| vfpsqrt |  |
| vfpsqrti |  |
|  |  |
| Sliding Window | |
| vswmad | {vB, vA}组成的sliding window source，vC是系数  rD[5:0] shift  rD[13:8] pattern的起始偏移（6）  rD[20:16] 系数的的起始偏移（5）  m0={vB, vA}[ rD[13:8]]  m1={vB, vA}[ rD[13:8]+1]  coeff0 = vC[rD[20:16]]  coeff1 = vC[rD[20:16]+1]  init = shift ? 1<<(shift-1) : 0 其实就是round点  Z = m0 \* coeff0 + m1 \* coeff1 + init |
| vswmac3 | W = W + m0 \* coeff0 + m1 \* coeff1  Z = (e + m0 \* coeff0 + m1 \* coeff1) >> shift {psl}决定使用逻辑右移 |
| vswmpy5 | m0={vB, vA}[ rD[13:8]]  m1={vB, vA}[ rD[13:8]+1]  m2={vB, vA}[ rD[13:8]+2]  m3={vB, vA}[ rD[13:8]+3]  coeff0 = vC[rD[20:16]]  coeff1 = vC[rD[20:16]+1]  coeff2 = vC[rD[20:16]+2]  coeff3 = vC[rD[20:16]+3]  init = shift ? 1<<(shift-1) : 0 其实就是round点  Z = m0 \* coeff0 + m1 \* coeff1 + m2 \* coeff2 + m3 \* coeff3 +init |
| vswmac5 | W = W + m0 \* coeff0 + m1 \* coeff1 + m2 \* coeff2 + m3 \* coeff3 {accumulate}模式  Z = (e + m0 \* coeff0 + m1 \* coeff1 + m2 \* coeff2 + m3 \* coeff3) >> shift {psl}模式  Z = m0 \* coeff0 + m1 \* coeff1 + m2 \* coeff2 + m3 \* coeff3 + init {init}模式 |
| vswsad | Sum of Absolute Difference  src0={vB, vA}[ rD[13:8]]  src1={vB, vA}[ rD[13:8]+1]  coeff0 = vC[rD[20:16]]  coeff1 = vC[rD[20:16]+1]  W = W + abs(src0-coeff0) + abs(src1-coeff1)  Z = ( E + abs(src0-coeff0) + abs(src1-coeff1) ) >> shift {psl}决定使用逻辑右移 |
| vswsub | rD[12:8] Src offset  rD[31:24] Saturate Value  src0={vB, vA}[ rD[12:8]]  src1={vB, vA}[ rD[12:8]+1]  src2={vB, vA}[ rD[12:8]+2]  src3={vB, vA}[ rD[12:8]+3]  threshold = 1<<rD[31:24] – 1  sat() 将结果clamp在+/-threshold范围中 {sat}  Z0 = sat(src0-vC) 8b, 16element  Z0[+16] = sat(src1-vC)  Z1 = sat(src2-vC)  Z1[+16] = sat(src3-vC)  Z0 = sat(src0-vC) 16b, 16element  Z1 = sat(src1-vC) |
| vswsubcmp | rD[5:0] shift  rD[7:6] #abs\_diff\_operation  rD[13:8] Src offset  rD[31:24] Saturate Value  threashold = 1<<rD[31:24] – 1  cmpfunc = (abs(src – vCi) < threashold) \* src  cmpcnt = (abs(src – vCi) < threashold) \* 1  src0={vB, vA}[ rD[12:8]]  src1={vB, vA}[ rD[12:8]+1]  src2={vB, vA}[ rD[12:8]+2]  src3={vB, vA}[ rD[12:8]+3]  W0 += cmpfunc(src0) + cmpfunc(src1) + cmpfunc(src2) + cmpfunc(src3)  W1 += cmpcnt(src0) + cmpcnt(src1) + cmpcnt(src2) + cmpcnt(src3) |
| vmswmad | Vector Multi – Sliding Window – Multiply and Add  KLT  Source是B A在32b一组交织  Coeff分高低16个一组，在一组中间隔取2个。  Coeff0 = vC[rD[20:16]\*16 + 2\*i]  Coeff1 = vC[rD[20:16]\*16 + 2\*I + 1]  8b x 32  A[0] \* C[0] + A[1] \* C[1] => Z[0]  A[1] \* C[0] + A[2] \* C[1] => Z[1]  A[2] \* C[0] + A[3] \* C[1] => Z[2]  A[3] \* C[0] + A[4] \* C[1] => Z[3]  A[4] \* C[2] + A[5] \* C[3] => Z[4]  A[5] \* C[2] + A[6] \* C[3] => Z[5]  A[6] \* C[2] + A[7] \* C[3] => Z[6]  A[7] \* C[2] + A[8] \* C[3] => Z[7]  16b x 16  A[0] \* C[0] + A[1] \* C[1] => Z[0]  A[1] \* C[0] + A[2] \* C[1] => Z[1]  A[2] \* C[2] + A[3] \* C[3] => Z[2]  A[3] \* C[2] + A[4] \* C[3] => Z[3]  Z += init |
| vmswmac3 | W=W+m1\*k1+m2\*k2  Z=(E+m1\*k1+m2\*k2)>>shift |
| vsswmac5 | Vector step SlidingWindow Accumulate  D[7:6] filter number 用于coeff的跳跃  E Step\_size 用于src的跳跃 |
| vsspmac | Vector Sliding Pattern MAC  {vB, vA}组成的sliding window source，vC是系数  rD[5:0] shift  rD[7:6] filter number 1/2/4/8  rD[13:8] pattern的起始偏移（6）  rD[20:16] 系数的的起始偏移（5）  rD[31:24] pattern 4x2  E Step\_size |
|  |  |
| 兼容模式 | |
| vacc3kto4k |  |
| vacc4kto3k |  |
| vl2d |  |
| vs2d |  |
|  |  |

## PCU

|  |  |
| --- | --- |
| Opcode | Function |
| bkrep | Block repeat, lc+1次重复 |
| Bk[st|rest] | Store/Restore on block repeat nest level. |
| Break[O] | Break from block repeat. [O] means jump Out all level. |
| br | Branch绝对地址imm32 |
| brAR | AddressReg |
| brr | 相对PC地址，imm7/imm32 |
| call | Call绝对地址imm32 |
| callAR | AddressReg |
| callR | 相对PC地址，imm32 |
| ret | Return from call |
| retb | Return from breakpoint interrupt or emulation software interrupt |
| reti | Return from maskable interrupt or software interrupt |
| retn | Return from non-maskable interrupt. |
| [d|e]int | Disable/Enable Interrupt |
| modlci | Modify Lci register. |
| monitor{on/off} | 跟ld/st一起调用产生exclusive access |
| movp | Move imm32 to PCU register. |
| NOP |  |
| PSU | 改变状态到：freerun, lightsleep, standby, deepsleep, shutdown, dynamic, debugon debugoff |
| RSTP | Reset the appropriate field in modA or modB PCU mode registers. |
| SETP | Set the appropriate field in modA or modB PCU mode registers. |
| TRAP | 4 Software interrupt. Return with retregi. |
| TRAPE | Emulated Software interrupt. Return with retregb. |
| verifend | Set eotbit pin. |
|  |  |

Branch可以指定1~4 delay slot.

## LSU

不能busrt在4K边界上

Vector单元输出512b/256b

SPU单元

寻址模式

* Direct: imm32
* Indirect GRF, scalar, (rN.ui.t, rN.ui.t+imm32)
* Indexed Base(reg) + Offset(reg/imm16)
* Parallel absolute Base(reg) + vector\_offset(i8, s16, c32) 用户必须保证没有bank conflict！（如果有会怎么样？）
* Parallel Bank relative：{rel}, 地址是相对应的bank中的offset。
* Stack SP

Post Modification

* +
* -
* +s0~7
* +imm

Modulo Mode:

Cyclic buffer address, use modu0/1/2/3 寄存器

|  |  |
| --- | --- |
| Opcode | Function |
| ld/st | => 1/2/4 scalar register |
| push/pop | 可以对一组8reg同时执行，减少指令数。  Auxreg0~5/Pgreg/modv/modE/retreg/bknest/sp |
| modr | Modify pointer, rN or modu# |
|  |  |
| vaddgen | modvA , regB, Z. 产生8个地址 |
| vbcp[ld/st] | Bank coupled vector parallel load/store |
| v[ld/st] | 一个地址连续读入 |
| vp[ld/st] | Parallel, 16个地址 |
| vp[ld/st]2d |  |
| vmodr |  |
| vmodr2d |  |
| v[push/pop] |  |
| vldov | 装入4个vR，地址overlap，错位一个单元 |
| vldchk | 地址的奇和偶部分数据装入两个寄存器 |
|  |  |

