Reasoning about Translation Lookaside Buffers (TLBs)

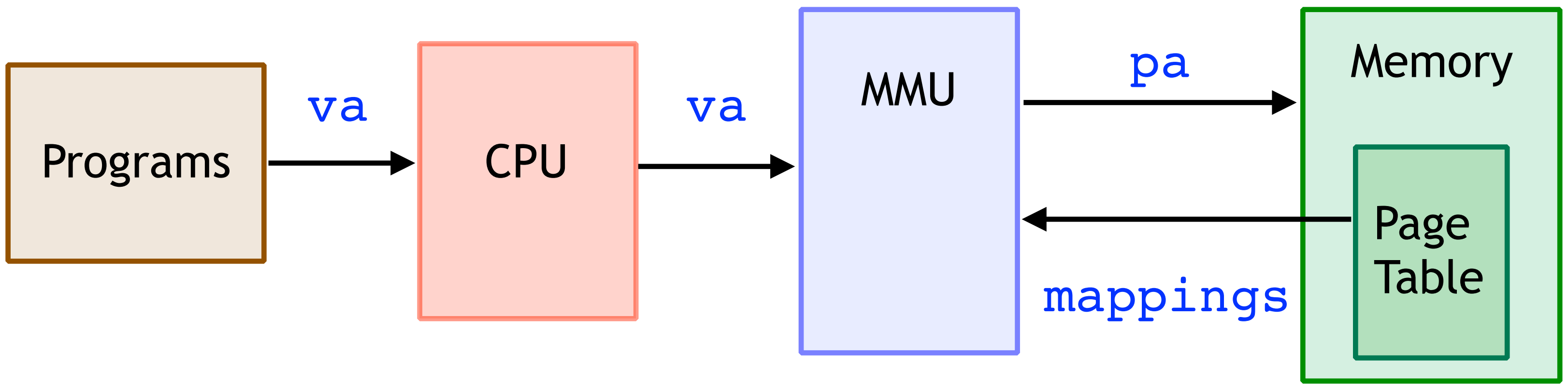
LPAR-21

Hira T. Syeda and Gerwin Klein
Trustworthy Systems @ Data61

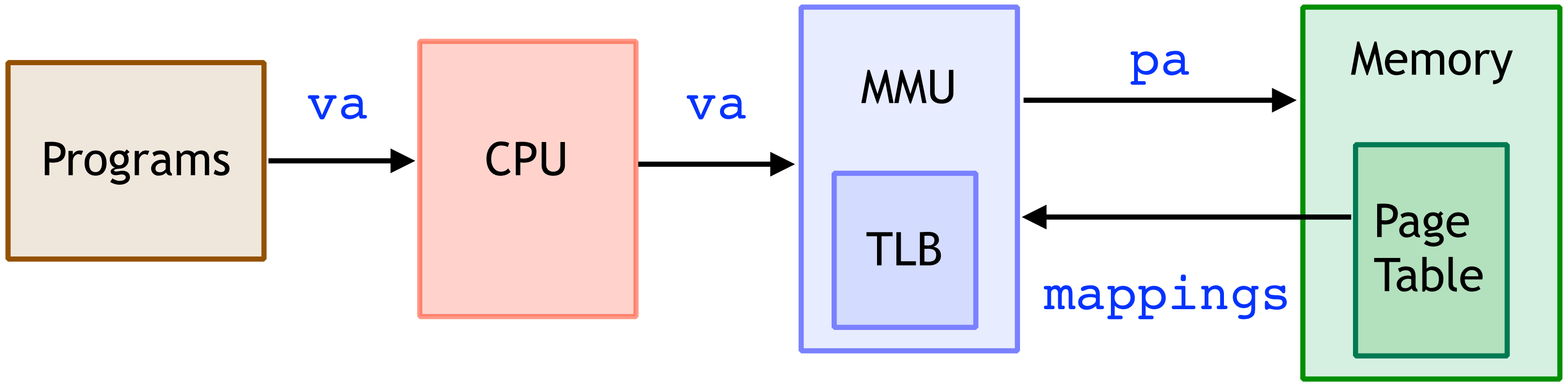
May 2017

www.ts.data61.csiro.au

What is a TLB



What is a TLB



- TLB
  - dedicated cache for page table walks
  - architecture specific

TLB Effects on Program Execution

TLB Effects on Program Execution

- TLB being cache
  - has *no* functional effects
  - only makes execution faster, *if* maintained correctly
  - is an assumption in seL4 proofs

TLB Effects on Program Execution

- TLB being cache
  - has *no* functional effects
  - only makes execution faster, *if* maintained correctly
  - is an assumption in seL4 proofs

- Programs must avoid accessing
  - incoherent entries
  - inconsistent entries

TLB Effects on Program Execution

- TLB being cache
  - has *no* functional effects
  - only makes execution faster, *if* maintained correctly
  - is an assumption in seL4 proofs

- Programs must avoid accessing
  - incoherent entries
  - inconsistent entries

- Deserves support by the hardware model

TLB Effects on Program Execution

- TLB being cache
  - has *no* functional effects
  - only makes execution faster, *if* maintained correctly
  - is an assumption in seL4 proofs

- Programs must avoid accessing
  - incoherent entries
  - inconsistent entries

- Deserves support by the hardware model

- Extend seL4 program logic for TLB reasoning
  - a formal TLB model for ARMv7 architecture

Contributions

- Formal model of ARMv7-style TLB in Isabelle/HOL
  - `lookup` function

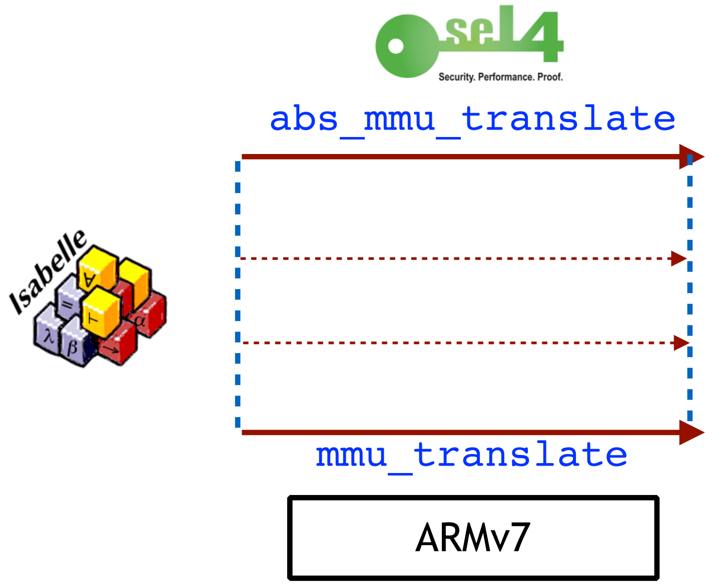
Contributions

- Formal model of ARMv7-style TLB in Isabelle/HOL
  - `lookup` function

- Extension to MMU model
  - `mmu_translate`, `mmu_read`, `mmu_write`
  - integration to ARMv7 formalised ISA

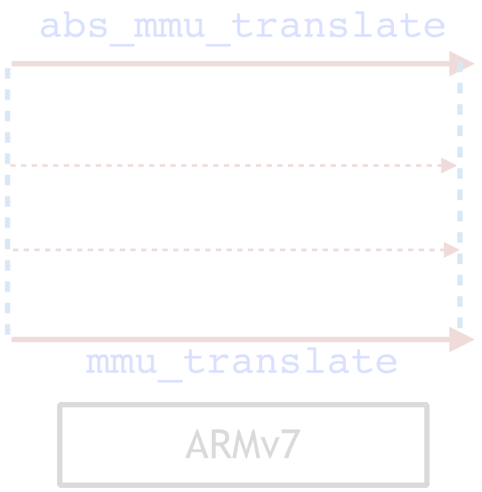
Contributions

- Formal model of ARMv7-style TLB in Isabelle/HOL
    - `lookup` function
- Extension to MMU model
    - `mmu_translate`, `mmu_read`, `mmu_write`
    - integration to ARMv7 formalised ISA
- Data refinement for
    - abstracting hardware details
    - easier reasoning

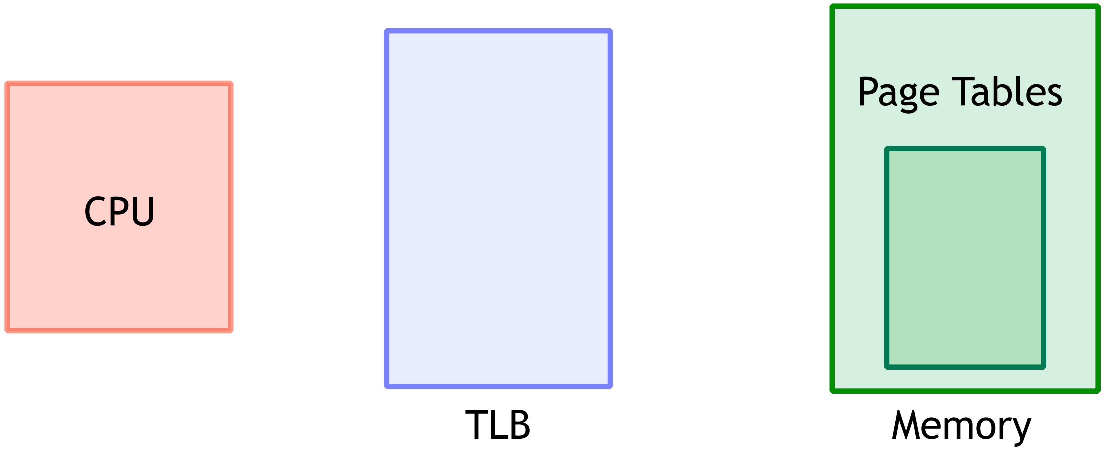


Contributions

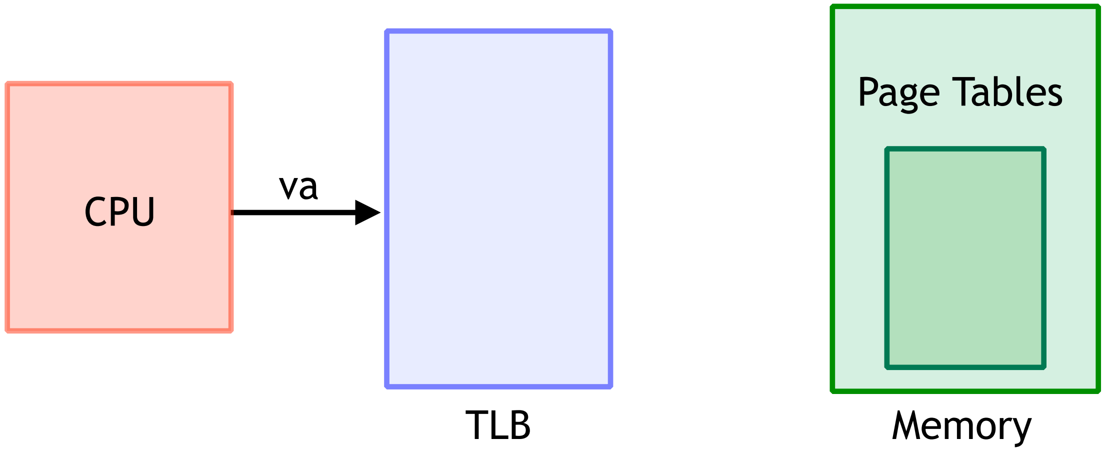
- Formal model of ARMv7-style TLB in Isabelle/HOL
  - `lookup` function
- Extension to MMU model
  - `mmu_translate`, `mmu_read`, `mmu_write`
  - integration to ARMv7 formalised ISA
- Data refinement for
  - abstracting hardware details
  - easier reasoning



