

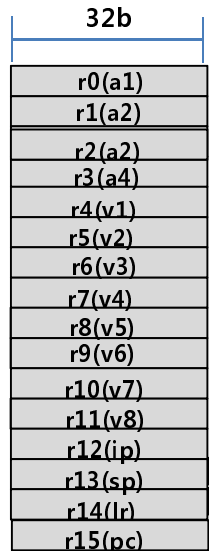
Transitioning from ARM v7 to ARM v8: All the basics you must know.



Mario Smarduch
Senior Virtualization Architect
Open Source Group
Samsung Research America (Silicon Valley)
m.smarduch@samsung.com

Registers-ARMv7 (no VFP/SIMD)

ARMv7 – ARM instructions



a1-a4 – args/return
v1-v5,v7,v8 – variable regs
v6 – platform reg.
 - subroutine must preserve v1-v8
ip – inter procedure call scratch reg
 - linker scratch reg. practically used as regular register
sp – stack pointer
lr – link reg, bl saves return addr
pc – program counter +8

Procedure call

```

a( ... )
  push {r6,r7,r8,lr}
  sub sp, #48
  add r7, sp, #8 ; frame-pointer
  ; mov or push stack arguments
  mov r3, arg
  str r3, [r7, #4]
  ....
  bl a
  ; adjust sp
  mov sp, r7
  pop {r6, r7, r8, pc}

b(...)
  ....
  
```

- 'pc' – accessible register, lr popped into it
- pop/push is alias of 'ldm/stm'
- Register set – 12 usable regs – 32 bit
- More cache activity, effects power

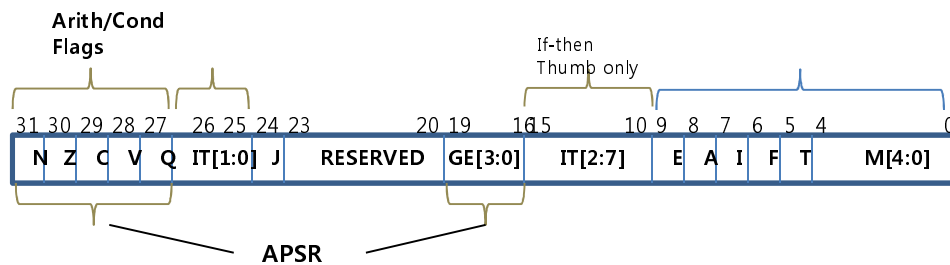
Exception Handling

```

vector_irq:
  ; use IRQ stack temporarily
  ; LR – user RA, SPSR user CPSR,
  - save LR, SPSR, r0 – ptr to irq stack
  ; changed modes to SVC mode
  - set spsr_mode to SVC mode
  ; movs in this context changes modes
  movs pc, lr

__irq_usr:
  ; svc mode
  - get LR, SPSR from irq stack save to SVC stack
  - get user lr, sp use {...}^ save to SVC stack
  - Dispatch IRQ handler
  ; restore user context
  - restore – r0-r12, {sp, lr}^
  - restore – user cpsr, - from spsr saved in irq mode
  - restore lr from one saved in irq mode
  movs pc, lr
  
```

- In irq mode most step off into SVC
- Irq mode can't take nested interrupts
 - LR – would be wiped out
 - Except HYP mode ELR_HYP



J,T – instruction mode ARM, Thumb, Jazelle, Jazelle-RCT
E – endiannes BE/LE – setend <le|be> then do a load
A – enable/disable Async faults due (memory errors)
I,F – irq, fiq mask bits

Modes

- Usr – 10000
- Fiq – 10001
- Irq – 10010
- Svc – 10011
- Mon – 10110
- Abt – 10111
- Und – 11011
- Sys – 11111
- Hyp – 11010 – hyp mode**

```

mrs r0, cpsr, or msr
msr cpsr_c, r0
  
```

Registers-ARMv8(no VFP/SIMD)

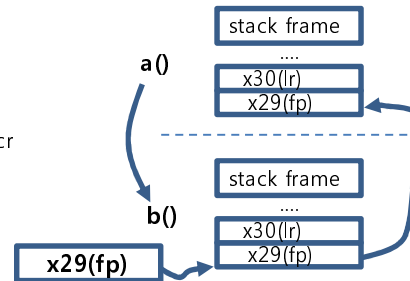
64b	
x0	w0
x1	w1
x2	w2
x3	w3
x4	w4
x5	w5
x6	w6
x6	w7
x7	w8
x8	w9
■	
■	
■	
x27	w27
x28	w28
x29(FP)	w29
x30(LR)	w30
SP/0	wsp
ELR	

Procedure call

```

a( ... )
; push LR, FP
stp x29, x30, sp[#-48] ; pre-incr
; set FP to current frame
mov x29, sp ; frame pointer
; pass arguments
mov w0, arg
...
bl b
ldp x29, x30, [sp], #48 ; post-incr
; return using LR
ret
b(...)
...