Modv0~3 register

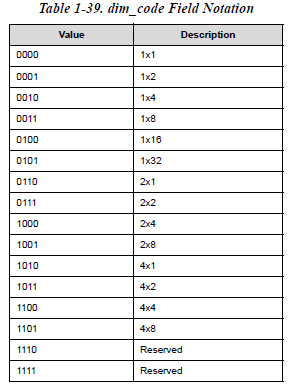
Stride: 32\*stb + 8 Byte, line pitch

Trns: transpose access

Chk: checkered

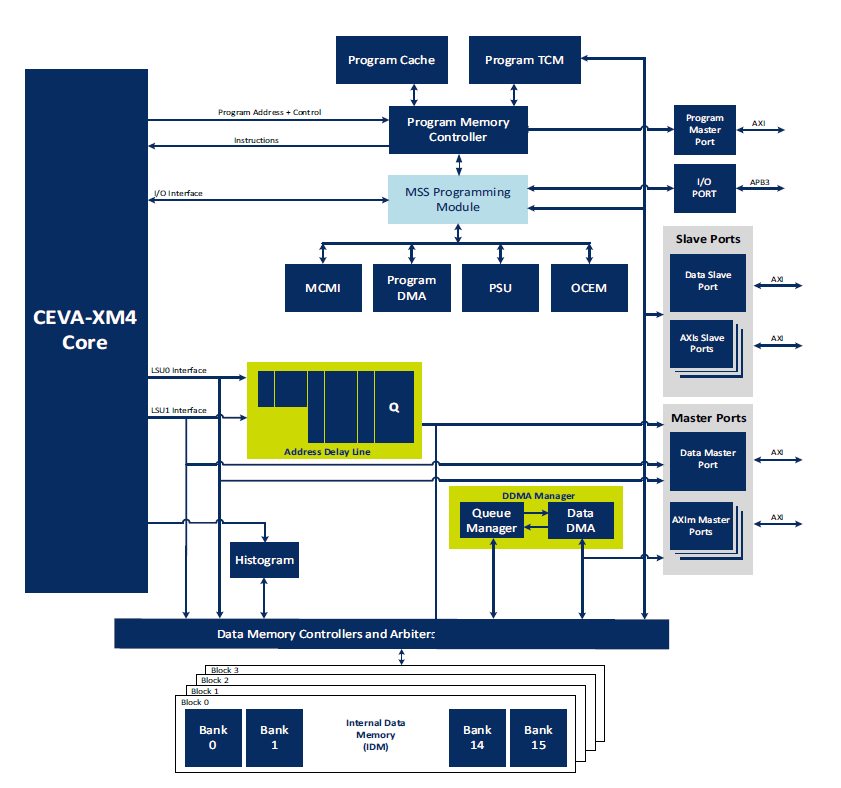
Fold:

Dim code: 1x32, 4x8 …

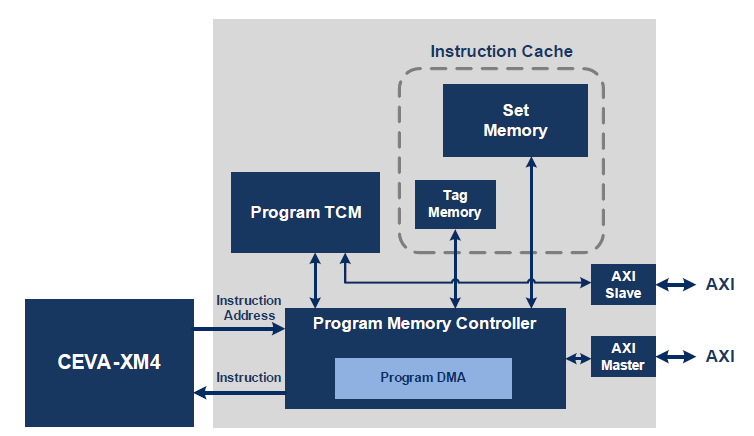


Modvl0~1

# 存储系统



## PMSS



程序地址0锁定在TCM，超出部分根据地址设定，如果是cacheable的会通过PCACHE访问。

可控的程序使用TCM或是软件lock住的cache。

IACU: Instruction Access Control Unit

SOU: Software Operation Unit

COF: Change Of program Flow

OCEM可以访问EPP R/W，cache sets/tags, TCM, Core Queue

Program as Packet: 变长VLIW包

PTCM由bar register决定，超出需要查tag

Cacheable: 到cache中取

Non-Cacheable: EPP出去

Arbitration 优先级（后两个可以写入程序，所以AXI为R/W）

* Core Read （可配置ID，0）
* PDMA （可配置ID，2）
* EDAP write
* OCEM （可配置ID，7）

L1IC:

Cache: 4Way Set Associative LRU, 512b cache line, 256bit external access (read only)

Hardware Pre-fetch: miss 取下一行

Software Cache Op: (PSWOU)通过写寄存器

* Lock/unlock
* Invalidate
* Pre-fetch

PDMA: 通过写寄存器， 128b带宽

Read Burst可以是1/3/7/f arlen

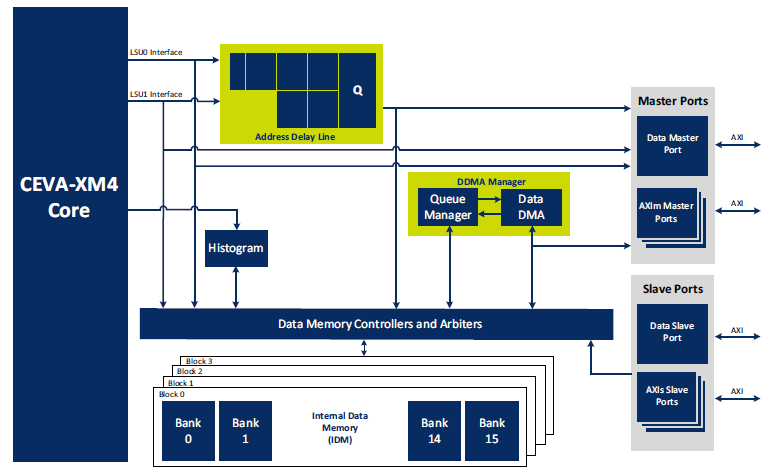
Write 单

EPP永远是16Bsize，

分16个区域，在4K边界上，可以定义对应的AXI bus的CACHE相关属性，可以关闭某个区域（如果外部存储不存在）

可以定义那个权限可以访问（全局而不是分区域）

## DMSS



LS0 256bR 256bW SPU/VPU Load/Store

LS1 256bR 64bW SPU Load/Store, VPU Load

Aligned

unaligned 减半

MOM（Memory Ordering Model）

* TSO: Total Store Order (read can bypass write buffer)
* SO: Strong Order (严格顺序)

DACU: Data Access Control Unit

RRA: Round Robin Arbitration (DDMA, EDAP, AXIs, Histogram 读写内部memory)

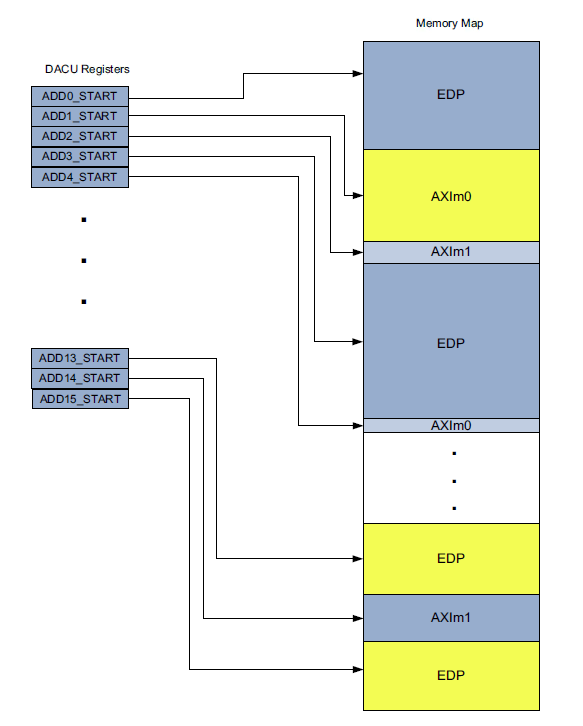
IOS: Internal Output Stage

EOS: External Output Stage

### DACU page 配置

类似于IACU，更多属性。

MID: Data支持多个master端口，指定是一个bit一个端口，可以同时往多个端口吗？有何意义

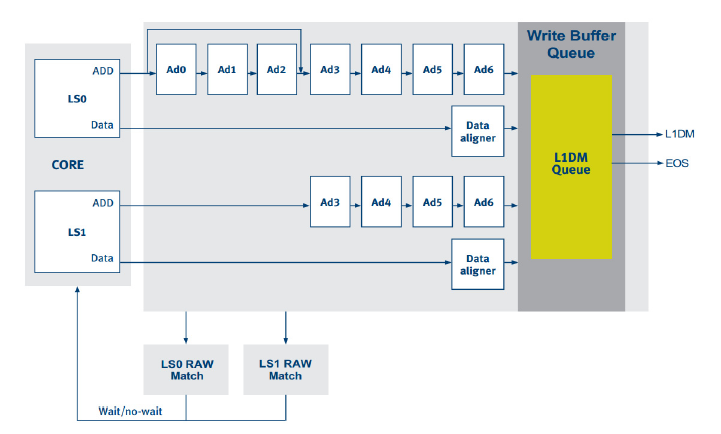


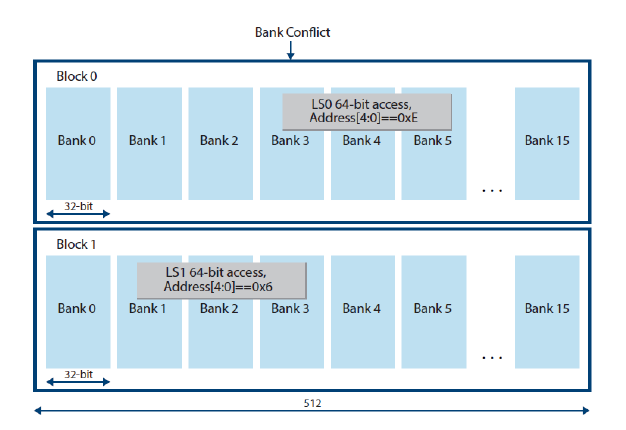
Core 不能访问被DTCM Bar覆盖的外部存储区域，但是DMA可以搬运

L2 Cache Control over MSS AXI port.