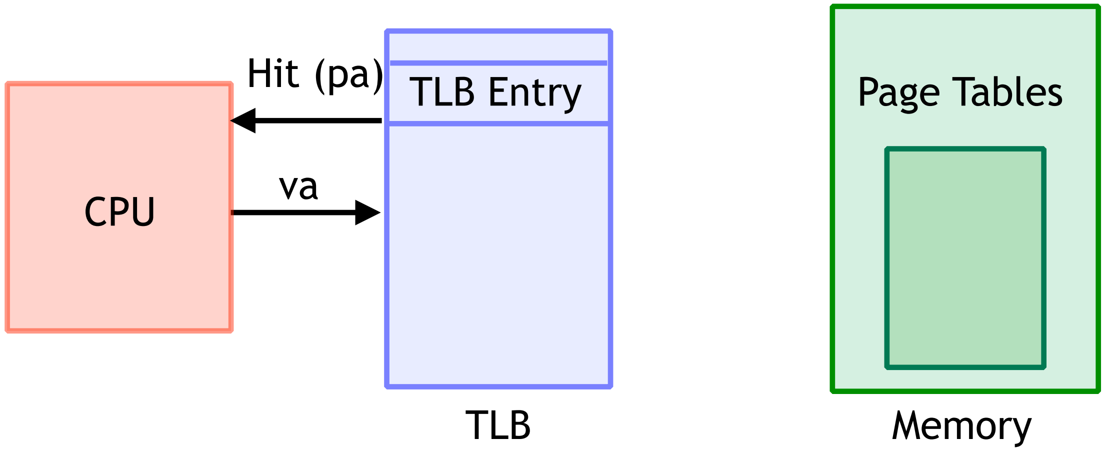
ARMv7-style TLB



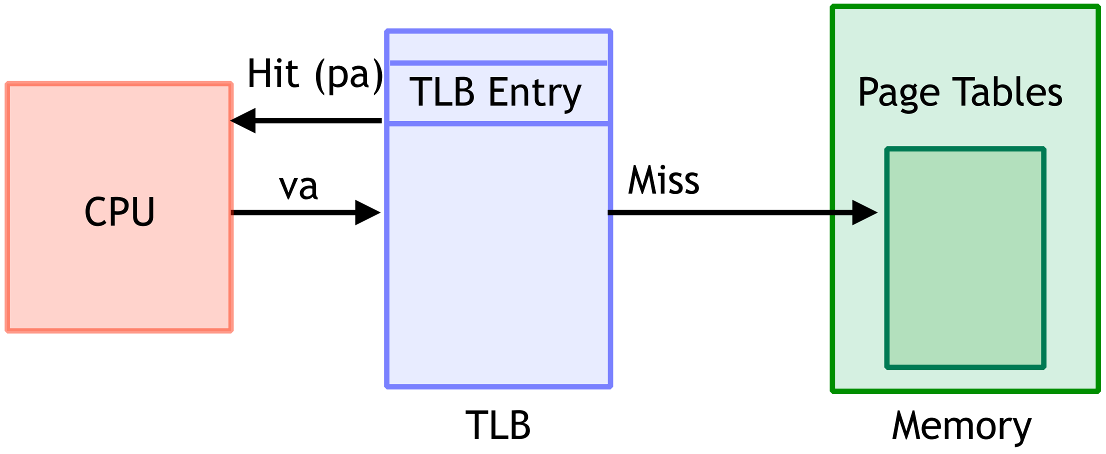
ARMv7-style TLB



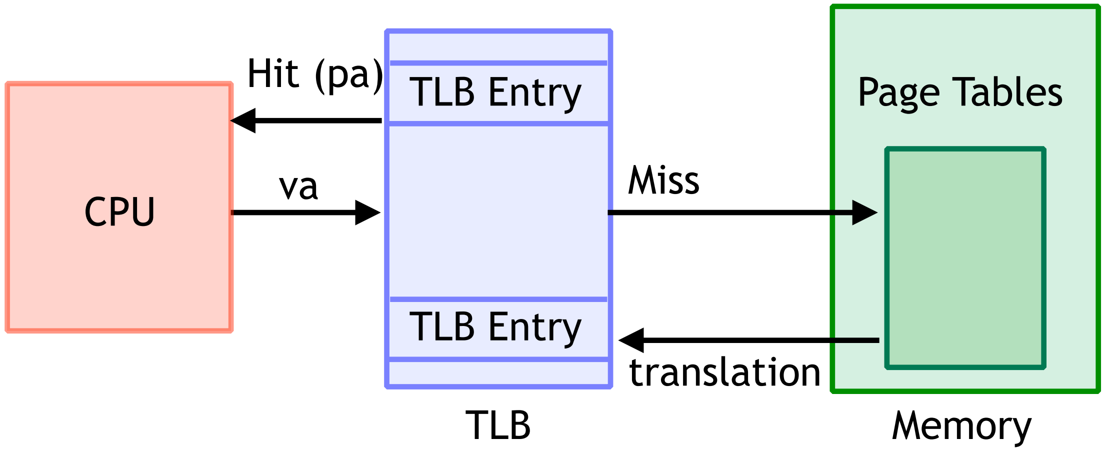
ARMv7-style TLB



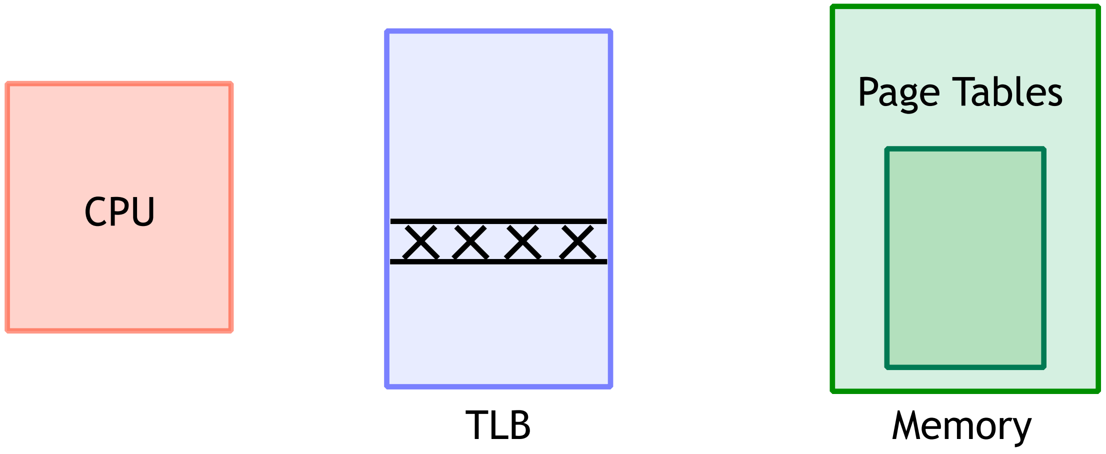
ARMv7-style TLB



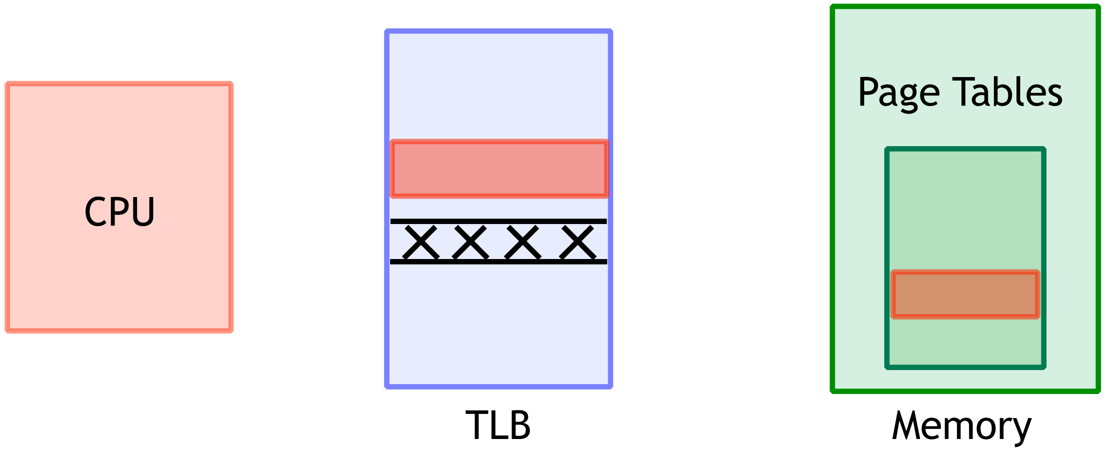
ARMv7-style TLB



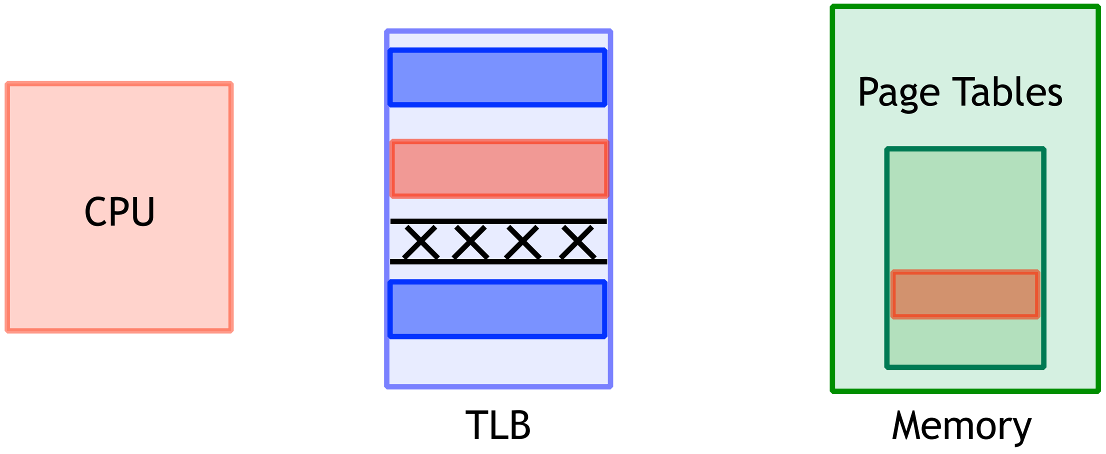
ARMv7-style TLB



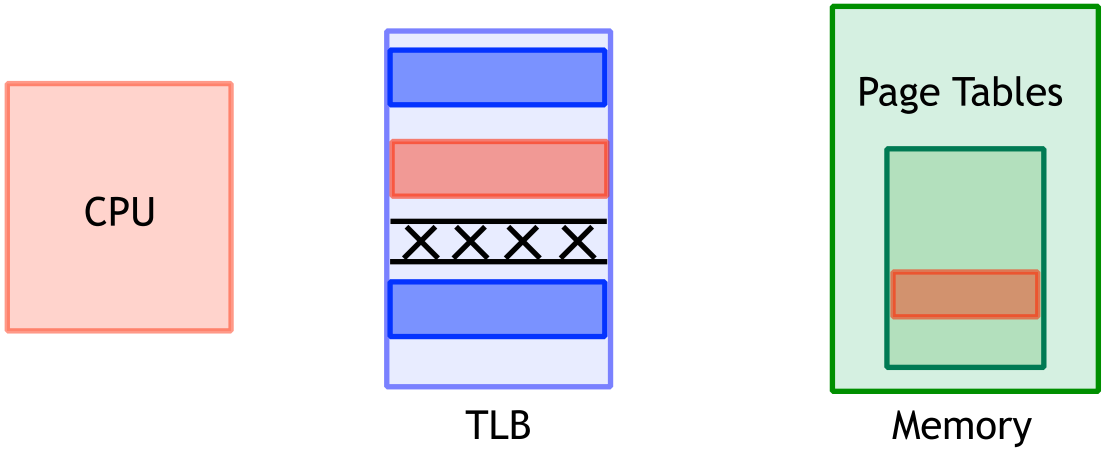
ARMv7-style TLB



ARMv7-style TLB



ARMv7-style TLB



- TLB maintenance operations
  - flush entries after write to page table
  - lazy invalidation
    - Address Space IDentifier - ASID

ARMv7-style Formal Model of TLB

```
lookup :: tlb ⇒ asid ⇒ vaddr ⇒ lookup_type
```

ARMv7-style
Formal Model of TLB

```
lookup :: tlb ⇒ asid ⇒ vaddr ⇒ lookup_type
```

ARMv7-style Formal Model of TLB

```
lookup :: tlb ⇒ asid ⇒ vaddr ⇒ lookup_type
```

```
datatype lookup_type = Miss | Incon | Hit tlb_entry
```

ARMv7-style Formal Model of TLB

```
lookup :: tlb ⇒ asid ⇒ vaddr ⇒ lookup_type
```

```
datatype lookup_type = Miss | Incon | Hit tlb_entry
```

```
type_synonym tlb = tlb_entry set
```

ARMv7-style Formal Model of TLB

```
lookup :: tlb ⇒ asid ⇒ vaddr ⇒ lookup_type

datatype lookup_type = Miss | Incon | Hit tlb_entry

type_synonym tlb = tlb_entry set
```

tlb_entry format

ASID	virtual base address	physical base address	permission bits

ARMv7-style
Formal Model of TLB

```
lookup :: tlb ⇒ asid ⇒ vaddr ⇒ lookup_type
```

```
datatype lookup_type = Miss | Incon | Hit tlb_entry
```

```
type_synonym tlb = tlb_entry set
```

tlb_entry format

ASID	virtual base address	physical base address	permission bits

```
datatype tlb_entry  =  EntrySmall asid (20 word) (20 word option) flags
                    |  EntrySection asid (12 word) (12 word option) flags
```

ARMv7-style Formal Model of TLB

```
lookup :: tlb ⇒ asid ⇒ vaddr ⇒ lookup_type
```

```
datatype lookup_type = Miss | Incon | Hit tlb_entry
```

```
type_synonym tlb = tlb_entry set
```

tlb_entry format

ASID	virtual base address	physical base address	permission bits

```
datatype tlb_entry  =  EntrySmall asid (20 word) (20 word option) flags
                    |  EntrySection asid (12 word) (12 word option) flags
```