```



- Registers 64 bit, 32 bit variant 'wx'
- 'pc' – no longer part of register file
- store/load pair – used for stack ops (16 bytes)
- FP used to keep track of stack frame (x29)
- 'sp' can be stack pointer or 0 reg 'wzr'
 - Depends on context use
- LR per each priv. level
- Register set – way more usable regs
 - less cache references, lowers power

Exception Handling

Exception taken at EL0

ELO	EL1
sp_el0	sp_el1
	elr_el1
	spsr_el1

; look up cause – unlike ARMv7 cause/vector not 1:1

```

mrs x1, esr_el1
; determine reason
b.eq dabort
b.eq pabort
...

```

```

; restore SPSR, LR from spsr_el1, elr_el1
ERET

```

- Makes sense after exception model
- 'modes' collapsed – to exception levels
- each exception level – has banked sp, elr, spsr
- as well as – exception syndrome register
- Save state, process exception, restore SPSR, LR, ERET
- No hidden meanings for 's' bit i.e, movs pc, ..

PState →

| NZCV | DAIF | SPSel | CurrentEL → state accessible in aarch64 at run-time

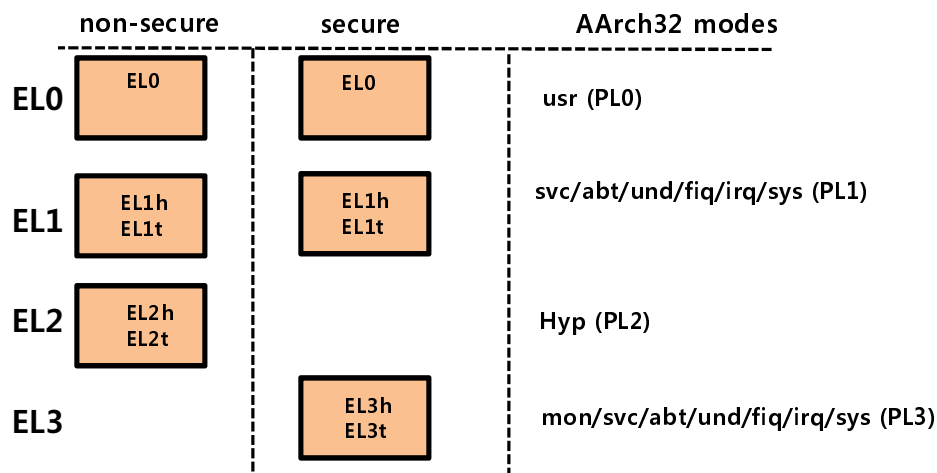
- Equivalent of CPSR in ARMv7
- **NZCV** – arith/cond flags
- **DAIF** – Debug, Async, Irq, Fiq masks
- **SPsel** – rare use
- **Current EL** – User, Kernel, HYP, Secure

- Pstate – fields more like registers
 - ; access field directly, Clear bit 2 disable IRQ
 - msr DAIFSet, #2
 - ;

- ... **BUT** PState is really – ELR_ElX, SP_ElX, SPSR_ElX, SPSR_abt. SPSR_fiq, SPSR_und, SPSR_abt – later on in exception model + VFP/SIMD, Debug

Exception Model

- v8 simplified the exception model vs v7?
 - State, Privilege, Security level confusing
 - Several usr, irq, fiq, svc, und, sys (also hyp, mon) –
 - Each had it's own stack, banked registers and briefly used
 - Also instruction state (J,T) – ARMv8 only arm64
 - Privilege – scattered over various states – usr – 0, system – to run privileged threads
 - Security level (TrustZone)
 - Secure apps by default have higher privilege
- ARMv8 (aarch64 Exception Model)
 - Simplified to Exception levels, higher values higher privilege
 - No modes a SP, LR, SPSR per each mode
 - Pstate CurrentEL → Exception Level



Exception Model – v7/v8 vectors

- With v8 Exception model vectors changes
- V7 vectors – entry per mode;