可以设计自己的一个cache，参考CortexA7

### Write Buffer





### Histogram

VPU0发出VHIST指令到TCM

16 histograms/cycle

### DMA

Task Descriptor

方式

* Message: 写入配置好的一个DW。DDEA->IDM, DDIA->DDEA
* Linear
* 2D
* One bank tile: only for IDM
* Two bank tile: only for IDM
* Duplicated One Bank: IDM as destination
* Duplicated Two Bank: IDM as destination

DDEA: Data DMA External Address

DDIA: Data DMA Internal Address

DDTC: Data DMA Transfer Control

Size (20)

IIT: IDM <> External or IDM<>IDM

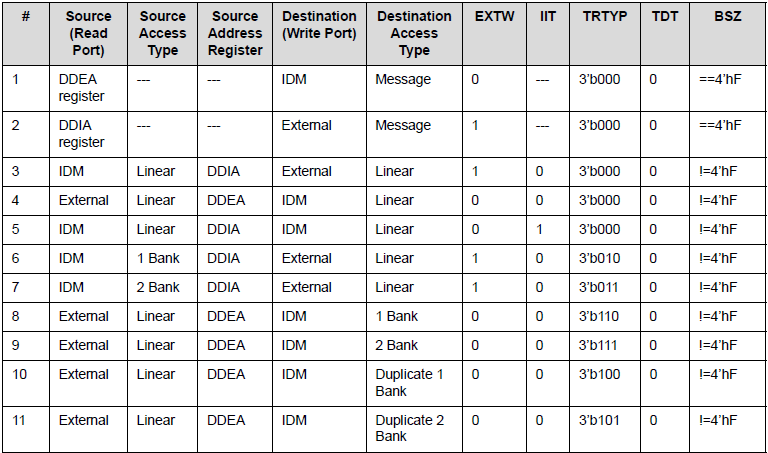
Burst Size: 1/4/8/16/32/64/128/256, FIXED/INCR, Message

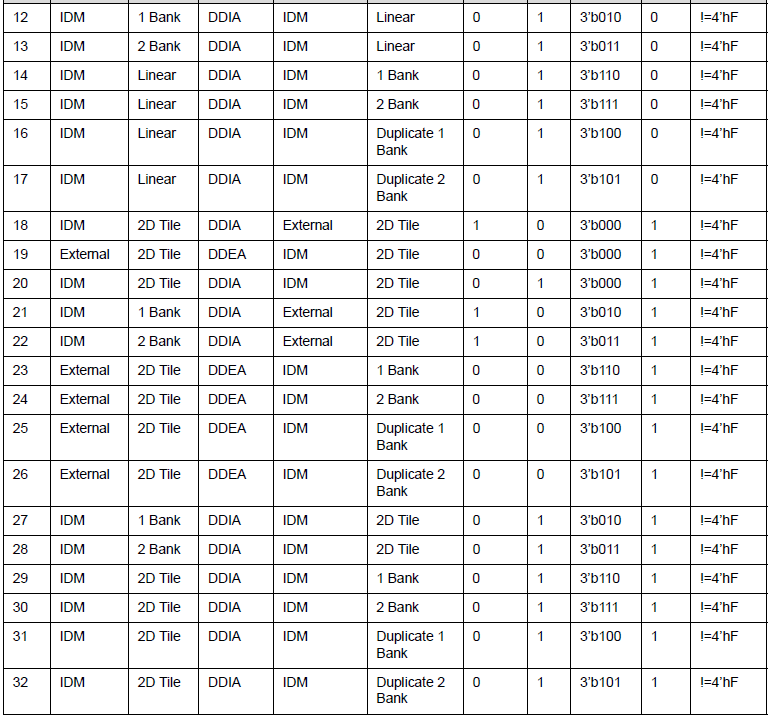
Interrupt enable DDIE

FIXED/INCR: 可以是重复写入burst length数据或是自动增加地址copy size的数据

Exact Read Mode: 不读入多余数据，产生小于data bus的读申请

Outstanding: 16 read, 4/8/16/32 write





Burst Length 最多可达256，slave是否支持，可否配置burst length？

### Queue Manager

支持8个Queue，可编程Queue的大小, 可以在IDM /External memory中。

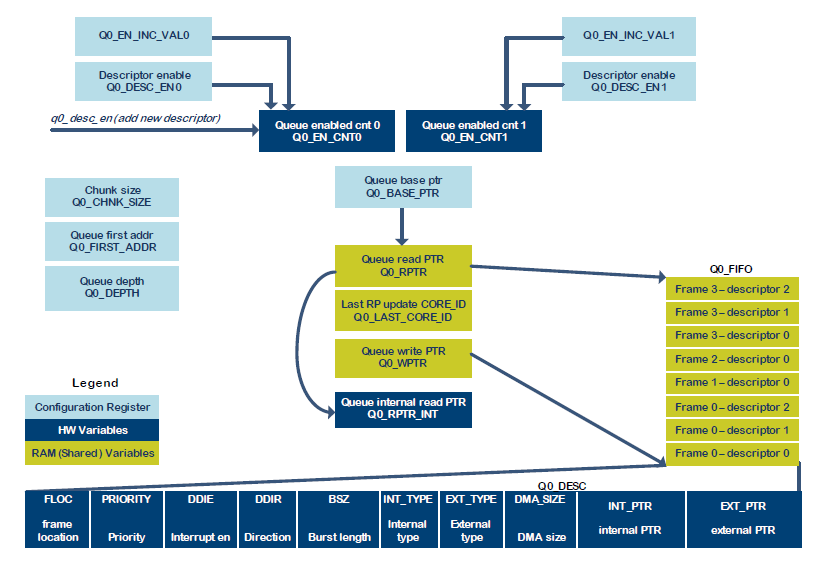
task => chunk => arbitration (按目标数据消耗排序，而不是按源端数据准备好排序)

多个task可以被合并为一个task frame

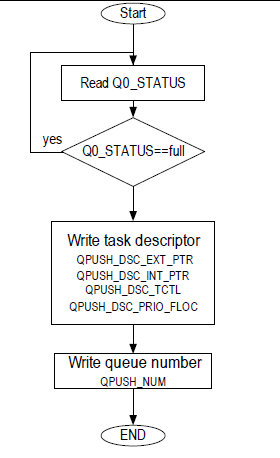
Task由descriptor描述，128b。2D的task占据256b. 其中每个task有priority（也可以用per queue的优先级设置），frame number（2b circular使用，可以区分回绕）, 和order within frame(8). 同一个frame中的task可以放在不同的queue中，执行要按顺序。

每个queue有两个内部counter，分别为 source和destination计数，都大于0表示可以传输。传输结束后减一。Counter的增加可以用对应queue的寄存器写入，一次最大可以增加64. 也可以通过外部端口增加。

|  |  |  |  |
| --- | --- | --- | --- |
| qn\_desc\_en | I | 16 | QMAN enable descriptor increment signal |



快速写入



Q Descriptor 5 DW: 快速写入queue的方法，queue和pointer都在IDM中。

* Qpush\_dsc\_ext\_ptr(32)
* Qpush\_dsc\_int\_ptr(32)
* Qpush\_dsc\_tctl(32)
* Qpush\_dsc\_tframe\_len(32)
* Qpush\_num(4)
* Qpush\_status\_rls
* Qpush\_status

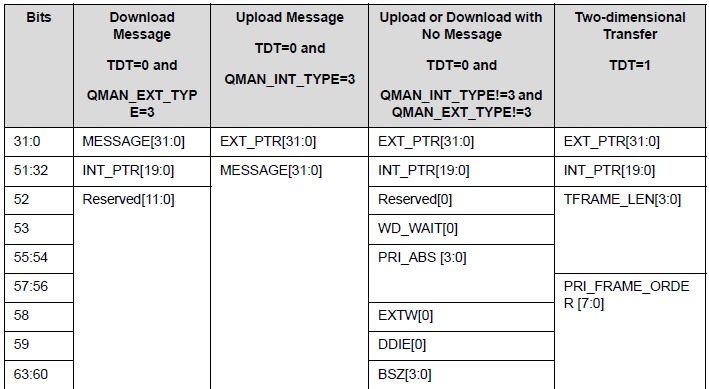
Q Manager Control 4DW

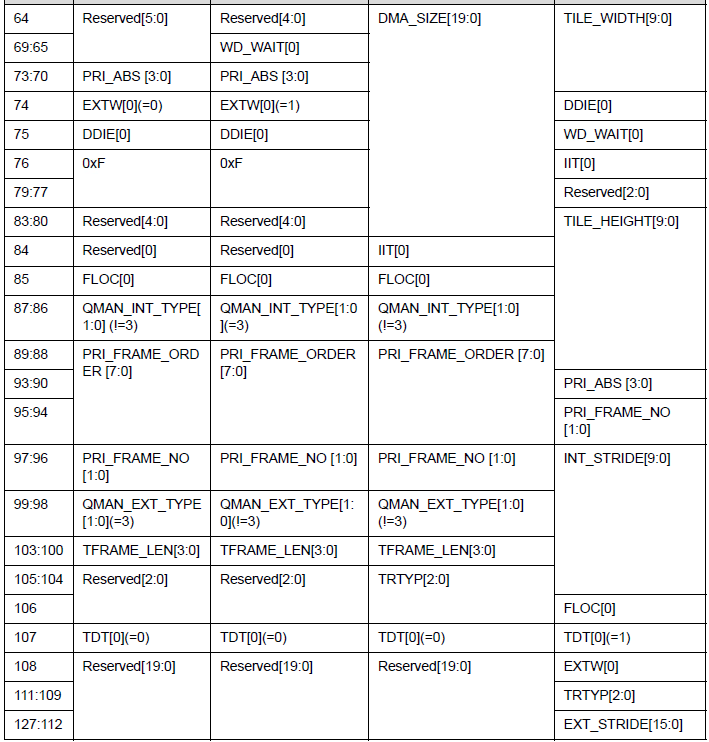
* Q[0-7]\_IDM\_CROS: error, cross IDM last address
* Q[0-7]\_Active
* empty\_vi\_mask(8): queue empty violation interrupt mask
* fnum\_vi\_mask: frame number violation interrupt mask
* empty\_vi(12): queue empty violation interrupt status
* fnum\_vi: frame number violation interrupt status
* Q[0-7]\_reset: software reset for each queue.