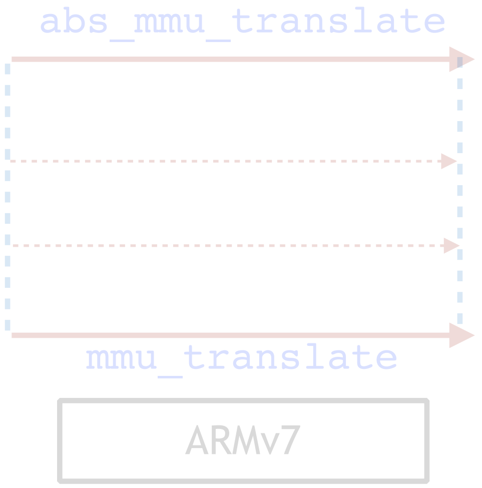
flush operations
- selective_invalidation
- asid_invalidation
- va_invalidation

Contributions

- Formal model of ARMv7-style TLB in Isabelle/HOL
    - `lookup` function

- Extension to MMU model
    - `mmu_translate`, `mmu_read`, `mmu_write`
    - integration to ARMv7 formalised ISA

- Data refinement for
    - abstracting hardware details
    - easier reasoning



From TLB to MMU

From TLB to MMU

- ARM Instruction Set Architecture (ISA) semantics
  - L3 - specification language for ISAs
  - detailed but
    - no address translation

From TLB to MMU

- ARM Instruction Set Architecture (ISA) semantics
  - L3 - specification language for ISAs
  - detailed but
    - no address translation

- State extension - `tlb`

From TLB to MMU

- ARM Instruction Set Architecture (ISA) semantics
  - L3 - specification language for ISAs
  - detailed but
    - no address translation

- State extension - `tlb`
- `mmu_translate` function

From TLB to MMU

- ARM Instruction Set Architecture (ISA) semantics
  - L3 - specification language for ISAs
  - detailed but
    - no address translation

- State extension - `tlb`
- `mmu_translate` function
- Update `mem_write` and `mem_read` functions

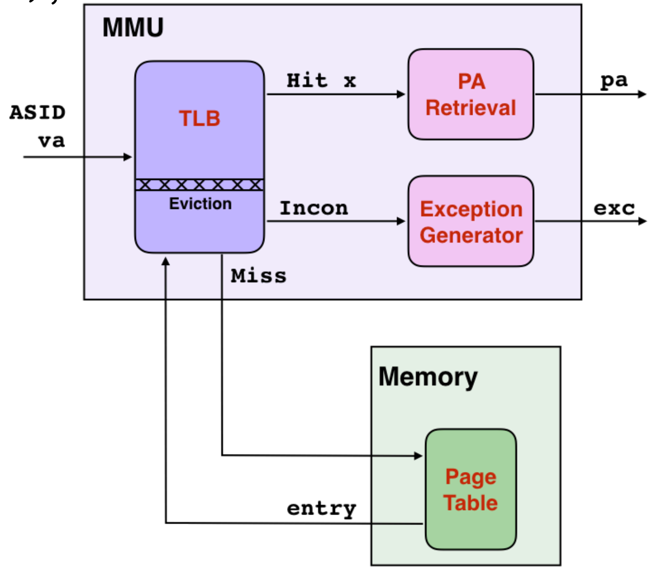
From TLB to MMU

- ARM Instruction Set Architecture (ISA) semantics
  - L3 - specification language for ISAs
  - detailed but
    - no address translation

- State extension - `tlb`
- `mmu_translate` function
- Update `mem_write` and `mem_read` functions
- Virtualize the subsequent memory accesses

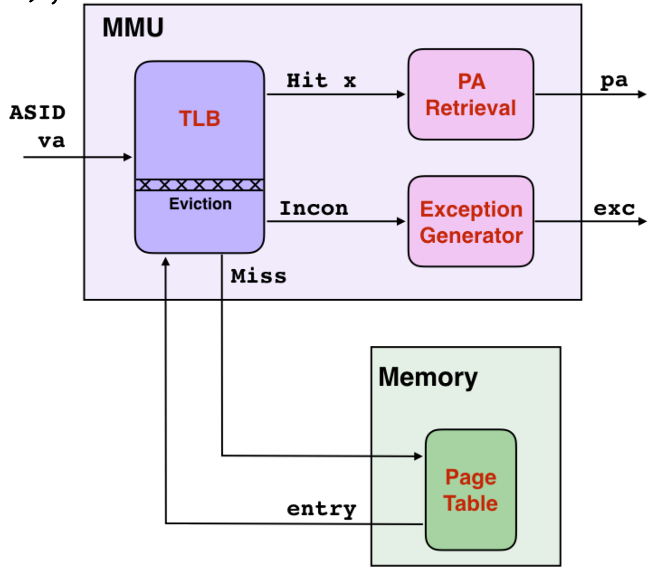
MMU and Memory Functions

```
mmu_translate va = do {
  update_state (λs. s⦇tlb := tlb s - tlb_evict s⦈);
  (mem, asid, ttbr0, tlb) ← read_state (MEM, ASID, TTBR0, tlb);
  case lookup tlb asid (addr_val va) of
  Miss ⇒
    let entry = pt_walk asid mem ttbr0 va
    in if is_fault entry then raise PAGE_FAULT
       else do {
              update_state (λs. s⦇tlb := tlb ∪ {entry}⦈);
              return (va_to_pa va entry)
            }
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒
      if is_fault entry then raise PAGE_FAULT
      else return (va_to_pa va entry)
}
```