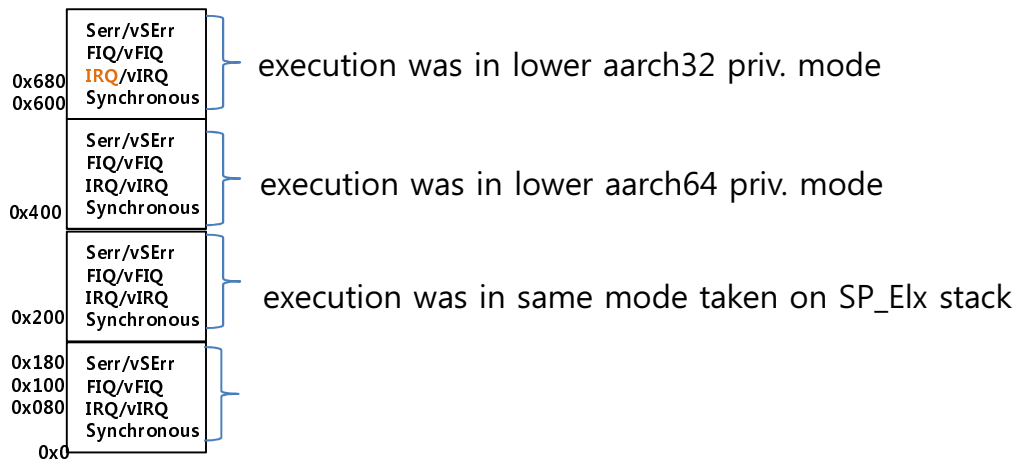
Non-Secure

Exception Table (tables differ depending on HYP, Monitor, Secure, Non-Secure)

0x00	Reset	
0x04	Undef Instruction	Exc. Addr → LR_und, SP_und → SP, CPSR → SPSR_und, - user SP & LR {SP, LR}^
0x08	call – SVC	Exc. Addr → LR_svc, SP_svc → SP, CPSR → SPSR_svc, - user SP & LR {SP, LR}^
0x0C	prefetch abort	Exc. Addr → LR_abt, SP_svc → SP, CPSR → SPSR_svc, - user SP & LR {SP, LR}^
0x10	data abort	Exc. Addr → LR_abt, SP_svc → SP, CPSR → SPSR_svc, - user SP & LR {SP, LR}^
0x14	-	Exc. Addr → LR_abt, SP_svc → SP, CPSR → SPSR_svc, - user SP & LR {SP, LR}^
0x18	IRQ	Exc. Addr → LR_irq, SP_irq → SP, CPSR → SPSR_irq, - user SP & LR {SP, LR}^
0x1C	FIQ	

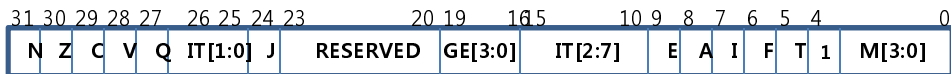
• V8 vectors

- Exception entry several possibilities
 - Executing AArch32 lower mode taken in higher mode AArch64 (i.e. virtualization, secure mode)
 - Executing AArch64 lower taken in higher mode AArch54 (i.e. virtualization, secure mode)
 - Exception taken to same level
 - Advantage – quicker dispatch and handling, table per each level

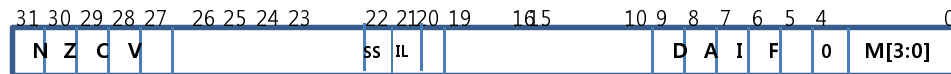


Exceptions – accessing previous state

- Additional PState
- Into EL1 → ELR_EL1, SPSR_EL1, SPSR_{abt, fiq, irq, und}, SP_EL1 used in EL1 mode
 - From EL0
- Into EL2 → ELR_EL2, SPSR_EL2, SP_EL2
- Also secure mode
- Two version of SPSR – one for aarch32 other for aarch64
- Aarch32



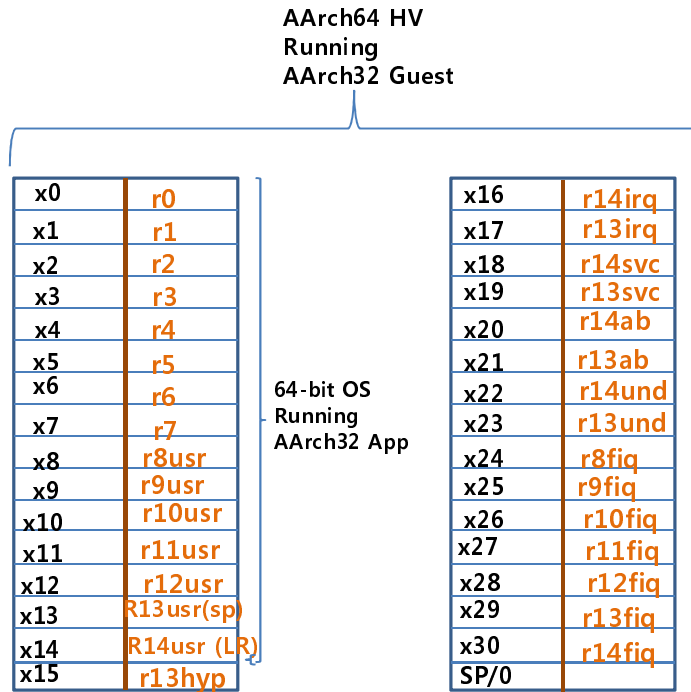
- T, J – tells you what instruction set was running, how to restore guest, what registers are valid
- Aarch64 – resembles run-time PState