Q Manager 13DW \* 8 = 104DW, 416B, (500B reserved), 每个queue包含的信息有（Qx\_）

* depth: max 2^13+1 descriptor
* pri\_abs(4): 绝对优先级
* pri\_abs\_sel: 0 是用task descriptor中的优先级，1：使用上面的优先级
* cont\_frame:
* en: enable
* first\_addr(32): Queue start addr
* base\_ptr(32): 指向RPTR/WPTR结构。
* chnk\_size(14)
* chnk\_lines(10)
* en\_inc\_val[0|1](6) 增加量
* dsc\_en[0|1] 增加操作
* dsc\_cnt\_cfg[0|1] 什么情况下dec 内部counter。
* en\_cnt\_dis[0|1]
* en\_cnt(14)
* wd\_wait
* status
* rptr\_int(17)
* dsc\_ext\_ptr(32)
* dsc\_int\_ptr(32)
* dsc\_tctl(32)
* dsc\_tframe\_len(32)
* en\_cnt[0|1] (14)
* wd\_wait：等当前task完成后才送下一个task，strong order。 (in upload task desc)

Task Descriptor





FLOC: location of a frame, 1 means start, 0 in the middle

TFrameLen: 剩余的task数。Floc==0， TFrameLen=0表明是最后一个task  
TDT: Two Dimensional Transfer

IIT: Internal to Internal Transfer

EXT\_PTR: external address pointer

INT\_PTR: internal address pointer

Ext/Int Type: 0 DMA address, 1: means address is just constant message.

同一个frame的task order要按顺序执行，但是frame number 2b要求同时间8个queue头上看到的queue的frame number应该不超过两个且连续！

如果让多核共同执行一个queue，要求一个frame中的task都必须在一个core中执行。

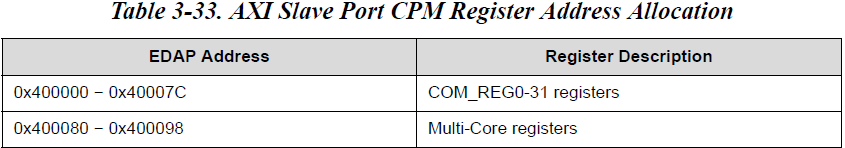
需要一个semaphore。（每个queue一个）

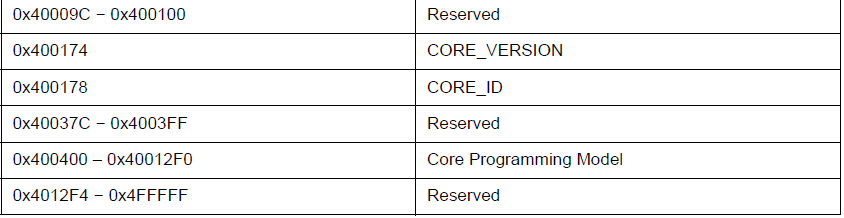
一个core获取semaphore后获取读指针，读头一个packet，获取共有多少task在这个frame中。再写回更新后的指针，释放semaphore。然后可以处理获取的一个frame的tasks了。处理过程使用internal指针（RPTR\_INT）

DMA operation:

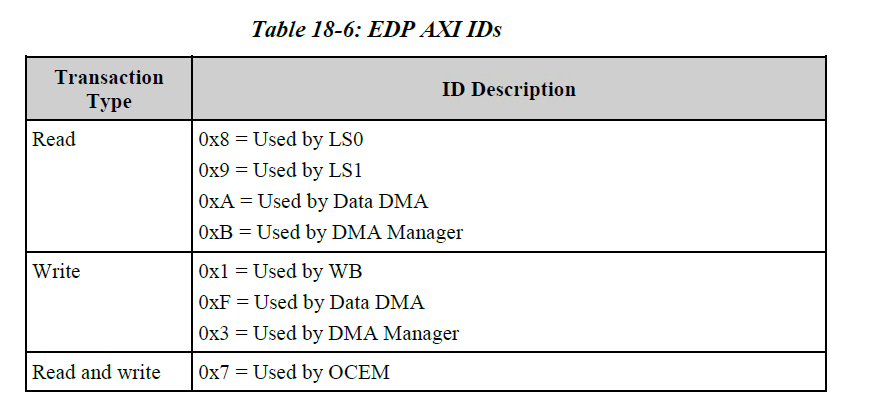
* DDMA move
  + DDMA Upload: IDM to External
  + DDMA Download: External to IDM
* Message to de-alloc packet data memory
* Message to destination indicate ready

### AXI Slave





### AXI ID



# Programming Model

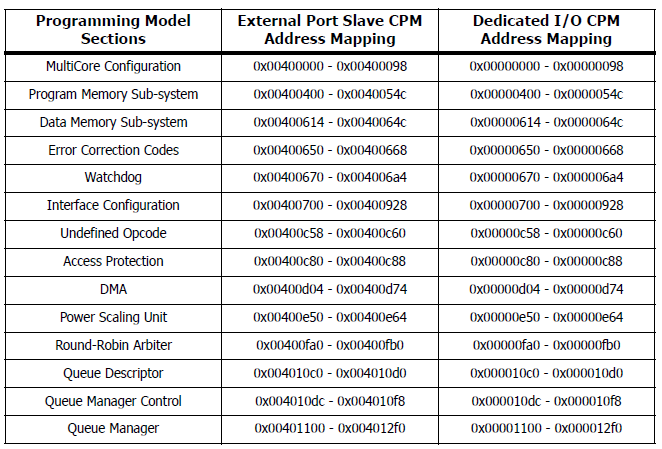
EDAP/Slave 访问，23bit 地址空间 8MB空间

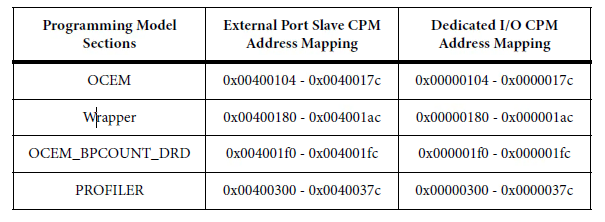
* [22:21] == 00 访问IDM（Internal Data Memory）
* [22:21] == 01 访问TCM（Program）
* [22:21] == 10 访问CPM（Core Programming Model）寄存器

IN/OUT指令访问CPM

* in page, cpm, dst, imm
* Out:

设置上CPM（Core Programming Model）开关就访问内部寄存器，否则就出APB Host端口

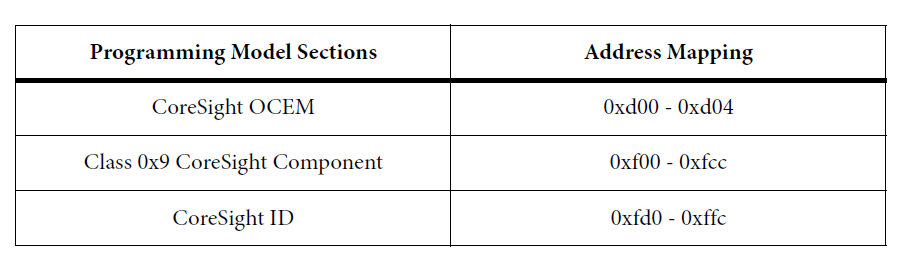


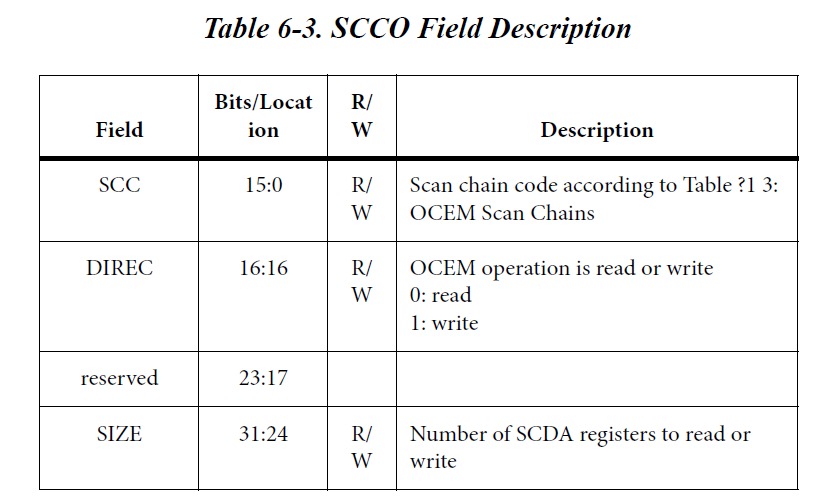


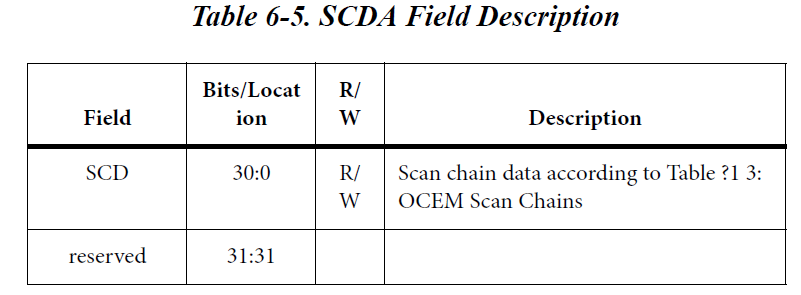
Total register space: 0x1000+0x400 = 4K + 1K = 5K