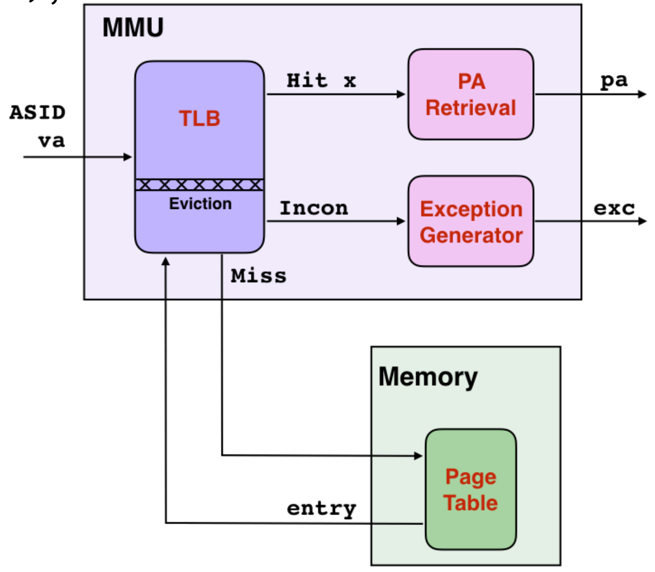
MMU and Memory Functions

```
mmu_translate va = do {
  update_state (λs. s⦇tlb := tlb s - tlb_evict s⦈);
  (mem, asid, ttbr0, tlb) ← read_state (MEM, ASID, TTBR0, tlb);
  case lookup tlb asid (addr_val va) of
  Miss ⇒
    let entry = pt_walk asid mem ttbr0 va
    in if is_fault entry then raise PAGE_FAULT
       else do {
             update_state (λs. s⦇tlb := tlb ∪ {entry}⦈);
             return (va_to_pa va entry)
          }
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒
      if is_fault entry then raise PAGE_FAULT
      else return (va_to_pa va entry)
}
```



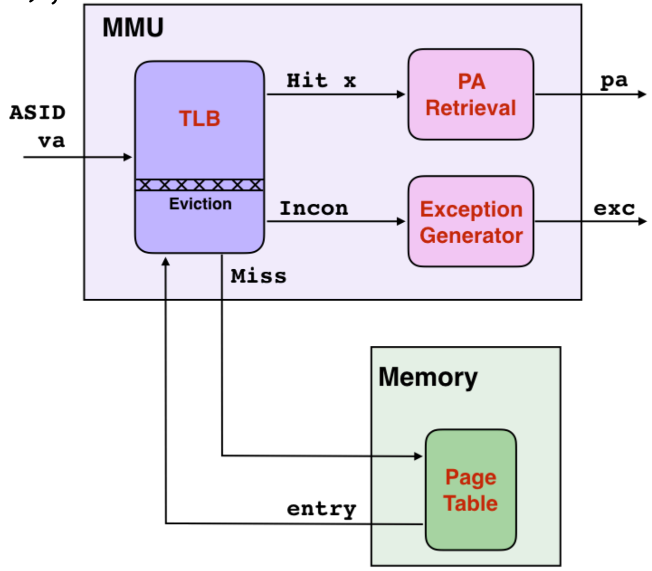
MMU and Memory Functions

```
mmu_translate va = do {
  update_state (λs. s(|tlb := tlb s - tlb_evict s|));
  (mem, asid, ttbr0, tlb) ← read_state (MEM, ASID, TTBR0, tlb);
  case lookup tlb asid (addr_val va) of
  Miss ⇒
    let entry = pt_walk asid mem ttbr0 va
    in if is_fault entry then raise PAGE_FAULT
       else do {
             update_state (λs. s(|tlb := tlb ∪ {entry}|));
             return (va_to_pa va entry)
           }
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒
      if is_fault entry then raise PAGE_FAULT
      else return (va_to_pa va entry)
}
```



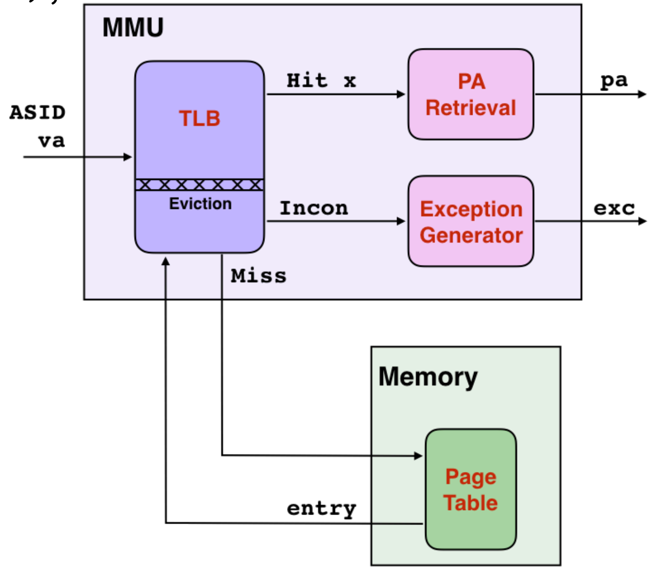
MMU and Memory Functions

```
mmu_translate va = do {
  update_state (λs. s(|tlb := tlb s - tlb_evict s|));
  (mem, asid, ttbr0, tlb) ← read_state (MEM, ASID, TTBR0, tlb);
  case lookup tlb asid (addr_val va) of
  Miss ⇒
    let entry = pt_walk asid mem ttbr0 va
    in if is_fault entry then raise PAGE_FAULT
       else do {
             update_state (λs. s(|tlb := tlb ∪ {entry}|));
             return (va_to_pa va entry)
           }
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒
      if is_fault entry then raise PAGE_FAULT
      else return (va_to_pa va entry)
}
```



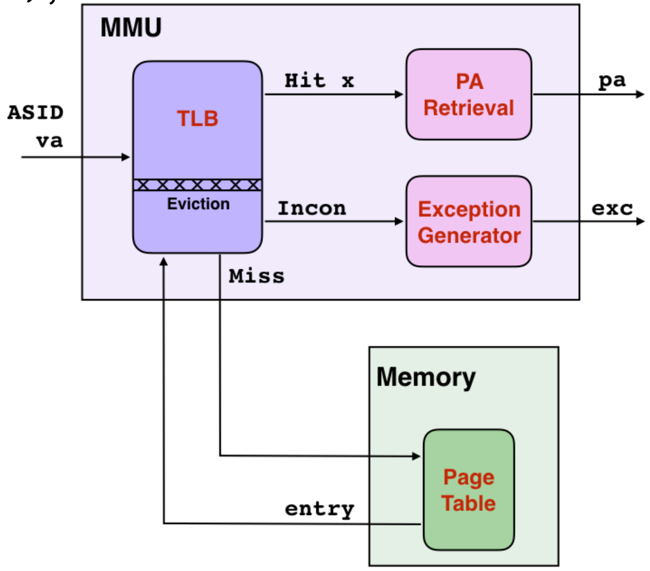
MMU and Memory Functions

```
mmu_translate va = do {
  update_state (λs. s(|tlb := tlb s - tlb_evict s|));
  (mem, asid, ttbr0, tlb) ← read_state (MEM, ASID, TTBR0, tlb);
  case lookup tlb asid (addr_val va) of
  Miss ⇒
    let entry = pt_walk asid mem ttbr0 va
    in if is_fault entry then raise PAGE_FAULT
       else do {
             update_state (λs. s(|tlb := tlb ∪ {entry}|));
             return (va_to_pa va entry)
           }
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒
      if is_fault entry then raise PAGE_FAULT
      else return (va_to_pa va entry)
}
```



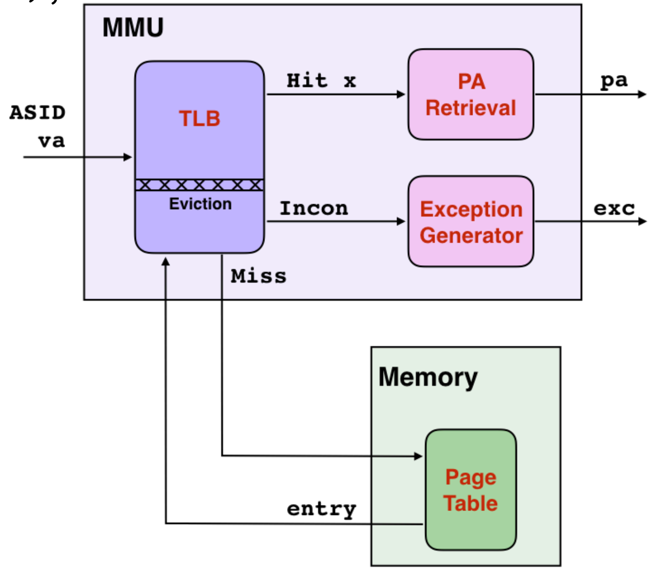
MMU and Memory Functions

```
mmu_translate va = do {
  update_state (λs. s⦇tlb := tlb s - tlb_evict s⦈);
  (mem, asid, ttbr0, tlb) ← read_state (MEM, ASID, TTBR0, tlb);
  case lookup tlb asid (addr_val va) of
  Miss ⇒
    let entry = pt_walk asid mem ttbr0 va
    in if is_fault entry then raise PAGE_FAULT
       else do {
              update_state (λs. s⦇tlb := tlb ∪ {entry}⦈);
              return (va_to_pa va entry)
            }
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒
      if is_fault entry then raise PAGE_FAULT
      else return (va_to_pa va entry)
}
```