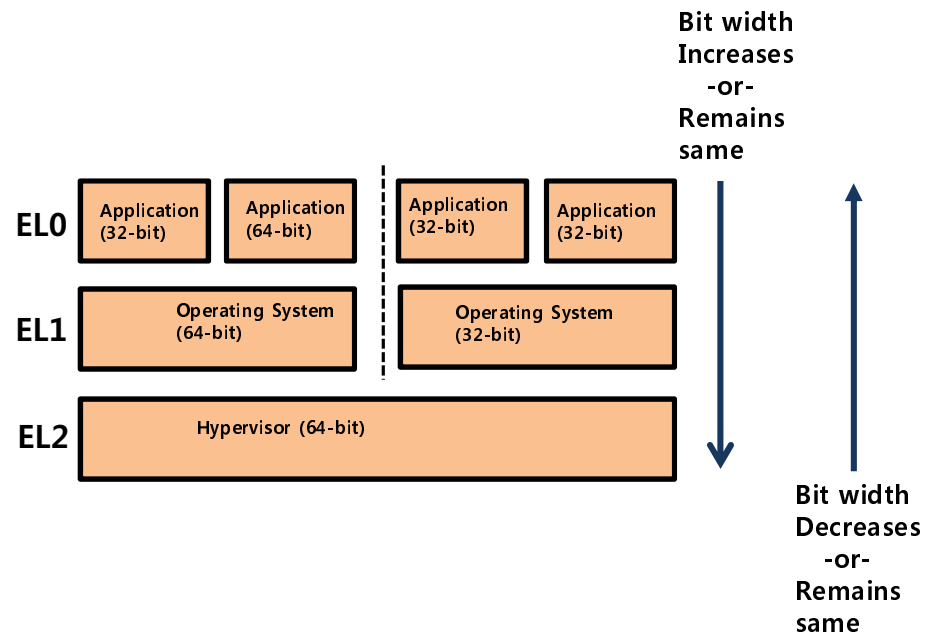
- **M[3:0] → 3,2=EL; 1=0; 0=SP_ELx or SP_ELO – what mode you came from**
 - **3:0 → 0x0** – EL0 running SP_ELO
 - **→ 0x5** – EL1 running
 - **→ 0x9** – EL2 running
 - There is also EL3 for secure mode