# OCEM

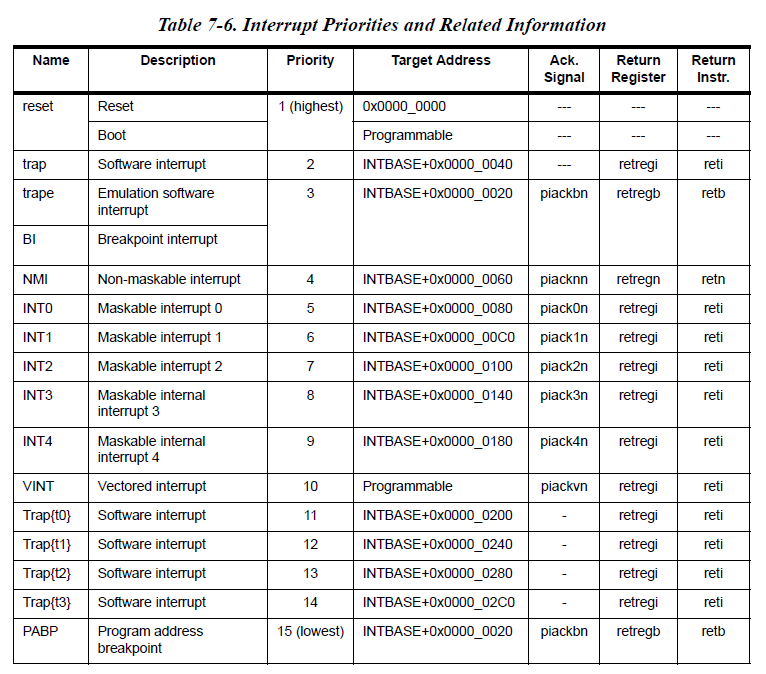
Coresight 4K Page







# 中断



外部只有0/1/2中断线，3/4被保留为内部设备（3- PDMA，4- DDMA）

|  |  |  |
| --- | --- | --- |
|  | SW | HW |
| Maskable | Trap 0~3 | Int0~4, vint |
| Non-Maskable | Trape | Nmi  BI/PABP |

User1 不可IO

User0 不可开关中断，PSU

Super 超级用户

# Multi-Core

## AXI exclusive access

需要Slave 支持！！SRAM单元的功能？

* core发送exclusive access
* 对应的slave必须回应EXOK，否则导致GVI

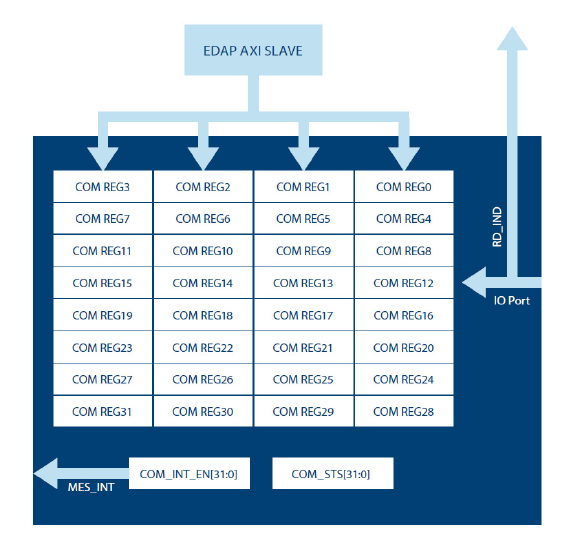
Core 要发出exclusive的访问需要

Monitor {on} + LD

Monitor {off} + ST

如果MC支持exclusive的访问

## MCCI: Multi-Core Communication Interface



只能通过外部写入32个命令寄存器，任何一个写入记录在COM\_STS位上，如果enable就产生中断MES\_INT。用户自己将其接到外部中断上。

Core通过IO指令读取命令寄存器，读取的同时产生一个pulse在32比特中的一根线上，返回到initiator上

Core通过写入COM\_STS比特清除该位。

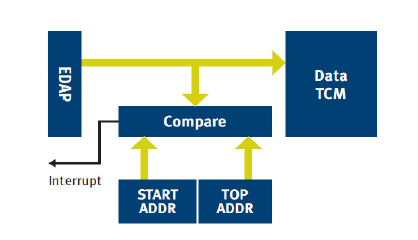
通常在多核的使用中，用户需要设计一个interrupt matrix. 每一个core输出的32b的pulse可以用于产生interrupt到指定的core上作为message的ack。

一般的设计中使用指定的分配，比如cmd0~3给core0，读0~3的寄存器会返回core0一个中断（core 0 还需要读出寄存器才知道这是message被处理好的中断）

复杂的情况下可以设计一个可配置的网络。

这样的方式是两次中断和查询过程。如果简化设计，CoreA通过写cmd寄存器通知CoreB任务，CoreB也可以通过写A的cmd寄存器返回完成信息更简单。

## Snoop



检查从EDAP或是其他三个SLAVEport来的访问是否落在指定监视的区间。

Some SOC use this feature to directly transfer ISP data to DTCM, and start/end address trigger interrupt for double buffer solution.

# Pipeline

IF1/2 Instruction Fetch

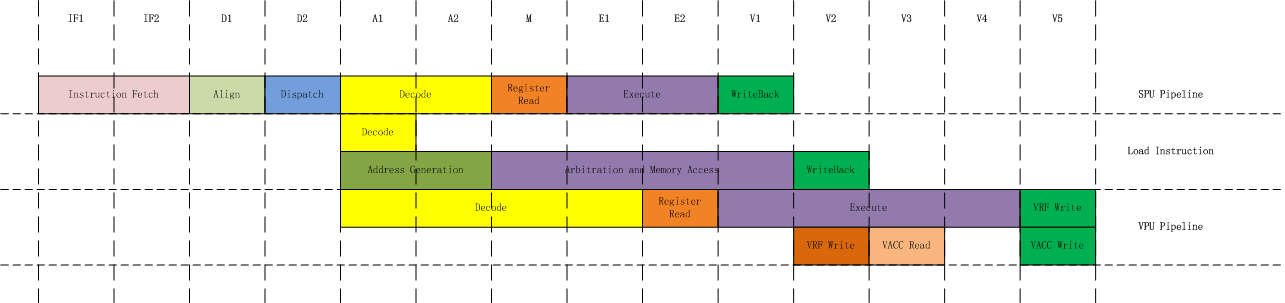
D1/2 Decode/Dispatch

A1/2 Address Generation

M Memory Access

E1/2 Scalar Execution

V1~5 Vector Execution



psu/psu\_core\_tree\_

# 扩展XTEND

分别对SPU0/VPU0 （SPU1~3/VPU1不参与扩展）

都提供两个操作数32/256b

和predicate 1/32b

指令编码23b

扩展数据26b

## 同步模式

紧密结合在pipeline上，参考10

SPU0的V1阶段取结果

VPU0的V5阶段取结果

指令需要编译器支持，显然只能汇编级编码

VXH.<user\_op\_name> vA.[u]c32, vB.[u]c32, vZ.c32

需要查看SmartNcode, Assembler and Linker

## 异步模式（Trigger）

执行时间超出pipeline允许，写入扩展自己的寄存器，如果确切知道多少时间结束可以在执行一些其他指令后move回GRP

不确定时间可以用

* 中断
* Poll（需要占用一个GPIO）

确定结束后move回GPR

# Design Inside

## fp\_top

scalar/vector共享的单元。cevaxm4\_fp\_top

Input:

* src0/1 32b
* src2 1b
* combine\_src0 32b
* combine\_src1 8b

output:

* res0/1\_stg3\_r 32b
* mod/flags 11

submodule

* add\_sub: alighn + 28bit add + norm\_shift + round(inc), 3 stage
* mpy: 24x24 (2stage), lshift, round, 3 stage
* cmp
* extract\_combine
* fp2int
* int2fp

Scalar可以配置一个给第一个SPU，或者是4个（给每个SPU）

VPU一次配8个

## Scalar

dsc\_div

dsc\_asu add sub unit

dsc\_ffs finding first set

dsc\_flags

dsc\_logic and/or/xor/andnot/not/nand/nor/xor

dsc\_mov

dsc\_mpy\_mac dw\_mult\_2\_stages (33, 33), (17, 17)