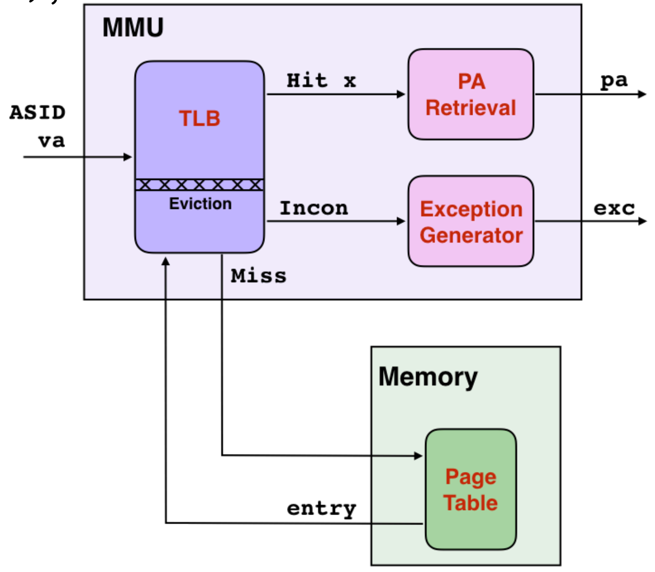
MMU and Memory Functions

```
mmu_translate va = do {
  update_state (λs. s(|tlb := tlb s - tlb_evict s|));
  (mem, asid, ttbr0, tlb) ← read_state (MEM, ASID, TTBR0, tlb);
  case lookup tlb asid (addr_val va) of
  Miss ⇒
    let entry = pt_walk asid mem ttbr0 va
    in if is_fault entry then raise PAGE_FAULT
       else do {
              update_state (λs. s(|tlb := tlb ∪ {entry}|));
              return (va_to_pa va entry)
            }
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒
      if is_fault entry then raise PAGE_FAULT
      else return (va_to_pa va entry)
}
```



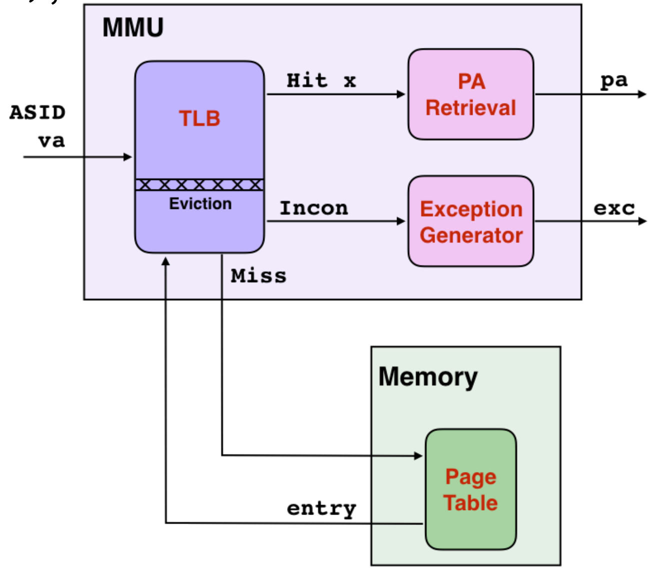
MMU and Memory Functions

```
mmu_translate va = do {
  update_state (λs. s(|tlb := tlb s - tlb_evict s|));
  (mem, asid, ttbr0, tlb) ← read_state (MEM, ASID, TTBR0, tlb);
  case lookup tlb asid (addr_val va) of
  Miss ⇒
    let entry = pt_walk asid mem ttbr0 va
    in if is_fault entry then raise PAGE_FAULT
       else do {
             update_state (λs. s(|tlb := tlb ∪ {entry}|));
             return (va_to_pa va entry)
          }
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒
      if is_fault entry then raise PAGE_FAULT
      else return (va_to_pa va entry)
}
```



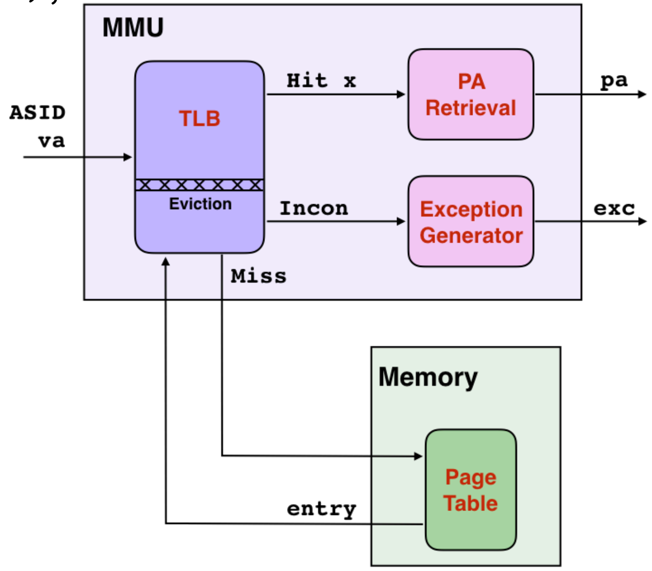
MMU and Memory Functions

```
mmu_translate va = do {
  update_state (λs. s(|tlb := tlb s - tlb_evict s|));
  (mem, asid, ttbr0, tlb) ← read_state (MEM, ASID, TTBR0, tlb);
  case lookup tlb asid (addr_val va) of
  Miss ⇒
    let entry = pt_walk asid mem ttbr0 va
    in if is_fault entry then raise PAGE_FAULT
       else do {
             update_state (λs. s(|tlb := tlb ∪ {entry}|));
             return (va_to_pa va entry)
           }
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒
      if is_fault entry then raise PAGE_FAULT
      else return (va_to_pa va entry)
}
```



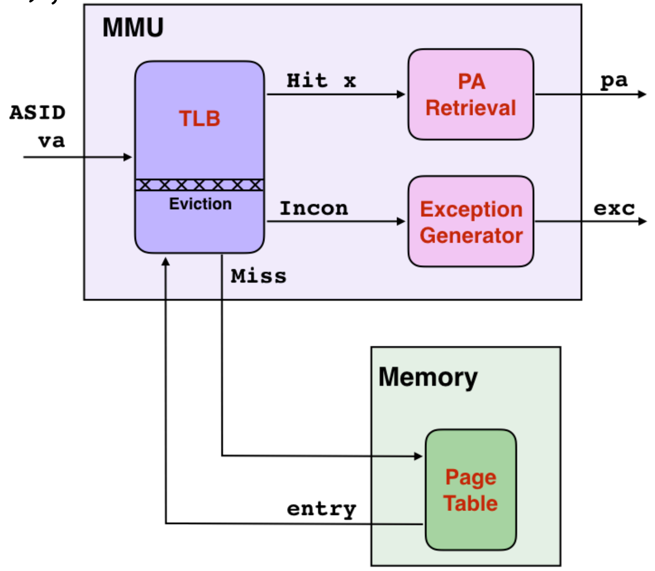
MMU and Memory Functions

```
mmu_translate va = do {
  update_state (λs. s(|tlb := tlb s - tlb_evict s|));
  (mem, asid, ttbr0, tlb) ← read_state (MEM, ASID, TTBR0, tlb);
  case lookup tlb asid (addr_val va) of
  Miss ⇒
    let entry = pt_walk asid mem ttbr0 va
    in if is_fault entry then raise PAGE_FAULT
       else do {
             update_state (λs. s(|tlb := tlb ∪ {entry}|));
             return (va_to_pa va entry)
           }
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒
      if is_fault entry then raise PAGE_FAULT
      else return (va_to_pa va entry)
}
```



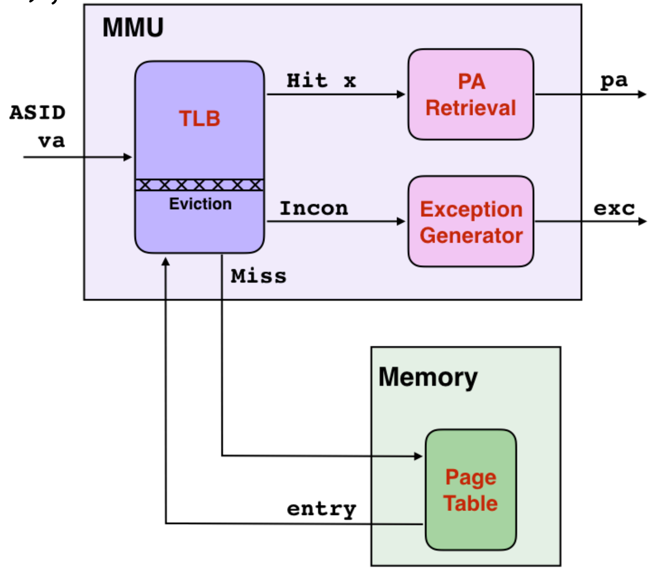
MMU and Memory Functions

```
mmu_translate va = do {
  update_state (λs. s(|tlb := tlb s - tlb_evict s|));
  (mem, asid, ttbr0, tlb) ← read_state (MEM, ASID, TTBR0, tlb);
  case lookup tlb asid (addr_val va) of
  Miss ⇒
    let entry = pt_walk asid mem ttbr0 va
    in if is_fault entry then raise PAGE_FAULT
       else do {
              update_state (λs. s(|tlb := tlb ∪ {entry}|));
              return (va_to_pa va entry)
           }
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒
       if is_fault entry then raise PAGE_FAULT
       else return (va_to_pa va entry)
}
```



MMU and Memory Functions

```
mmu_translate va = do {
  update_state (λs. s(|tlb := tlb s - tlb_evict s|));
  (mem, asid, ttbr0, tlb) ← read_state (MEM, ASID, TTBR0, tlb);
  case lookup tlb asid (addr_val va) of
  Miss ⇒
     let entry = pt_walk asid mem ttbr0 va
     in if is_fault entry then raise PAGE_FAULT
        else do {
              update_state (λs. s(|tlb := tlb ∪ {entry}|));
              return (va_to_pa va entry)
           }
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒
      if is_fault entry then raise PAGE_FAULT
      else return (va_to_pa va entry)
}
```



```
mmu_write (val, va, sz) = do {
  pa ← mmu_translate va;
  mem_write (val, pa, sz)
}
```

```
mmu_read (va, sz) = do {
  pa ← mmu_translate va;
  mem_read (pa, sz)
}
```