Exceptions – accessing aarch32 state

- If you trap from aarch32 into aarch64 you need aarch32 register state



- Bit size change only through exceptions

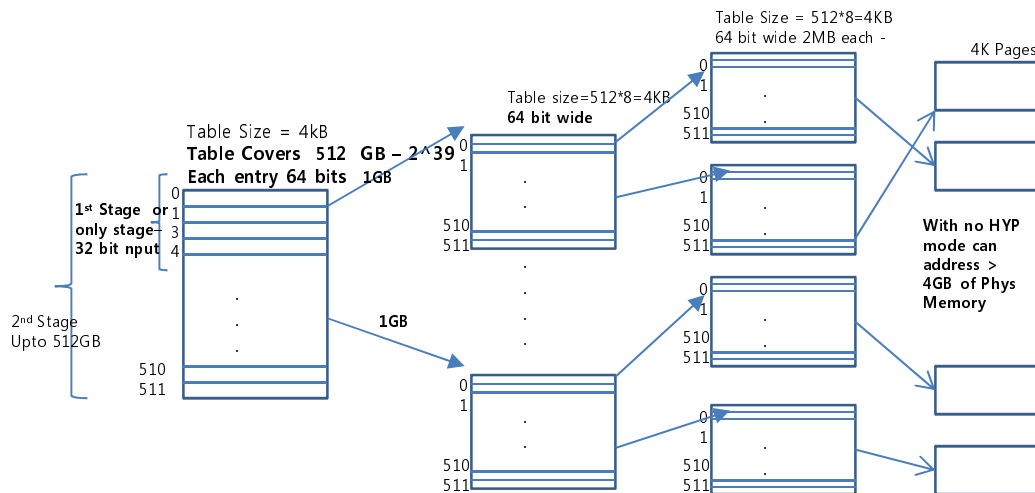


Instruction Set

- In addition to ARM, ARMv7 supports
 - Thumb2 – mix of thumb/arm instructions – compact code
 - Jazelle DBX (direct byte execution)
 - Some or no Java byte codes implemented, needs custom JVM
 - JVM – BXJ – enter Jazelle – use software assists for unimplemented byte codes
 - Jazelle RCT (run time compilation target)
 - Optimized for JIT/JVM – instruction set aids language (i.e. array boundary checks)
 - After byte codes compiled – ENTERX/LEAVEX – enters/leaves ThumbEE
 - ARMv8 does not support these modes – but you can run in ARMv7 mode on ARMv8 cpu
- Contrasting Instructions
 - ARMv8 new clean instruction set
 - Predication removed – i.e. `moveq r1, r2`
 - `movs` on exceptions – no applicable in ARMv8
 - Removal of Coprocessors – for example `dccmvac` – v7 `p15,0,Rt,c7,c0,1`; v8 `dc cvac`
 - GIC (general interrupt controller) – register for CPU, Dist., Redistr. – not memory mapped
 - Stack – armv8 – `ldrp/strp` – min. 16 – 2 regs push/pop – 16 byte aligned
 - `strex/ldrex` – lockless semaphores, local/global monitor – has been around in v7
 - No bus locking `swp`

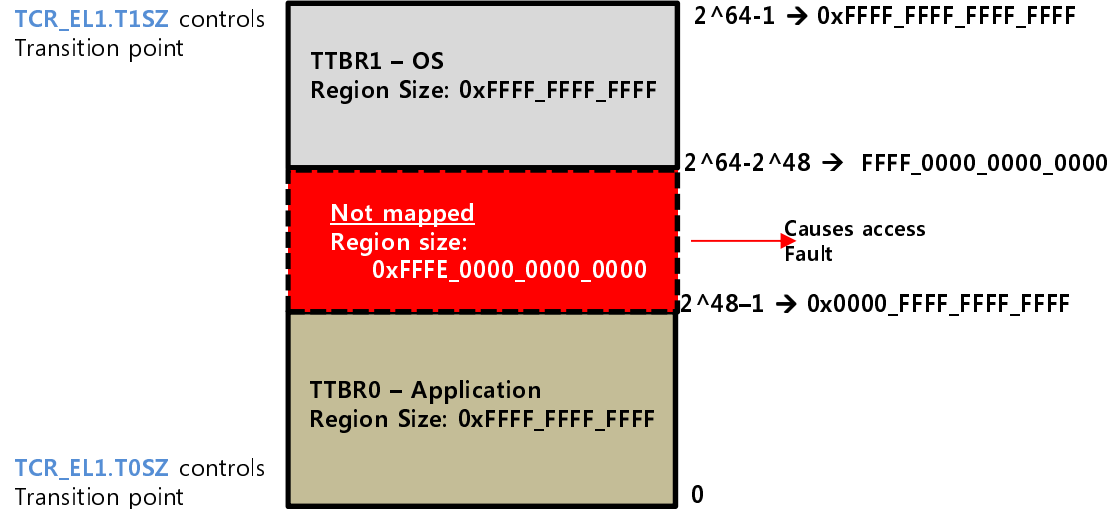
MMU – ARMv7

- ARMv7 – Rev C – introduced LPAE – precursor to ARMv8
- 1st level pages 32-bit input VA (2^{32}) output upto 40 bit PA range
- For 2nd stage (Virtualization – some more later) input range is 2^{40}
- App developer not much difference – except you may support larger DBs for example
- You can still run old VMSA table format – i.e. supersections,
- For application usual 3GB/1GB split
- TTBR0, TTBR1, TTBCR controls size of tables, inner/outer page table cacheability
 - TTBR1 – kernel, TTBR0 user – context switched



- Few additions to sections/pages & tables
 - APTable – permissions for next lever page tables
 - PXNTable, PXN – prevent priv. execution from non-priv memory
 - XNTable, XN – prevent execution period
 - Few other for Secure/Non-Secure settings

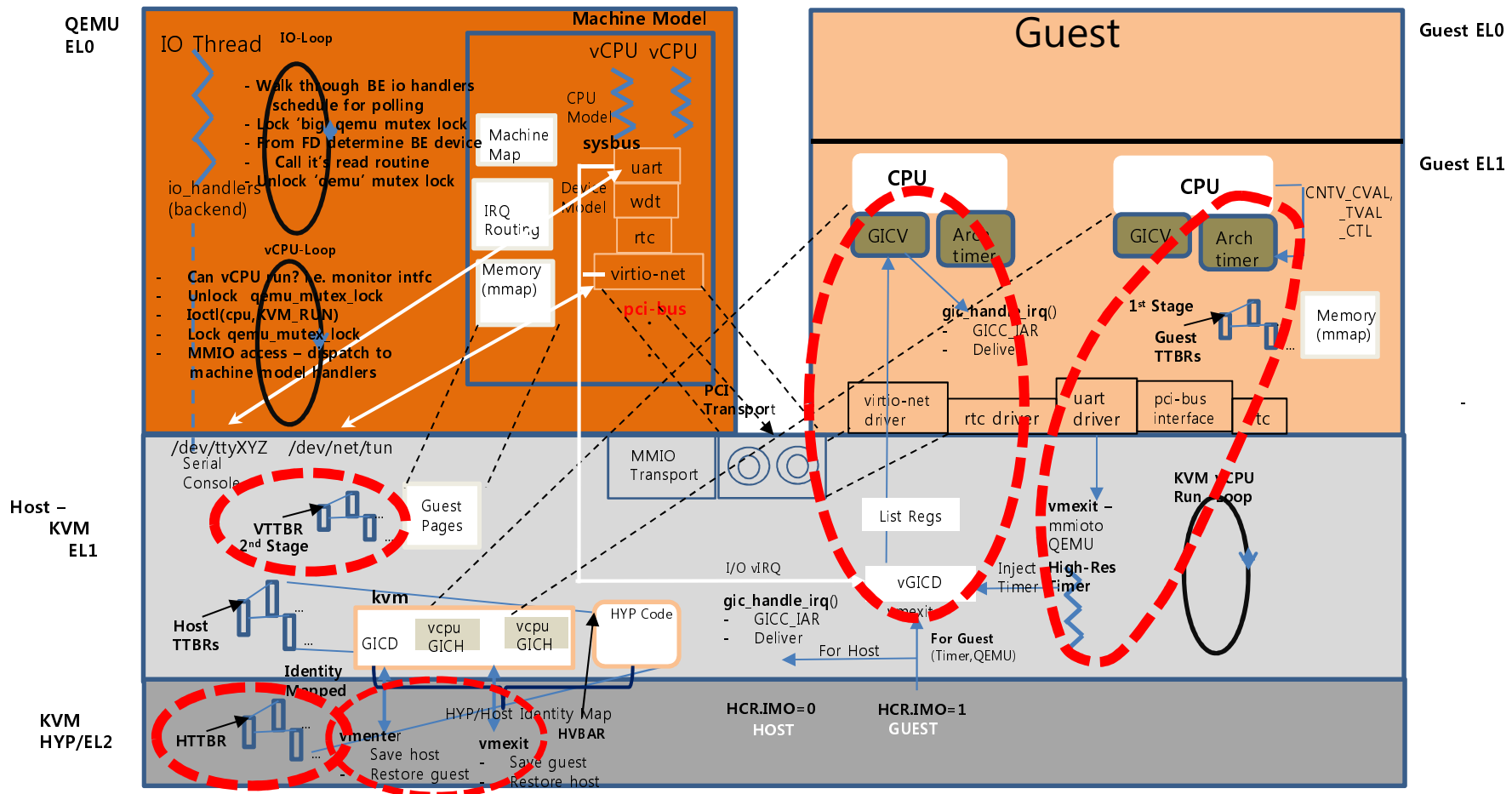
ARMv8 MMU



- Input VA range 49 bits – upper bit sign extended
- Output PA impl. dependent (48 bits)
- 2 page table types – 4Kb, 64Kb pages
- 4 & 3 level tables – 4Kb and 64Kb page table sizes
 - 64Kb format – fewer TLB faults
- Both formats for 1st and 2nd stage tables supported
- Tool chains take care of expanding addresses
- Page table formats a lot like LPAE

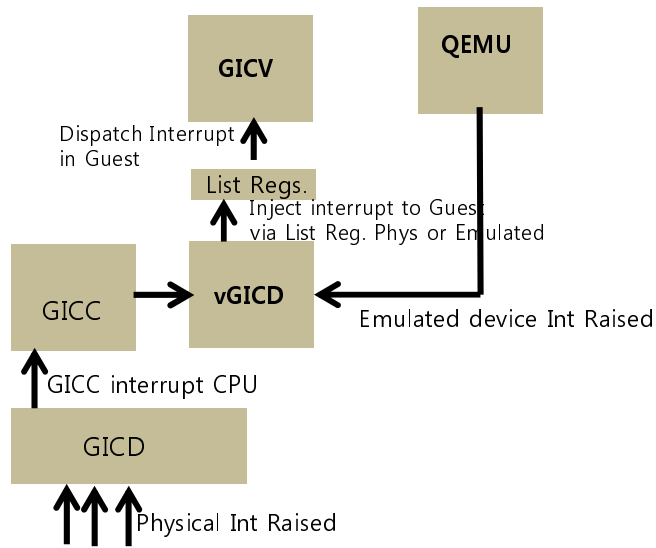
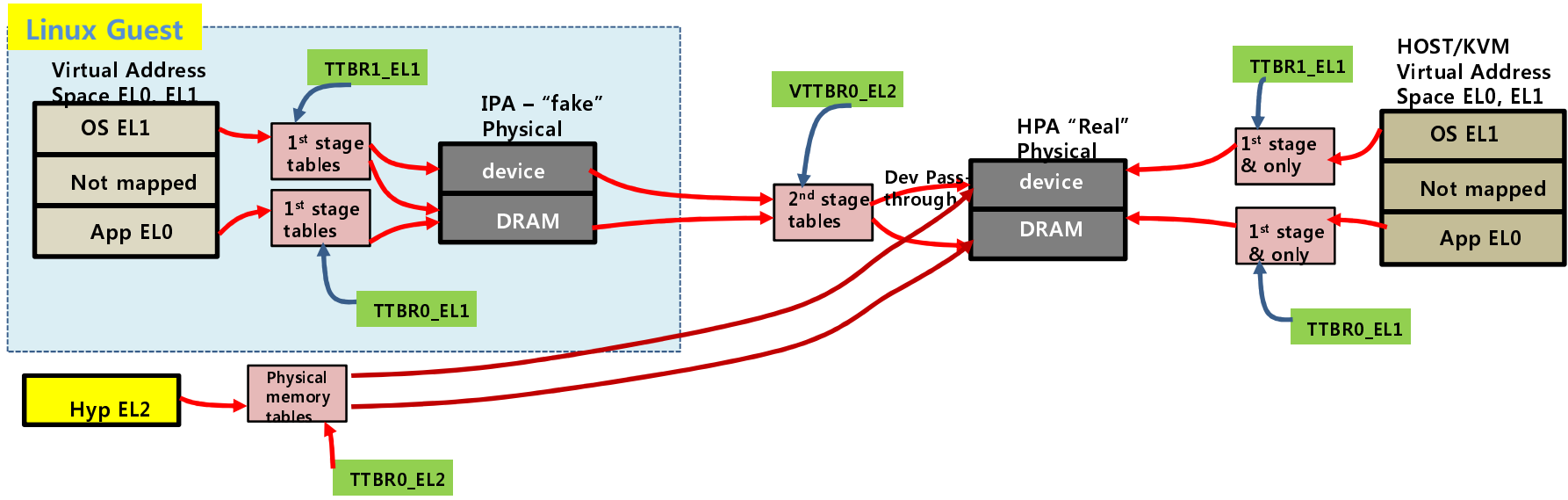
Virtualization – big picture w/QEMU

- Big Picture – with virtualization besides another priv-level 2 you add
 - 2nd stage mmu tables, virtual interrupts & timers, emulated/virtio devices, pass-through devices...
 - HYP or PL2 privilege level & it's mmu, config regs, and KVM
 - ARMv8 more complexity – handle v7/v8 guests – v7 guests several instruction sets & 32 bit
 - Set Exit reason – HCR reg., Decode exit reason – HSR register (armv8 EL2 regs)
 - World switch sw controlled save/restore as much or little as you need



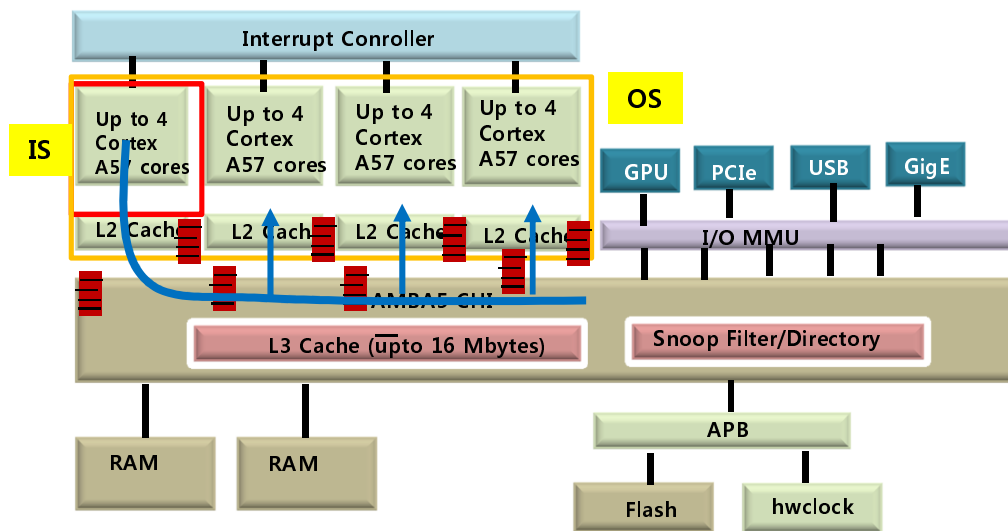
MMU/Interrupts with Virtualization

- With Virtualization things get complex – thanks to hw extensions – you get near native



ARMv8 more than just processor - interconnect

- AMBA5 CHI (coherent hub interface)
 - Goes beyond AXI4/3 – handle big server
 - Message based, clock scales to cluster – terabit fabric – flow control with buffers
 - Several coherency options – snoop filter, directory
 - Should support for PCI Config Cycles (not pre-defined address ranges)
 - Bearing on Cache Coherency, TLB management, Barriers



- Cache line invalidate –
 - dci x0 – miss broadcast snoop to clusters
 - Tags update on response for snoop requests
- Barriers
 - isb, dsb, dmb
 - driven to all cluster – wait for completion side effect, memory retire

Caches Shareability, Coherence

- Both v7, v8 have these concepts – v7 since AXIv4
- Inner/Outer shareability –
 - Inner shareable – cluster, outer shareability all clusters
 - Boundaries really SoC dependent
 - OS spanning multiple clusters – inner, outer same & shareable
 - Cache coherence, cache op. attributes (WB,WT) manage by page tables
 - barrier operations – must specify scope 'ish', 'osh'
- PoU or PoC
 - Point of Union – mem location visible to all CPU Instr/Data caches, TLB walks
 - Point of Coherence – all agents see the same values – i.e I/O
 - suffix 'u', 'c' used for cache operations

Running ARMv8

- Reference <https://github.com/mjsmar/running-arm64-howto.git>
 - Summary – on Ubuntu 14.04 install gcc-aarch64-linux-gnu – see reference includes guest build too
 - Contains config files, host/guest rootfs ...
-
- Download Foundation Model - the host platform - must register with ARM
 - Get a minimum FS i.e. linaro-image-minimal-generic-armv8-20130526-319.rootfs.tar.gz
 - Setup NFS environment – change to root
 - mkdir /srv/armv8rootfs; cd /srv/armv8rootfs; tar xzf path/linarm-image*.gz
 - For example use Host IP 192.168.0.115
 - add to /etc/exports: /srv/ 192.168.0.115/255.255.255.0(rw sync no_root_squash no_subtree_check insecure)
 - Build host kernel – config select VIRTIO_BLK, SCSI_BLK, VIRTIO_NET, HVC_DRIVER, VIRTIO_CONSOLE, VIRTIO, VIRTIO_MMIO; enable NFS Boot
 - Download recent kernel >= 3.17 – make arch=arm64 CROSS_COMPILE=aarch64-linux-gnu- defconfig
 - Followed by 'menuconfig' select options (or use prepared config file) and 'Image'
 - Get boot wrapper – clone git://git.kernel.org/pub/scm/linux/kernel/git/cmarinas/boot-wrapper-aarch64.git
 - Mini boot loader Foundation model first runs, in boot wrapper directory
 - Create links – dtc → kernel dtc directory; foundation-v8.dts → arch/arm64/boot/dts/foundat-v8.dts
Image → arch/arm64/boot/Image; rtsm_ve-motherboard.dts → ...; rtsm_ve-aemv8a.dts → ...
skeleton.dtsi → ...
 - # kernel command lines to boot from NFS
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-
BOOTARGS="root=/dev/nfs nfsroot=192.168.0.115:/srv/armv8rootfs/ rw ip=192.168.0.151:192.168.0.115:
192.168.0.115:255.255.255.0:kvmfm:eth0 console=ttyAMA0" FDT_SRC=foundation-v8.dts
 - You should have linux-system.axf
 - Create tap device – tunctl; create bridge add tapX to bridge; set brX IP to 192.168.0.115
 - Restart NFS services – sudo service nfs-kernel-server restart;
 - showmount -e 192.168.0.115 – should see nfs mount point; add Foundation_v8 to your PATH
 - Run: Foundation_v8 -image <path>/linux-system.axf -network bridged -network-bridge=tap0
 - Should be able to ssh now
-
- see reference how to build kvmtool and run guest in Foundation Model; may add qemu build too - later



Transitioning from ARM v7 to ARM v8: All the basics you must know.



Q & A

Thank you.

Mario Smarduch
Senior Virtualization Architect
Open Source Group
Samsung Research America (Silicon Valley)
m.smarduch@samsung.com