dsc\_pou Predicate Operation Unit

dsc\_shifts\_top

dsc\_slb

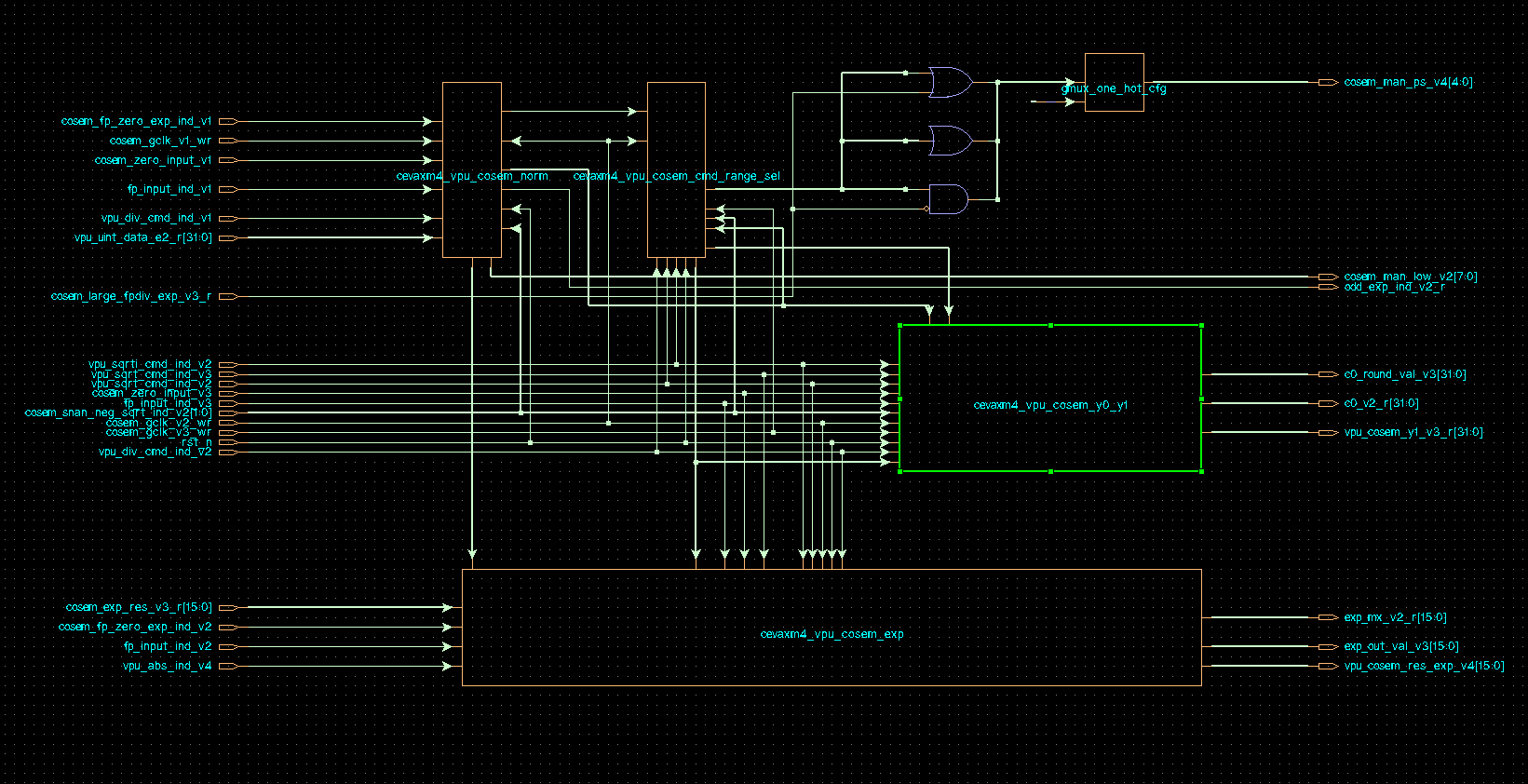
dsc\_source provide source data

只有SPU0含有32x32和除法器

## Non-Linear Processing Unit

vpu\_div\_sqrt\_sqrti

* exp v1-4
* norm v1
* cmd\_range\_sel v2
* y0\_y1 v2-3



## VPU

vpu：

* 16bit\_add
* 32bit\_add
* adder\_v3 24b adder
* alu
* alu\_2
* mul\_calc 32 16x8, 32 16x16

floating point en:

* 8 fp\_top instances

Include cosem

* 16 div\_sqrt\_sqrti instances

vpu\_vfr 寄存器

# Simulation Environment

design/top/

<design>\_param\_file.v

cevaxm4\_gen\_param.v

cevaxm4\_pmem\_param.v

cevaxm4\_dmem\_param.v

simulation/asm/verilog/top

top\_rtl\_def.v 由installer产生，含有大量仿真用的宏

work/

envdef.v: 定义一下仿真参数，包括Internal/External Data Init Memory路径

param\_file.v 空

siggen\_file.v

siggen\_task\_file.v

test\_file.v 包含test

**Macros**

INSERTION\_DELAY: ceva\_clk => ceva\_clk\_balanced

SDF:　不使用an\_error\_bus

SVA

FV\_AN

DBG\_VERIFIER: DBG\_verifier

RUN\_BFM: ocem\_host

TOGGLE\_EXTERNAL\_WAIT

INIT\_DATA\_FROM\_FILES

CEVAXM4\_NO\_AXI4: 支持到AXI3

CEVAXM4\_AXIM#\_AXI\_WIDTH\_ENC

CEVAXM4\_AXISLV#\_AXI\_WIDTH\_ENC

CEVAXM4\_ECC

CEVAXM4\_PMSS\_SPLIT\_B0

大量通过generate if 方式连接

## Main Glu Logic

error捕捉类型：

* Power isolation
* Aph/axi thpt
* Memdb
* Slv0/1/2\_axi
* Epp/edp axi 4k page error
* Sv assertion error
* Pmc general/b0/b1/iarb/dma/miss\_hit fv error
* Trace monito
* Verifeq

Gp\_in:

7:0 loopback gpout[31:24], rd\_max\_outstantding,

15:8 loopback gpout[23:16], wr\_max\_outstanding

23:16 loopback gpout[15:8], gp\_ui\_in

31:24 loopback gpout[31:24], 0

DMSS gpio state: decode => edp AXI0/1 ready

如果环境不驱动EDP/EDAP，将两者连上

## clkgen:

## sim\_cntrl

引出CEVA寄存器

DMSS GPIO State:

DGS\_

* IDLE
* ACTIVE
* EDP\_AWREADY
* EDP\_RRDY
* EDP\_ARREADY
* EDP\_BREAD
* AXI0\_BREADY
* AXI1\_BREADY

DGC\_#

* STOP
* ACCESS
* AWREADY
* RRDY
* ARREADY
* BREAD
* AXI0\_BREADY
* AXI1\_BREADY

start\_pulse

test\_active：跟随start\_pulse起来

结束采集error线

PC dependent?

pc\_mem.v

可以在某PC stop

**主循环**

init\_top

test\_start\_init

Boot from PC: boot=1 vector=`BM\_PC

core\_id = 01020304

enable\_checkers

一个case，包含n test，超出后调用

finalize\_report

end\_simulation

test\_loop: 用test\_num来计数

fork1

load\_mem\_using\_edap(T)

fork2

提供boot/vector (包含siggen\_file.v，index test\_num)

wait\_for\_timeout设置timeout

fork3

generate\_start\_pulse

init\_data\_memory -> initi\_internal\_data\_memory

init\_program\_memories

init prog mem blk 0x3cc0

init pcache

[init\_ecc\_memories]

2cycle

Load\_data\_mem: 从mem/<test>/\_int\_data\_blk#\_bnk#.mem装入数据

Load\_program\_mem

Load\_int\_prog\_mem\_blk# : 从mem/<test>/\_int\_prg\_blk#\_bnk#.mem装入数据

Load\_parity\_mem\_blk#:从mem/<test>/\_int\_parity\_blk#\_bnk#.mem装入数据

Load\_ext\_prog\_mem: 从mem/<test>/\_ext\_prg\_blk#\_bnk#.mem装入到pmss\_host模型中

触发事件release\_resets\_event

8cycle后release\_resets

…DSP正常运转

等待stop\_signal, test完成

dump\_internal\_data\_memory：写出到initmem/<config>/end\_data\_blk#\_bnk#

print\_final\_values:打印寄存器的最后值到文件final\_reg\_values.log

activate\_resets：开启reset

disable test\_loop：终止其他的fork

test\_num ++

dump memory if needed

task区域

ECC Error产生区域

Assertion区域

Siggen\_rf : Direct Debug

### sim\_rptu: report Unit

打印和log文件最后的summary

### sim\_erru: error report unit

* 去rptu
  + test\_message
  + analyzer\_fail
  + comparison\_fail
  + asm\_fail
  + force\_fail
  + no\_errors 仿真结束后正确就拉起
* Ana

产生test\_message来源是

* ASM编译错，comparison\_fail, 编译时产生的结果，如果不看就跑仿真，是不会执行下去的。
* 结果比较错误，core提供
* 37bit出错线，对应37条message，配置到各端口模型上和分析模块上。没有全部用完
* evc\_error ?
* time\_out

## AXI

### axi\_if\_master

input APB

连到EDAP/AXIS

由寄存器写入控制transaction

内部用SV的queue实现

实现的task

* 0:normal write:
* 1:normal read
* 2: rw\_sequence: fork 0/1, 同时读写
* 3: write data from last read：
* 4: transaction after light sleep：wait idle起来后读或写
* 5: transaction after sleep without stand by：wait idle起来且gpin0（？）后读写
* 6: transaction during debug to ptcm：等ocm\_debug\_r
* 7: read transaction after gpout0\_high：wait gpout[0]
* 8: wait for pdst：等待一次PDMA完成操作
* 9: write after gpout0 high：wait gpout[0]
* 10: write after gpin0：wait gpin0

Init\_edap:

读入数据from edap\_init\_addr.mem

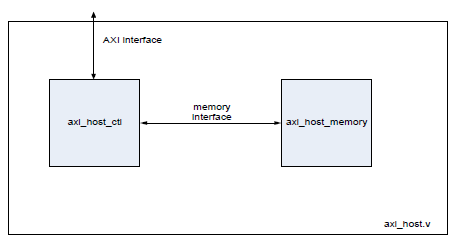
或者用DMA初始化

### axi\_host

用在

dmss\_edp  
dmss\_axim0/1

pmss\_epp



#### axi\_host\_ctl

地址

0xFFFF\_F12x SLVERR

0xFFFF\_F56x DECERR

0xFFFF\_F9Ax EXOKAY

支持吗？

OKAY\_SHARED Shared Clean

OKAY\_UD Unique Dirty

#### axi\_host\_memory

SRAM

### cal\_axi\_slave\_parity

### cal\_axi\_master\_parity

## APB3

IOP to external APB devices, or OCEM APB interface。

### apb3\_adec

输入：

* Core的IOP

输出

* Apb3\_host
* Apb3\_master
* edap\_master
* axis#\_master (0-3)
* TPIU

地址译码

[31:16]==7000 APB3

[31:8]==0x00 EDAP

0x0000\_06xx ETM

0x0000\_08xx TPIU

[31:8]==0xf0 [7:0]==0x68~90 AXIS 0

[31:8]==0xf1 [7:0]==0x68~90 AXIS 1

[31:8]==0xf2 [7:0]==0x68~90 AXIS 2

0xffff\_ffff ERROR

### apb3\_host

8个寄存器，使用gated clock write。

Write enable来源于模块gapb\_if\_cfg，它实现了N（=8）个连续寄存器的读写

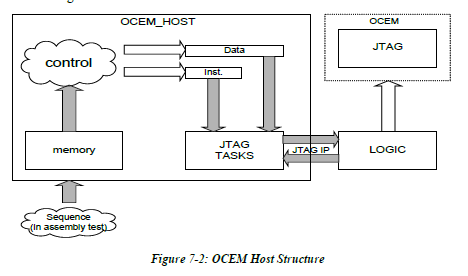
### apb3\_master

for OCEM

使用queue，转发APB

### cal\_apb3\_slave\_parity

## ocem\_host



## Extension

### ehwvu\_host

for VPU XTEND

### ehwu\_host

for SPU XTEND

## 分析用

axi\_if\_an

axi\_thoughput\_an

apb3\_an

apb\_througput\_an

power\_isolation\_an

freerun\_an

logger

sim\_counter

## ETM支持

Instance from ARM

* CSTPIULIGHT
* ETMR4

sim\_top\_en\_etm (外部模式)