For Program Reasoning

For Program Reasoning

- Concrete MMU introduces

For Program Reasoning

- Concrete MMU introduces
  - unspecified entry replacement
    - can result in TLB miss and reloading

For Program Reasoning

- Concrete MMU introduces
  - unspecified entry replacement
    - can result in TLB miss and reloading
  - state change during
    - memory read
    - memory write outside the page table

For Program Reasoning

- Concrete MMU introduces
  - unspecified entry replacement
    - can result in TLB miss and reloading
  - state change during
    - memory read
    - memory write outside the page table
  - potential inconsistencies

For Program Reasoning

- Concrete MMU introduces
  - unspecified entry replacement
    - can result in TLB miss and reloading
  - state change during
    - memory read
    - memory write outside the page table
  - potential inconsistencies

- Programs

For Program Reasoning

- Concrete MMU introduces
  - unspecified entry replacement
    - can result in TLB miss and reloading
  - state change during
    - memory read
    - memory write outside the page table
  - potential inconsistencies

- Programs
  - must avoid inconsistencies

For Program Reasoning

- Concrete MMU introduces
  - unspecified entry replacement
    - can result in TLB miss and reloading
  - state change during
    - memory read
    - memory write outside the page table
  - potential inconsistencies

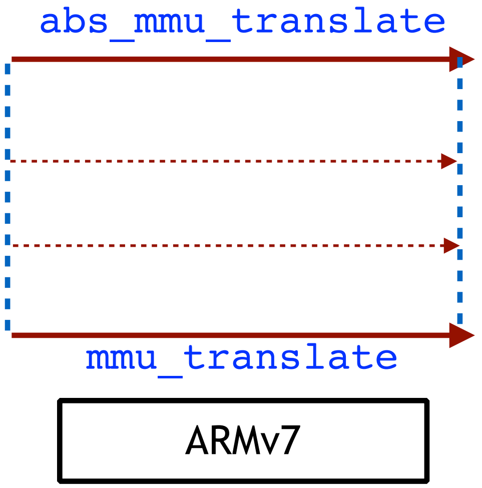
- Programs
  - must avoid inconsistencies
  - should not require reasoning about eviction and state change

Contributions

- Formal model of ARMv7-style TLB in Isabelle/HOL
  - `lookup` function

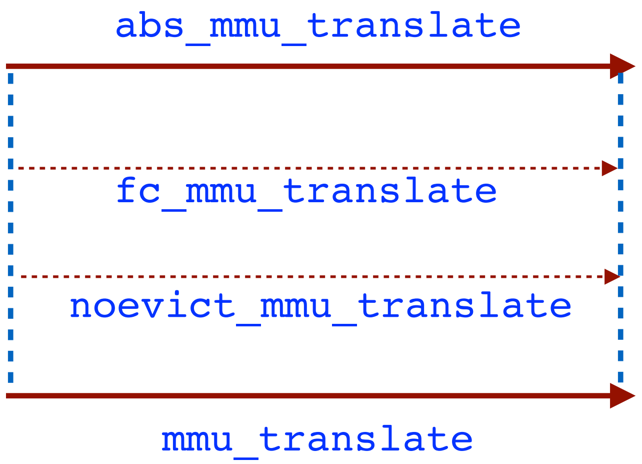
- Extension to MMU model
  - `mmu_translate, mmu_read, mmu_write`
  - integration to ARMv7 formalised ISA

- Data refinement for
  - abstracting hardware details
  - easier reasoning



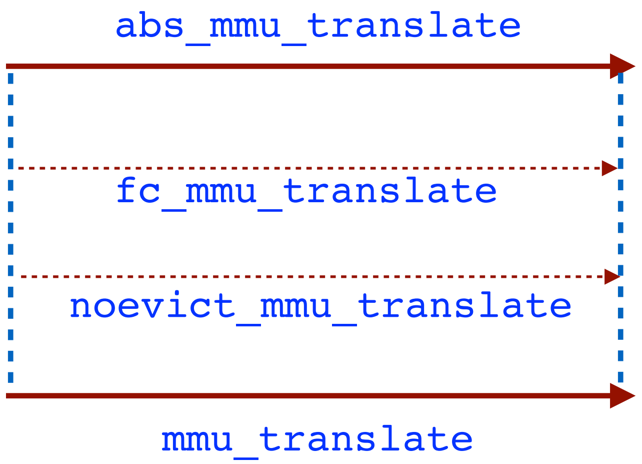
MMU Abstraction

- Stepwise data refinement for
    - abstracting eviction
    - state invariance in case of
        - memory read
        - memory write outside of page tables
    - complete abstraction for TLB



MMU Abstraction

- Stepwise data refinement for
    - abstracting eviction
    - state invariance in case of
        - memory read
        - memory write outside of page tables
    - complete abstraction for TLB



Any program that is safe with abstracted MMU
will be safe with concrete MMU

Abstracting Eviction

Abstracting Eviction

- TLB with fewer entries is always more consistent than one with more entries

```
Miss < Hit < Incon
```

$\texttt{t} \subseteq \texttt{t'} \Longrightarrow \texttt{lookup t a v} \leq \texttt{lookup t' a v}$

Abstracting Eviction

- TLB with fewer entries is always more consistent than one with more entries

```
Miss < Hit < Incon
```

t ⊆ t’ ⟹ lookup t a v ≤ lookup t’ a v

- `noevict_mmu_translate`
  - identical to `mmu_translate` except it doesn’t evict entries

Abstracting Eviction

- TLB with fewer entries is always more consistent than one with more entries

$$\texttt{Miss} < \texttt{Hit} < \texttt{Incon}$$

$$\texttt{t} \subseteq \texttt{t'} \Longrightarrow \texttt{lookup t a v} \leq \texttt{lookup t' a v}$$

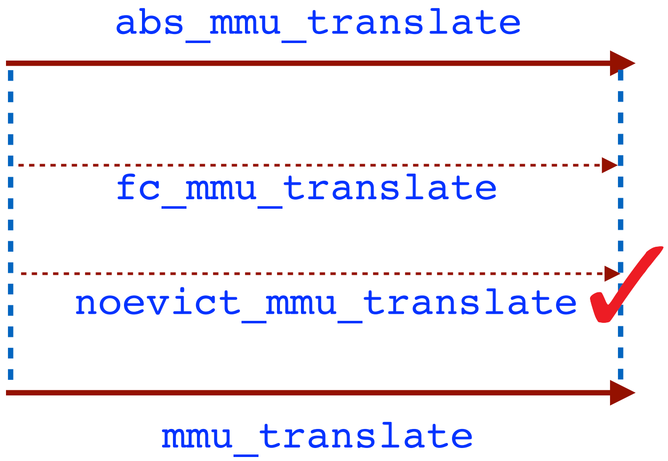
- `noevict_mmu_translate`
  - identical to `mmu_translate` except it doesn't evict entries

$$\begin{tikzcd}
\mathbf{t} \arrow[r, "\texttt{noevict\_mmu\_translate va}"] \arrow[d, dotted, no head, "\texttt{TLB } s \subseteq \texttt{TLB } t"'] & \mathbf{t'} \arrow[d, dotted, no head, "\texttt{TLB } s' \subseteq \texttt{TLB } t'"] \\
\mathbf{s} \arrow[r, "\texttt{mmu\_translate va}"'] & \mathbf{s'}
\end{tikzcd}$$

`invariant: consistent w.r.t va`

MMU Abstraction

- Stepwise data refinement for
  - abstracting eviction ✓
  - state invariance in case of
    - memory read
    - memory write outside of page tables
  - complete abstraction for TLB



Any program that is safe with abstracted TLB
will be safe with concrete TLB

State Invariance

- `fc_mmu_translate`
  - fc stands for fully-cached
  - caching page table entirely in TLB (no TLB miss)

State Invariance

- `fc_mmu_translate`
  - fc stands for fully-cached
  - caching page table entirely in TLB (no TLB miss)

$$\begin{tikzcd}
\mathtt{t} \arrow[r, "\mathtt{fc\_mmu\_translate}\ \ \mathtt{va}"] \arrow[d, dash, dotted, "\mathtt{TLB}\ \mathtt{s} \subseteq \mathtt{TLB}\ \mathtt{t}"'] & \mathtt{t'} \arrow[d, dash, dotted, "\mathtt{TLB}\ \mathtt{s'} \subseteq \mathtt{TLB}\ \mathtt{t'}"] \\
\mathtt{s} \arrow[r, "\mathtt{noevict\_mmu\_translate}\ \ \mathtt{va}"'] & \mathtt{s'}
\end{tikzcd}$$

```
fully_cached_pt t
invariant: consistent w.r.t va
```

State Invariance - Memory Access

State Invariance - Memory Access

- Memory read

$$\begin{tikzcd}
\mathbf{s} \arrow[r, "\texttt{fc\_mmu\_read}\ \ \texttt{va}"'] & \mathbf{t} \\
\texttt{fully\_cached\_pt s} & \texttt{TLB t = TLB s}
\end{tikzcd}$$

State Invariance - Memory Access

- Memory read

$$\begin{tikzcd} \mathbf{s} \arrow[r, "\texttt{fc\_mmu\_read}\ \ \texttt{va}"'] & \mathbf{t} \end{tikzcd}$$

`fully_cached_pt s`   `TLB t = TLB s`

- Memory write outside of the page table

$$\begin{tikzcd} \mathbf{s} \arrow[r, "\texttt{fc\_mmu\_write}\ \ \texttt{va}"'] & \mathbf{t} \end{tikzcd}$$

`pt_walk va s = pt_walk va t`   `TLB t = TLB s`

State Invariance - Memory Access

- Memory read

$$\begin{tikzcd} s \arrow[r, "\texttt{fc\_mmu\_read}\ \ \texttt{va}"'] & t \end{tikzcd}$$

`fully_cached_pt s`   `TLB t = TLB s`

- Memory write outside of the page table

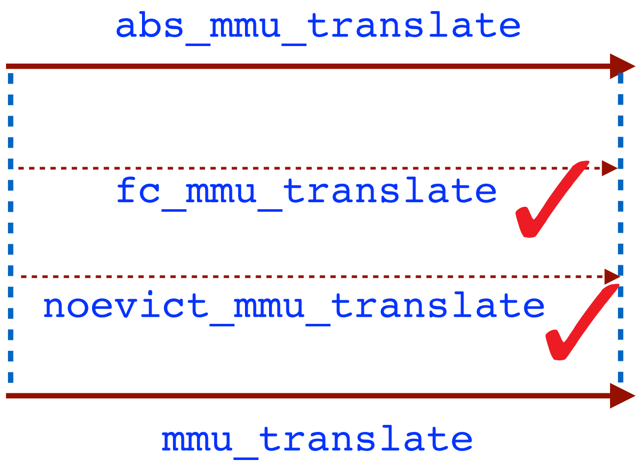
$$\begin{tikzcd} s \arrow[r, "\texttt{fc\_mmu\_write}\ \ \texttt{va}"'] & t \end{tikzcd}$$

`pt_walk va s = pt_walk va t`   `TLB t = TLB s`

  - user-level programs
  - seL4 static address mappings

MMU Abstraction

- Stepwise data refinement for
  - abstracting eviction ✓
  - state invariance in case of
    - memory read ✓
    - memory write outside of page tables
  - complete abstraction for TLB



Any program that is safe with abstracted TLB
will be safe with concrete TLB

Completely Abstracted TLB

- No TLB lookup is required
  - extend state with (`ASID x 32 bit`) instead of `tlb`

Completely Abstracted TLB

- No TLB lookup is required
  - extend state with `(ASID x 32 bit)` instead of `tlb`

```
mmu_translate_set va = do {
  (mem, asid, ttbr0, incon_set) ← read_state (MEM, ASID, TTBR0, incon_set);
  if (asid, addr_val va) ∈ incon_set then raise IMPLEMENTATION_DEFINED
  else let entry = pt_walk asid mem ttbr0 va
       in if is_fault entry then raise PAGE_FAULT else return (va_to_pa va entry)
}
```

Completely Abstracted TLB

- No TLB lookup is required
  - extend state with `(ASID x 32 bit)` instead of `tlb`

```
mmu_translate_set va = do {
  (mem, asid, ttbr0, incon_set) ← read_state (MEM, ASID, TTBR0, incon_set);
  if (asid, addr_val va) ∈ incon_set then raise IMPLEMENTATION_DEFINED
  else let entry = pt_walk asid mem ttbr0 va
      in if is_fault entry then raise PAGE_FAULT else return (va_to_pa va entry)
}
```

Completely Abstracted TLB

- No TLB lookup is required
  - extend state with (ASID x 32 bit) instead of tlb

```
mmu_translate_set va = do {
  (mem, asid, ttbr0, incon_set) ← read_state (MEM, ASID, TTBR0, incon_set);
  if (asid, addr_val va) ∈ incon_set then raise IMPLEMENTATION_DEFINED
  else let entry = pt_walk asid mem ttbr0 va
      in if is_fault entry then raise PAGE_FAULT else return (va_to_pa va entry)
}
```

Completely Abstracted TLB

- No TLB lookup is required
  - extend state with (ASID x 32 bit) instead of tlb

```
mmu_translate_set va = do {
  (mem, asid, ttbr0, incon_set) ← read_state (MEM, ASID, TTBR0, incon_set);
  if (asid, addr_val va) ∈ incon_set then raise IMPLEMENTATION_DEFINED
  else let entry = pt_walk asid mem ttbr0 va
      in if is_fault entry then raise PAGE_FAULT else return (va_to_pa va entry)
}
```

Completely Abstracted TLB

- No TLB lookup is required
  - extend state with (`ASID x 32 bit`) instead of `tlb`

```
mmu_translate_set va = do {
  (mem, asid, ttbr0, incon_set) ← read_state (MEM, ASID, TTBR0, incon_set);
  if (asid, addr_val va) ∈ incon_set then raise IMPLEMENTATION_DEFINED
  else let entry = pt_walk asid mem ttbr0 va
      in if is_fault entry then raise PAGE_FAULT else return (va_to_pa va entry)
}
```

Completely Abstracted TLB

- No TLB lookup is required
    - extend state with (`ASID x 32 bit`) instead of `tlb`

```
mmu_translate_set va = do {
  (mem, asid, ttbr0, incon_set) ← read_state (MEM, ASID, TTBR0, incon_set);
  if (asid, addr_val va) ∈ incon_set then raise IMPLEMENTATION_DEFINED
  else let entry = pt_walk asid mem ttbr0 va
      in if is_fault entry then raise PAGE_FAULT else return (va_to_pa va entry)
}
```

$$\begin{tikzcd}
\mathbf{t} \arrow[r, "\texttt{va abs\_mmu\_translate pa}"] & \mathbf{t'} \\
\mathbf{s} \arrow[u, dotted, dash, "\texttt{incon\_set } \mathbf{s} \subseteq \texttt{ incon\_set } \mathbf{t}"] \arrow[r, "\texttt{va fc\_mmu\_translate pa}"'] & \mathbf{s'} \arrow[u, dotted, dash, "\texttt{incon\_set } \mathbf{s'} \subseteq \texttt{ incon\_set } \mathbf{t'}"']
\end{tikzcd}$$

invariant: va ∉ incon_set t

Completely Abstracted TLB

- No TLB lookup is required
  - extend state with `(ASID x 32 bit)` instead of `tlb`

```
mmu_translate_set va = do {
  (mem, asid, ttbr0, incon_set) ← read_state (MEM, ASID, TTBR0, incon_set);
  if (asid, addr_val va) ∈ incon_set then raise IMPLEMENTATION_DEFINED
  else let entry = pt_walk asid mem ttbr0 va
      in if is_fault entry then raise PAGE_FAULT else return (va_to_pa va entry)
}
```

$$\begin{tikzcd}
\mathbf{t} \arrow[r, "\mathtt{va\ abs\_mmu\_translate\ pa}"] \arrow[d, dash, dotted, "\mathtt{incon\_set}\ \mathbf{s} \subseteq \mathtt{incon\_set}\ \mathbf{t}"'] & \mathbf{t'} \arrow[d, dash, dotted, "\mathtt{incon\_set}\ \mathbf{s'} \subseteq \mathtt{incon\_set}\ \mathbf{t'}"] \\
\mathbf{s} \arrow[r, "\mathtt{va\ fc\_mmu\_translate\ pa}"'] & \mathbf{s'}
\end{tikzcd}$$

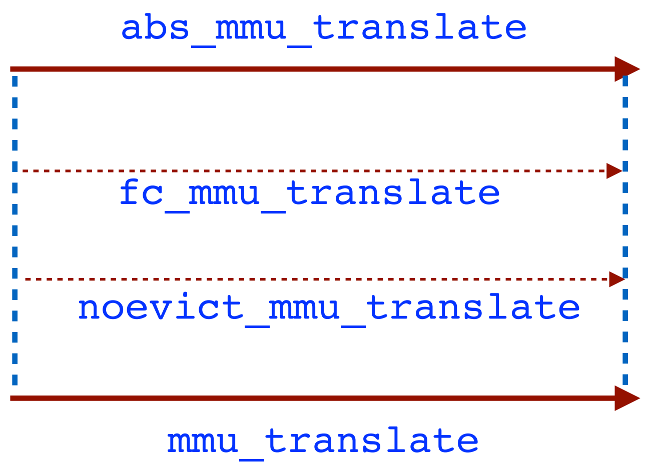
No state change at all, apart from potentially raising exceptions

`invariant: va ∉ incon_set t`

Taken Together

- TLB should be transparent to programs if maintained correctly

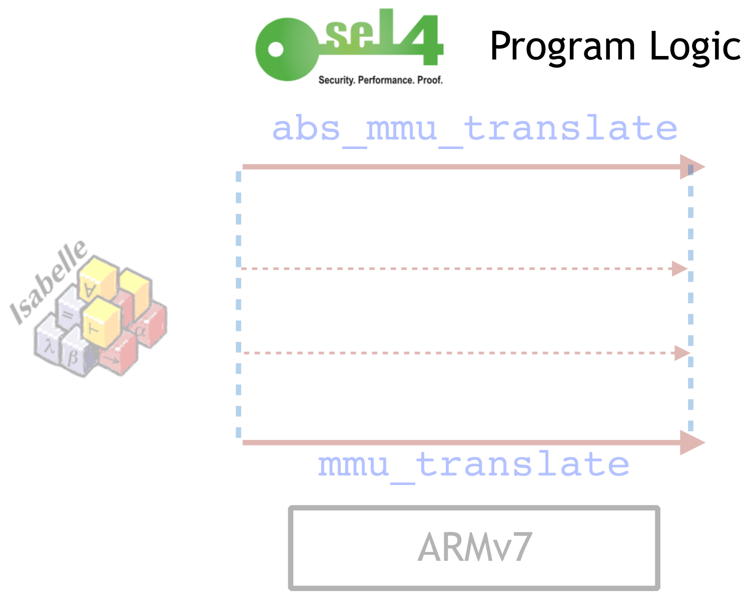
- Refinement chain



- Cached page table walks in ARMv7-A

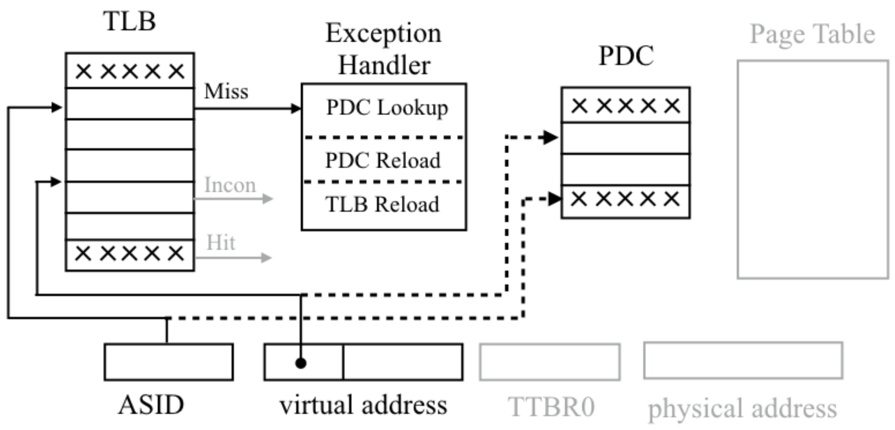
- Abstraction
  - hides low-level hardware TLB details
  - easy to reason about
  - reduction theorems