* sim\_top\_ext\_etm\_r4
  + ETMR4
* tpiu\_monitor
* CSTPIULITE

# CEVA SDT

## 安装

必须在虚拟机上使用才能check out license. 共有两个license，仅在编译才会用到。

安装文件在/LnxShare/eda/Ceva/cevaXM4\_V15.1.1.\_ToolBox.exe

安装密码1842-5040-6341-2040

解压CV library from /LnxShare/jszheng/CEVA-XM4\_CEVA-CV\_V1.0.0.zip

**Windows**

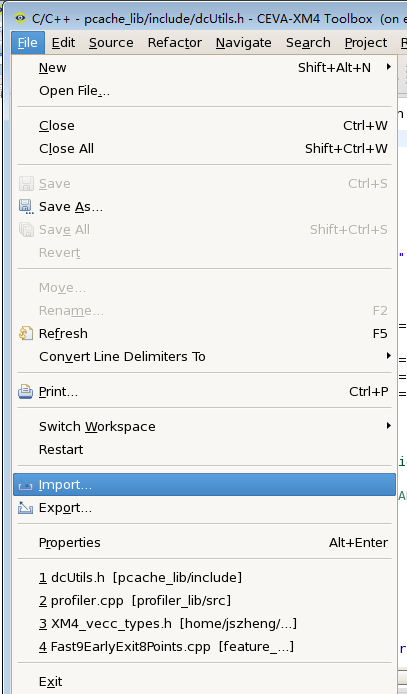
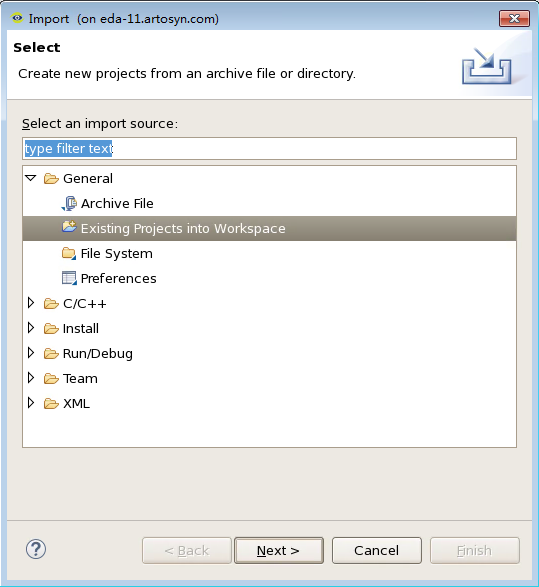
设置环境变量

LM\_LICENSE\_FILE = [27003@192.168.200.241](mailto:27003@192.168.200.241)

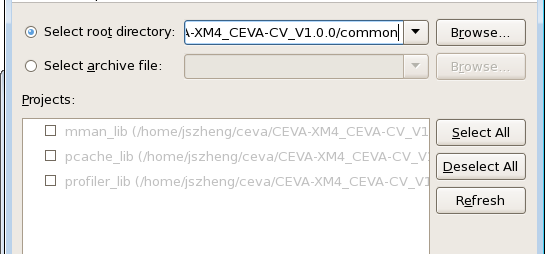
启动IDE

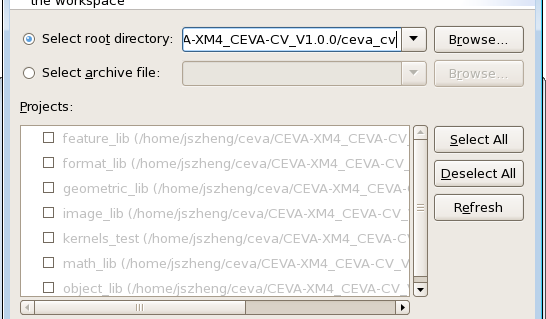


import cv library

Select all project from ‘common’ and ‘ceva\_cv’ directory





Choose kernels\_test project to build. (will take some time cause it will build all other project that depends on)

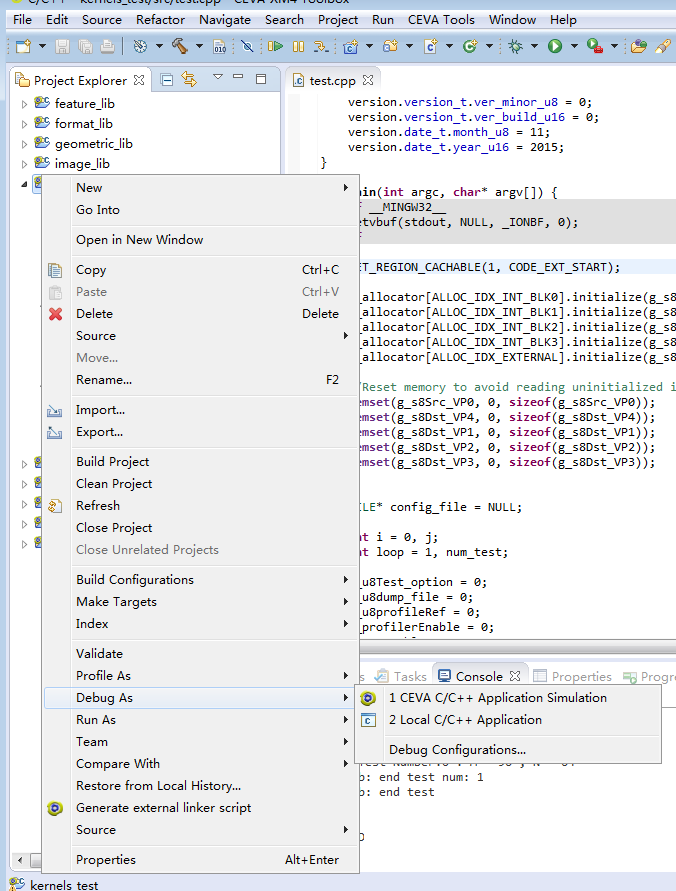
**Linux**

% module load ceva

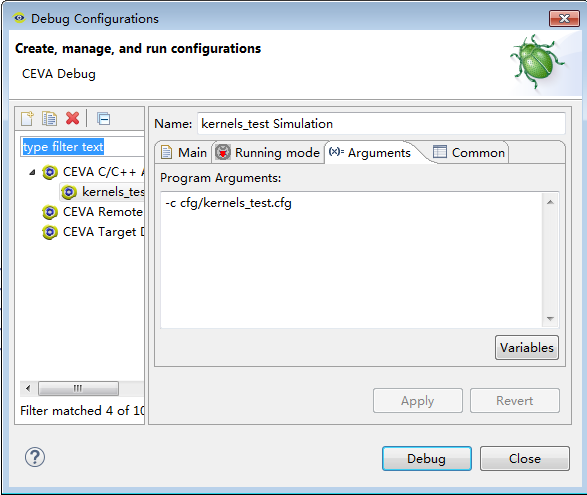
% toolbox

Build will fail cause all include path to ‘common’ using windows path separator(‘\’) which cause build error. Fix all projects setting and re-run the build.

**Run the test**

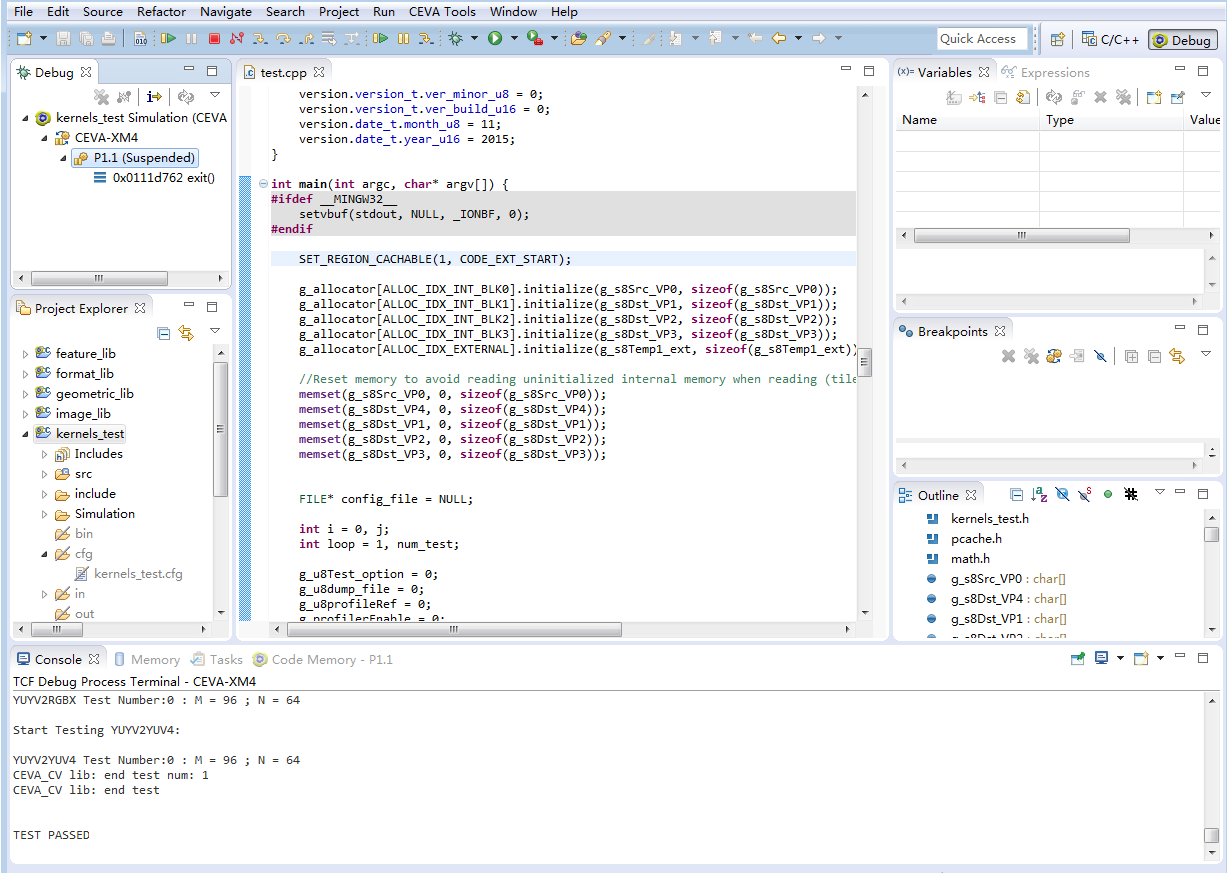


Choose Debug configurations…, add the argument as follows



The click Debug button to start.