Future Direction



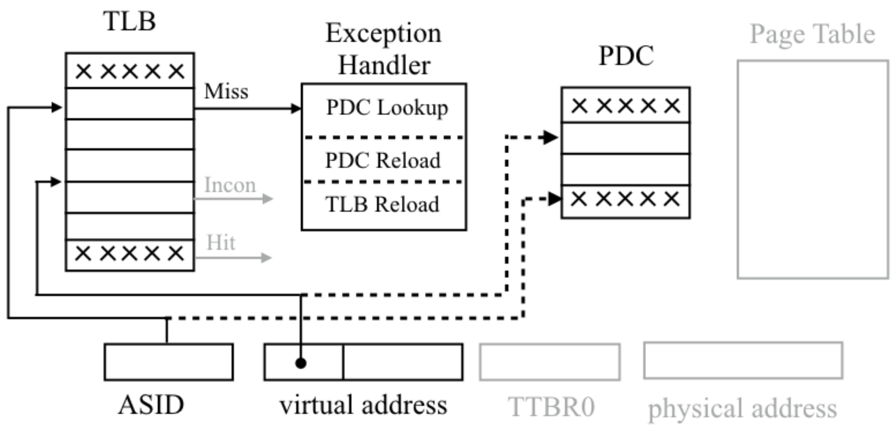
Thank You

Caching Partial Walks



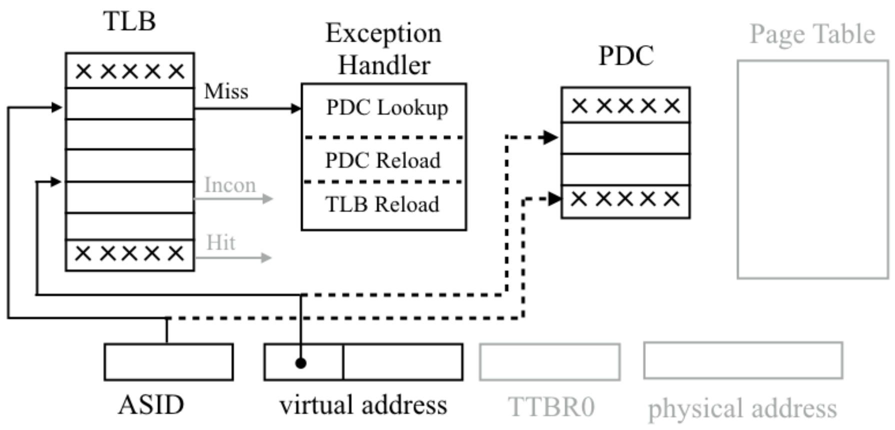
Caching Partial Walks

- Hardware caching of partial page table walks
  - ARMv7-A



Caching Partial Walks

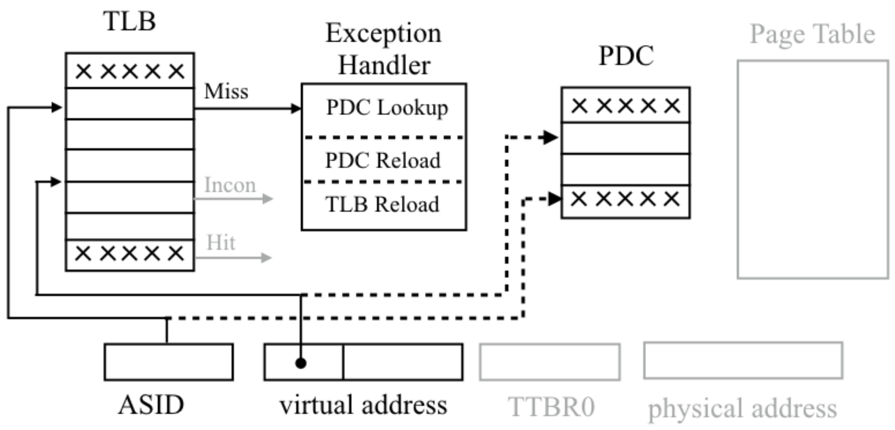
- Hardware caching of partial page table walks
  - ARMv7-A



Caching Partial Walks

- Hardware caching of partial page table walks
  - ARMv7-A

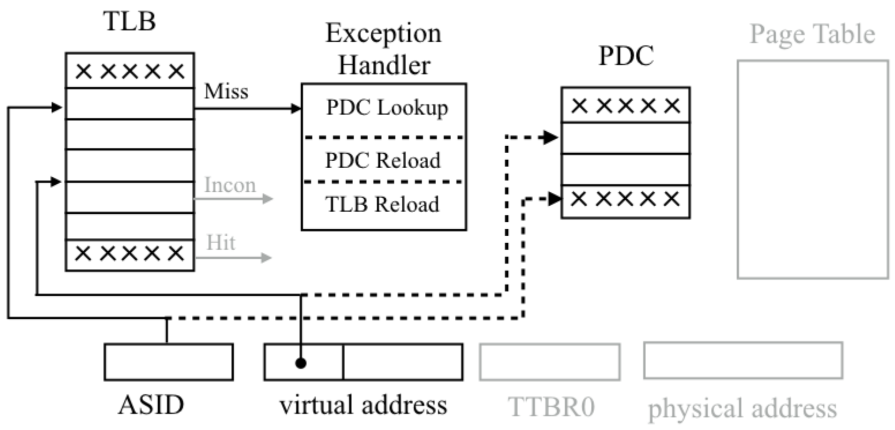
- State extension
  - TLB and PDC



Caching Partial Walks

- Hardware caching of partial page table walks
  - ARMv7-A

- State extension
  - TLB and PDC

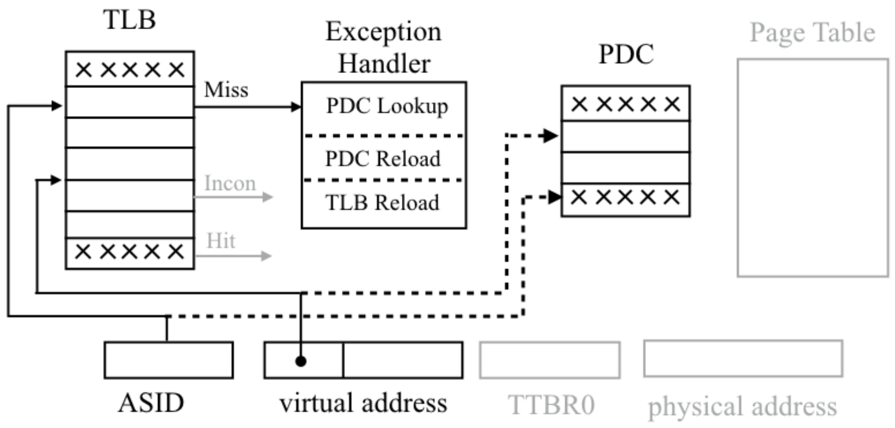


Caching Partial Walks

- Hardware caching of partial page table walks
  - ARMv7-A

- State extension
  - TLB and PDC

- MMU functions
  - concrete, saturated

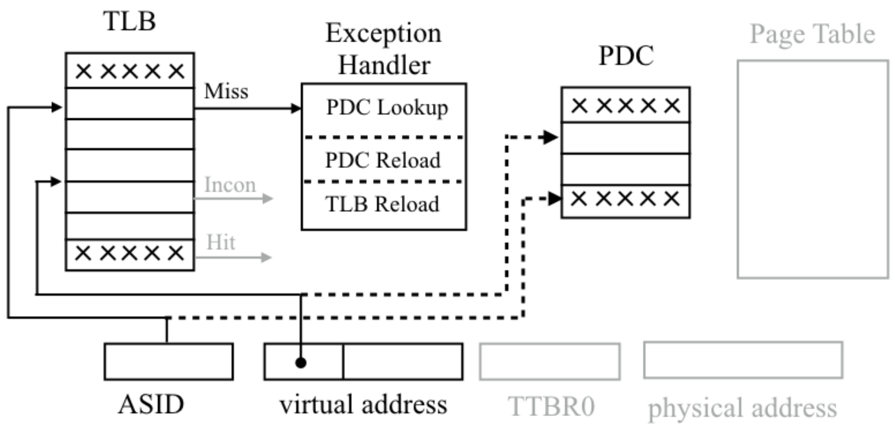


Caching Partial Walks

- Hardware caching of partial page table walks
  - ARMv7-A

- State extension
  - TLB and PDC

- MMU functions
  - concrete, saturated

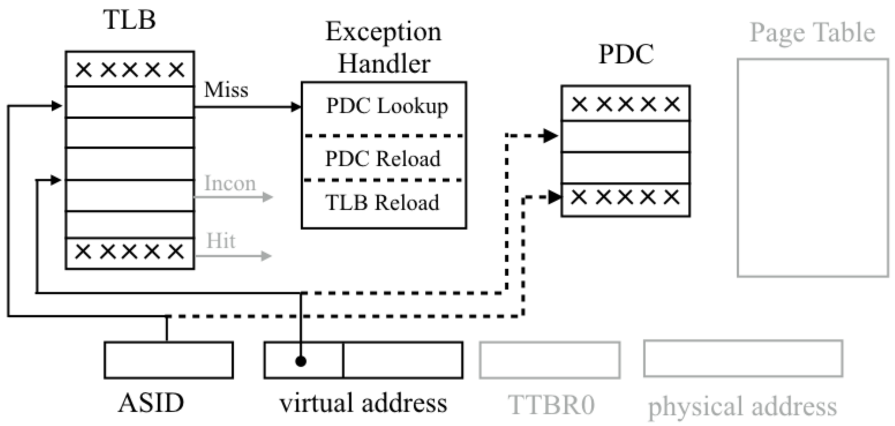


Caching Partial Walks

- Hardware caching of partial page table walks
  - ARMv7-A

- State extension
  - TLB and PDC

- MMU functions
  - concrete, saturated



- Saturation
  - Cache hierarchy
  - Step-wise refinement to fully abstract PDC and TLB