All test should passed



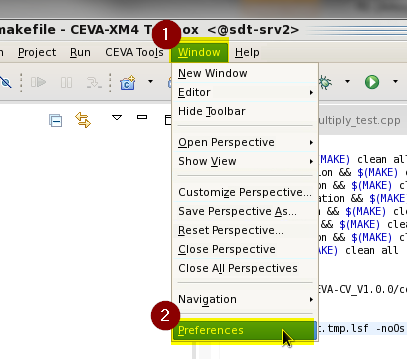
Linux Compile Issues

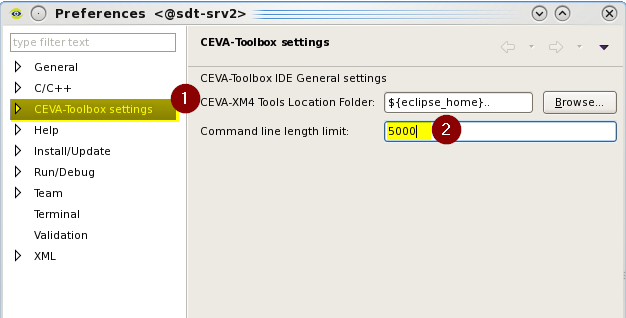
* The include path using character '\' which is not support under Linux. 全部改为‘/’
* <CVLIB>/ceva\_cv/tests/test.lsf需要用dos2unix转码，否则mpp会报错。注意mpp并不输出详细的解释，但是由于提前退出，会在工作目录下留下mpp####的临时文件，打开文件就知道哪个lsf文件导致的错误。

lsf file: Linker Script File

如果你的CVlib放置的目录太深，有可能遇到报错，按如下修改。

However, when linking the project, we were required to tinker with the Eclipse option of dumping command line options to files:

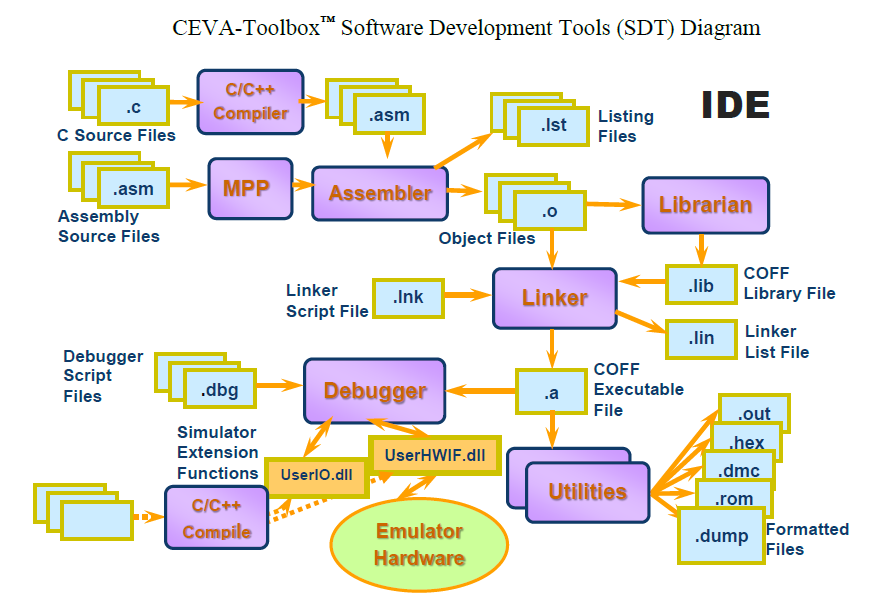




By default, the project wasn’t built with the latest XM4 V15.1.1 delivery, but when changed the command line length limit to 0 it linked successfully.

将limit改为0好像有问题，应该是个够大的数。

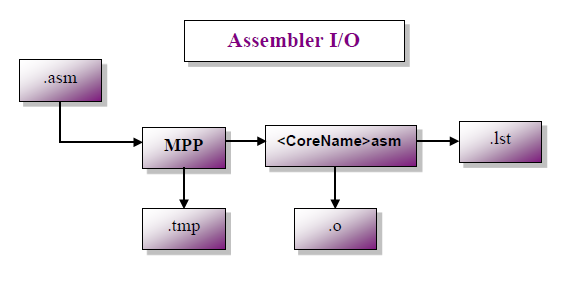
## 使用



xm4cc

xm4cpp

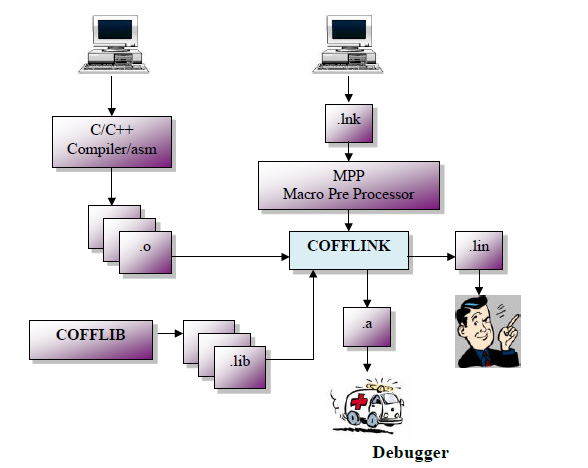
xm4cc.cfg



mpp

disasm <coff\_file>

-enc 0x######## CEVA-XM4



coff link

cofflib

coffutil

coffdump

coff2hex

hex2rom

hex2dmc

Cpp 编译好的文件有

.i 预处理过的c

.ii 预处理过的cpp

.s 无需mpp的汇编

.asm 需mpp的汇编

.lst 反汇编（带binary）

## 程序的运行

DebugAPI

* Simulation
  + ISS: Instruction Set Simulation. 外部的存储访问是通过observer实现，输入是SignalAPI
  + CAS: Cycle Accurate Simulation 有外部bus的仿真，开ESL
* ESL: ISS and CAS, expose core and MSS signals
* Emulation: JBox

DCli: 一个实现

% dcli –core CEVA-XM4

>load coff xxx.a

>go

>start tcl

## 汇编语言

.EQU

.IF

.ELIF

.ELSE

.ENDIF

.IFDEF

.IFNDEF

.PURGE (undef)

.MACRO

.ENDM

.DB

.DW

.DD

.EXTERN

.FF Form Feed

Label:

%l: temp label

#imm short immediate

##imm long immediate

$label address of label

Sections

.CSECT .CODE

.DSECT .DATA

text

data

bss

# TODO

## On Chip Emulation and ETM

ETM R4是ARM的IP，可以用ETM-M7 或是ETM-A7替代吗？

## ECC

内部RAM EDC

AMBA EDC：Core 不看是否有ECC，照发transaction.

# 名词

|  |  |
| --- | --- |
| MSS | Memory Sub System |
| SPU | Scalar Processing Unit [0-3]  sflp: in 1FPU or 4FPU |
| VPU | Vector Processing Unit [0-1]  vflp: 0 or 16 (8 per VPU)  nonlinear\_units: 0 or 32 (16 per VPU) |
| dmss | Data Memory Sub System |
| pmss | Program Memory Sub System |
| dman | DMA Manager  qman\_num: 0 or 8 |
| LSU | Load Store Unit [0-1] |
| PCU | Program Control Unit |
| PSU | Power Scaling Unit |
| pmem | Program Memory  PTCM 0/32/64/128/256 KB  Cache 32/64/128 KB |
| dmem | Data Memory  DTCM 128/256/512 KB |
| ocem | On-Chip Emulation Modules |
| RTT | Real Time Tracer  Wrapper: RTT wrapper  [ETM-R4]: ARM License  [tpiu\_lite]: |
|  |  |
| EPP | External Program Port |
| EDP | External Data Port |
| AXIM | SOC AXI Master Port  axi\_master\_number : 1 or 3  axim\_width: 128 or 256 |
| AXIS | SOC AXI Slave Port  axi\_slave\_number : 0 or 1 or 3  axi\_slave\_width: 0 or 128 or 256 |
| TCE | Tightly Coupled Extension |
| Xtend | Extension Interface  spu\_xtend  vpu\_xtend |
| SIL | Safety Integrity Level  ISO-26262 SIL-B |
|  |  |
| GRF | General Register File r0-r31 (32) |
| VRF | Vector Register File 40x256b register  v0-v31 as general purpose  v24-v39 as accumulator  format  i0-i7 32b integer  f0-f7 32b float  s0-s15 16b short  c0-c31 8b character |
| PRF | Predicate Register File  pr0-15 16x 1b  vpr0-6 7x32b |
| ARF | Address Register File |
| SRF | System Register File |
|  |  |
| IACU | Instruction Access Unit |
| DACU | Data Access Unit |
| GVI | General Violation Indication |
| MoM | Memory Ordering Models |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |