Advanced Operating Systems
(263-3800-00L)

Timothy Roscoe
Fall 2014

https://www.systems.ethz.ch/courses/fall2014/aos
Milestone 1:

• Build a page table
  – The ARMv7 virtual memory system
• Turn on the MMU
  – ARM “coprocessors” and CP15
  – Caches: What every OS designer needs to know
• Create a prototype process (“init”)  
  – Register calling conventions
• Exit the kernel to user space
• Create a system call to print a character
But first:

Hardware access from C

• You may have written code like this:

```c
#define GPIO_BASE 0x4A310000

static uint32_t *gpio_oe = \(uint32_t *)\(GPIO_BASE + 0x0134\);
static uint32_t *gpio_dataout = \(uint32_t *)\(GPIO_BASE + 0x013C\);
...

// Enable LED output
*gpio_oe = new_reg_value;
```

• Why doesn’t it work?
Hint...

// GPIO address for the LED line
#define GPIO_BASE 0x4A310000

static volatile uint32_t *gpio_oe = 
 (uint32_t *)(GPIO_BASE + 0x0134);
static volatile uint32_t *gpio_dataout = 
 (uint32_t *)(GPIO_BASE + 0x013C);
...

// Enable LED output
*gpio_oe = new_reg_value;

- volatile: value may change unexpectedly
  ⇒ Compiler will not optimize away loads
Paged Memory Management
Page-based virtual memory
(review)

- **Page table** stores the translation
  - TLB is a cache for the page table(s)
  - May be loaded by hardware or software
- **Base page size** determined by hardware
  - 4kB, 8kB typical
- **Superpage** is multiple \(2^n\) of base page size
  - Used to increase TLB coverage, reduce contention
Linear page table

- Array of frame numbers indexed by page number
  - Example: PDP-11 (16-bit address, 8kB pages)
- Not feasible for a 32-bit address space...
Multi-level or hierarchical page tables

- Given:
  - 4KB ($2^{12}$) page size
  - 48-bit address space
  - 4-byte PTE

- Problem:
  - Would need a 256 GB linear page table!
  - $2^{48} \times 2^{-12} \times 2^2 = 2^{38}$ bytes
Multi-level or hierarchical page tables

- Example: 2-level page table
  - Level 1 table: each PTE points to a page table
  - Level 2 table: each PTE points to a page (paged in and out like other data)

- Level 1 table stays in memory
- Level 2 tables paged in and out
x86-64 Paging

Virtual address

VPN1  VPN2  VPN3  VPN4  VPO

Page Map Table  Page Directory Pointer Table  Page Directory Table  Page Table

PM4LE  PDPE  PDE  PTE

BR

40  12

PPN  PPO

Physical address
Features

• Saves memory for mostly-empty address spaces
  – But more memory references required for lookup
• Natural support for superpages
  – “Early termination” of table walk
• Frequently implemented in hardware (or microcode…)
• Also the generic page table “abstraction” in Linux
Problems with hierarchical page tables

• Depth scales with size of virtual address space
  – 5–6 levels deep for a full 64-bit address space
  – AMD64 (48-bit virtual address) needs 4 levels

• A sparse address-space layout requires lots of memory for (mostly empty) tables
  – Not a big problem for the traditional UNIX memory model
ARMv7 page tables

• MMU supports:
  – Supersections (16MB)
  – Sections (1MB)
  – Large pages 64kB
  – Small pages 4kB

• Two-level hierarchical page table:
  – First-level table (16Kb in size) holds section translations and pointers to second-level tables
  – Second-level tables hold page translations
# 1st level page descriptors

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ignored</td>
<td>0 0</td>
</tr>
<tr>
<td>Page table base address</td>
<td>IMP</td>
</tr>
<tr>
<td></td>
<td>domain</td>
</tr>
<tr>
<td></td>
<td>sbz NS sbz</td>
</tr>
<tr>
<td>Section base address</td>
<td>NS nG S AP TEX AP IMP</td>
</tr>
<tr>
<td></td>
<td>domain XN CB</td>
</tr>
<tr>
<td>Supersection base address</td>
<td>NS 1 nG S AP TEX AP IMP</td>
</tr>
<tr>
<td></td>
<td>Extended base address XN CB</td>
</tr>
<tr>
<td>reserved</td>
<td>1 1</td>
</tr>
</tbody>
</table>
### 2\textsuperscript{nd} level page descriptors

<table>
<thead>
<tr>
<th>Large page address</th>
<th>nG</th>
<th>S</th>
<th>AP</th>
<th>SBZ</th>
<th>AP</th>
<th>C</th>
<th>B</th>
<th>XN</th>
</tr>
</thead>
<tbody>
<tr>
<td>ignored</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Small page address</th>
<th>nG</th>
<th>S</th>
<th>AP</th>
<th>TEX</th>
<th>AP</th>
<th>C</th>
<th>B</th>
<th>XN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>XN</td>
</tr>
</tbody>
</table>

**Note:** no hardware-managed dirty or accessed bits!
Small page translation

Translation table base register

Translation base

SBZ

Virtual address

L1 table index

L2 table index

Page index

Translation base

Page index

Access control 0 1

Access control 0 1

Small page base address

Page index

Note: addresses in physical memory!
Page table for small pages

Base address

4k entries 16kB

4kB page

2nd level page table

2nd level page table

Translates VA[0:11] = offset in page

VA[12:19] translates

Translates VA[20:31]

1st level page table

...01

...01

...1x

...1x

256 entries 1kB

...
Large page translation

Translation table base register

Virtual address

Address of 1st level descriptor

1st level descriptor

Address of 2nd level descriptor

2nd level descriptor

Physical address

Translation base

SBZ

? L1 table index L2 index Page index

Translation base L1 table index 0 0

Page table base address Access control 0 1

Page table base address L2 table index 0 0

Large page base address Access control 0 1

Large page base address Page index
Page table for large pages

1st level page table

Translates VA[20:31]

2nd level page table

Translates VA[12:19], but effectively VA[16:19]

64kB page

256 entries 1kB

Base address

4k entries 16kB

16 identical entries

VA[0:15] = offset in page
Section translation
Page table for sections

Base address

4k entries 16kB

256 entries 1kB

1st level page table

Translates VA[20:31]

1MB section

VA[0:19] = offset in section
Supersection translation

- Translation table base register
- Virtual address
- Address of 1st level descriptor
- 1st level descriptor
- Physical address

Diagram:

Translation base → Supersection index

Translation base → SBZ

Translation base → L1 table index 00

Translation base → Supersection base addr

Translation base → Access control + extension bits 10

Translation base → Section index
Where is the page table?

• Two page table base registers:
  – TTBR0: low addresses (MSB’s == 0)
  – TTBR1: everything else

• Which one do we use?
  – TTBRC.N indicates how big the TTBR0 table is
  – $N = x \implies$ TTBR0 is used if top $x$ bits of VA are zero
  – $N = 0 \implies$ TTBR0 is used for everything
Split virtual address space

\[ \text{TTCR.N} = x \]
\[ x \neq 0 \]

\[ \implies \]

Translated using TTBR1

\[ 0xFFFFFFFF \]

Translated using TTBR0

\[ 0b0..01... \]

Virtual address space

\[ 0x00000000 \]

\[ x \text{ leading zeroes} \]
Example Barrelfish virtual address space

Virtual address space

Translated using TTBR1
Kernel mappings

TTCR.N = 1

0xFFFFFFFF

0x8000000

Or just set TTCR.N = 0 and do it all with one table...

Translated using TTBR0
User process

0x00000000
ARM Coprocessors and CP15
Privileged processor state

• All processors need a way to access “privileged” state
  – MMU configuration
  – System information (caches, etc.)
  – Fault information
  – Debugging hardware support
Options

• Extra processor instructions
  – MIPS TLB instructions
  – Alpha PALmode

• Extra registers
  – ia32 CRx, MSRs (Model-specific registers)

• “Coprocessors”
  – ARM CPx, in particular CP15
Enabling the MMU

cp15_enable_mmu:
    ldr r0, =0x55555555
    mcr p15, 0, r0, c3, c0, 0  // Set ASID to 0
    mov r0, #0
    mcr p15, 0, r0, c13, c0, 1 // Set the Domain Access register
    mov r0, #1
    mcr p15, 0, r0, c3, c0, 0  // Enable: D-Cache, I-Cache, Alignment, MMU
    ldr r1, =0x1007
    mrc p15, 0, r0, c1, c0, 0  // read sys. conf. reg.
    orr r0, r0, r1
    mcr p15, 0, r0, c1, c0, 0   // enable MMU
    // Clear pipeline
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    // Wait on some CP15 register
    mrc p15, 0, r0, c2, c0, 0
    mov r0, r0
    sub pc, pc, #4
    bx lr
Enabling the MMU

```assembly
 cp15_enable_mmu:
    ldr r0, =0x55555555
    mcr p15, 0, r0, c3, c0, 0
    // Set ASID to 0
    mov r0, #0
    mcr p15, 0, r0, c13, c0, 1
    // Set the Domain Access register
    mov r0, #1
    mcr p15, 0, r0, c3, c0, 0
    // Enable: D-Cache, I-Cache, Alignment, MMU
    ldr r1, =0x1007
    mrc p15, 0, r0, c1, c0, 0 // read sys. conf. reg.
    orr r0, r0, r1
    mcr p15, 0, r0, c1, c0, 0 // enable MMU
    // Clear pipeline
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    // Wait on some CP15 register
    mrc    p15, 0, r0, c2, c0, 0
    mov    r0, r0
    sub    pc, pc, #4
    bx      lr
```

Moving values to and from “coprocessors” (in this case number 15)

We’ll provide code to do this, but you should take a look at it (and Chapter 4 of the Cortex A9 TRM)
MCR and MRC

• Moving values to and from coprocessors:

MRC \langle cp\rangle, \langle op\rangle, \langle ARM\ reg\rangle, \langle cp\ reg\rangle, \langle cp\ reg2\rangle, \langle op2\rangle
MCR \langle cp\rangle, \langle op\rangle, \langle ARM\ reg\rangle, \langle cp\ reg\rangle, \langle cp\ reg2\rangle, \langle op2\rangle

Huge flexibility: can do almost anything
Coprocessor 15: System Control

• Overall system control and configuration
• MMU configuration and management
• Cache configuration and management
• System performance monitoring.

Other coprocessors for ARM implement DSP, FP, Java acceleration, Security, etc.
Caches: What every OS designer needs to know
Caching (review)

• Cache:
  something that remembers previous results
  ⇒ can sometimes give the answer faster
  E.g. memory caches, TLBs, etc.

• Work by locality:
  – Temporal locality:
    if I needed x recently, I’m likely to need it again soon.
  – Spatial locality:
    if I needed x, I’m likely to need something close by.
Memory caching (review)

- Fast memory between registers & slow RAM
  - 1–5 vs. 10–100 cycles
- Holds recently used data and/or instructions
- Compensates for slow RAM if hit rate high (~90%)
- Hardware: (mostly) transparent to software
- Size: few kB – several MB
- Typically a hierarchy (2 – 5 levels)
Cache issues

• Caches are hardware, transparent to software
• So, why worry about them in the OS?
• Well...
  – Performance
  – Synonyms
  – Homonyms
• And pretty much essential for multiprocessors
  – Can't really scale without them
  – A later lecture will cover MP/OS cache issues
Cache performance really matters.

<table>
<thead>
<tr>
<th></th>
<th>Cycles</th>
<th>Normalised</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>L2 cache</td>
<td>15</td>
<td>7.5</td>
</tr>
<tr>
<td>L3 cache</td>
<td>75</td>
<td>37.5</td>
</tr>
<tr>
<td>Other L1/L2</td>
<td>130</td>
<td>65</td>
</tr>
<tr>
<td>Memory</td>
<td>~300</td>
<td>~150</td>
</tr>
<tr>
<td>1-hop remote L3</td>
<td>190</td>
<td>95</td>
</tr>
<tr>
<td>2-hop remote L3</td>
<td>260</td>
<td>130</td>
</tr>
</tbody>
</table>

(AMD “Barcelona” @ 2GHz)
DMA consistency problem

- DMA (normally) uses physical addresses and bypasses cache
  - CPU access inconsistent with device access
  - Need to flush cache before device write
  - Need to invalidate cache before device read
Cache organization

• Transfer units
  – registers ↔ L1 cache ≤ 1 word (1 – 16 bytes)
  – cache ↔ RAM (or cache) 16 – 32 bytes, or more
  – *Cache line*: also unit of storage allocation in cache

• Cache line associated information:
  – Valid bit
  – Modified bit
  – Tag

• Improves memory access by:
  – Absorbs reads (increases b/w, reduces latency)
  – Make writes asynchronous (hides latency)
  – Clusters reads & writes (hides latency)
Cache access

- **Virtually indexed:**
  - Lookup by virtual address
  - Concurrent with address translation

- **Physically indexed:**
  - Lookup by physical address
Cache indexing

Tag: distinguishes lines of a set ...address bits not used for indexing
Cache addressing

- Address hashed $\Rightarrow$ index of *line set*
- Associative lookup within set using *tag*
- $n$ lines/set: *$n$-way set-associative cache*
  - $n=1$: direct mapped
  - $n=\infty$: fully associative
    - *Usually $n=1 – 4$, occasionally 32 – 64*
- Hashing must be simple $\Rightarrow$
  - Use least significant bits of address
Cache mapping

• Different memory locations map to same cache line
  
  Locations mapping to set $i$ are said to be of colour $i$
  $n$-way assoc. caches hold $n$ lines of the same colour

• Cache miss types:
  - **Compulsory** miss: data cannot be in cache
  - **Capacity** miss: all cache entries in use by other data
  - **Conflict** miss: set mapped to address is full
    - Can’t happen in fully-associative cache
  - **Coherence** miss: forced by hardware coherence protocol
Cache write policy

• **Write back**: store only updates cache
  – Memory updated when dirty line replaced (flushed)
    • Clusters writes
    • Memory inconsistent with cache
    • Hard for multiprocessors

• **Write through**: store updates cache & memory
  • Memory always consistent with cache
  • Increased memory/bus traffic

• What if store to line not currently in cache?
  – Write allocate: allocate new cache line & store
  – No allocate: store to memory & bypass cache

• Typical combinations:
  – Write-back & write-allocate
  – Write-through & no-allocate
Cache addressing schemes

• So far, assumed cache only sees a virtual or physical address

• But indexing and tagging can use different addresses
  – Virtually-indexed, virtually tagged (VV)
  – Virtually-indexed, physically-tagged (VP)
  – Physically-indexed, virtually tagged (PV)
  – Physically-indexed, physically-tagged (PP)
Virtually indexed, virtually tagged

- Also called:
  - Virtually addressed
- Also (misleadingly!)
  - Virtual cache
  - Virtual address cache
- Only uses virtual addresses
  - Operates concurrently with MMU
Virtually-indexed cache issues

- **Synonyms** (aliases): multiple names for same data
- Several VA map to the same PA
  - Frames shared between processes
  - Multiple mappings of frame within AS
- May access stale data:
  - Same data cached in several lines
  - On write, one synonym updated
  - Read on other synonym → old val!
- Are synonyms a problem?
  - Depends on page and cache size
  - No problem for r/o data
  - Or i-caches
Example: MIPS R4x00 synonyms

- ASID-tagged, on-chip L1 VP cache
  - 16kB cache, 32B lines, 2-way set associative
  - 4kB (base) page size
  - Set size = 16kB/2 = 8kB > page size
  - Overlap of tag & index bits, but from different addresses!

- Remember, location of data in cache determined by index
  - Tag only confirms whether it’s a hit
  - Synonym problem iff VA_{12} \neq VA’_{12}
  - Similar issues on other processors, e.g. ARM11 (set size 16kB, page size 4kB)
Avoiding synonym problems

• Hardware synonym detection
• Flush cache on context switch
  – Doesn’t help for aliasing within address space
• Detect synonyms and ensure
  – All read only, OR
  – Only one synonym mapped at a time
• Restrict VM mapping so synonyms map to same cache set
  – E.g. on R4x00, ensure that $VA_{12} = PA_{12}$
Virtually-indexed cache issues

- **Homonyms**: same names for different data
- VA used for indexing is context dependent
  - Same VA refers to different PAs
  - Tag does not uniquely identify data
  - Wrong data is accessed!
  - OS must prevent this!
- Homonym prevention:
  - Flush cache on context switch
  - Force non-overlapping address-space layout
  - Tag VA with **address-space ID** (ASID)
  - Use physical tags
Address mismatch problems: Aliasing

- Page aliased in different address spaces
  - AS1: $VA_{12} = 1$, AS2: $VA_{12} = 0$
- One alias gets modified
  - In a write-back cache, other alias sees stale data
  - Lost-update problem
Address mismatch problems: Remapping

- Unmap page with dirty cache line
- Re-use (remap) frame to a different page (in same or different AS)
- Write to a new page
  - Without mismatch, new write overwrites old (hits same cache line)
  - With mismatch, order can be reversed: “cache bomb”
Summary: VV caches

• Fastest: don’t rely on TLB for retrieving data
  – Still need TLB lookup for protection
  – Or other mechanism to provide protection

• Suffer from synonyms and homonyms
  – Requires flush on context switch
    • Makes context switches expensive
    • May even be required on kernel ⇒ user switch
  – ... or guarantee of no synonyms or homonyms

• Require TLB lookup for writeback!

• Used on i860, ARMv5/StrongARM/XScale

• Used for i-cache on many architectures
  – Alpha, Pentium 4, etc.
VV caches with keys

• Add *address space identifier* (ASID) part of tag
  – On access compare with CPU’s ASID register

• Removes homonyms, creates synonyms
  – Perhaps faster context switch
  – ASID recycling still requires a cache flush

• Doesn’t solve:
  – Synonym problem
  – Write-back problem
Virtually indexed, physically tagged

- Virtual addr \(\Rightarrow\) line
- Physical addr \(\Rightarrow\) tag
- Address translation required to retrieve data
- Index concurrently with MMU
- Use MMU output for check
VP caches

• Medium speed
  – Lookup in parallel with address space translation
  – Tag comparison after address translation
• Bigger tags (can’t leave off set-number bits)
  – Increases area, latency, power consumption
• No homonym problem
• Potential synonym problem
Physically indexed, physically tagged

- Only uses physical addresses
- Translation must complete before cache access can start
  - Or must it?
- Note: page offset invariant under virtual address translation
  - Index bits $\subseteq$ offset
  - Cache accessed without translation
PP caches

• Advantages:
  – No synonyms
  – No homonyms
  – Easy to manage
  – Cache might use bus snooping for DMA data

• Need address translation before lookup?
  – Well...
Fast PP cache

Translation and lookup in parallel

Diagram:
- CPU
  - page
  - index
  - page offset
- TLB
  - tag
  - index
  - byte

Table:

<table>
<thead>
<tr>
<th>VD</th>
<th>tag</th>
<th>word 0</th>
<th>word 1</th>
<th>word 2</th>
<th>word 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>VD</td>
<td>tag</td>
<td>word 0</td>
<td>word 1</td>
<td>word 2</td>
<td>word 3</td>
</tr>
<tr>
<td>VD</td>
<td>tag</td>
<td>word 0</td>
<td>word 1</td>
<td>word 2</td>
<td>word 3</td>
</tr>
<tr>
<td>VD</td>
<td>tag</td>
<td>word 0</td>
<td>word 1</td>
<td>word 2</td>
<td>word 3</td>
</tr>
<tr>
<td>VD</td>
<td>tag</td>
<td>word 0</td>
<td>word 1</td>
<td>word 2</td>
<td>word 3</td>
</tr>
</tbody>
</table>
Write buffers

- Store operations take long to complete
  - E.g. if cache line must be read or allocated
- Avoid stalling CPU by buffering writes
- **Write buffer** is FIFO queue of incomplete stores
  - also called store buffer or write-behind buffer
- Can also read intermediate values out of buffer
  - Service load of a value still in the write buffer
  - Avoids unnecessary stalls of load operations
- Implies that memory contents are temporarily stale
  - May need to wait until a stop hits memory
  - On a multiprocessor, CPUs see different order of writes
  - “weak store order”, to be revisited in SMP context!
The good news: Cortex A9 caches

- Both caches:
  - 32 byte line size (8 words)
  - 4-way set-associative
  - Pseudo round-robin or pseudo random replacement
  - Critical word first filling
- I-cache:
  - Virtually indexed
  - Physically tagged
- D-cache
  - Physically indexed
  - Physically tagged
  - 4 outstanding read plus 4 outstanding write misses
- Write buffer:
  - 4 64-bit slots, with merging
Summary

• The OS has to *manage caches* if it is to provide:
  – Correctness
  – Performance

• Interactions between caches and memory translation are complex and subtle

• OSes typically try to hide these from the user
ARM calling conventions
Calling conventions

• How processor registers are used by software
  – How the compiler uses registers
  – How arguments and results are passed
  – Procedure linkage
  – Etc.

• You’ll need to set up the initial register values for your new process.
ARM register usage

• ARM register names have aliases

<table>
<thead>
<tr>
<th>r0</th>
<th>a1</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>a2</td>
</tr>
<tr>
<td>r2</td>
<td>a3</td>
</tr>
<tr>
<td>r3</td>
<td>a4</td>
</tr>
<tr>
<td>r4</td>
<td>v1</td>
</tr>
<tr>
<td>r5</td>
<td>v2</td>
</tr>
<tr>
<td>r6</td>
<td>v3</td>
</tr>
<tr>
<td>r7</td>
<td>v4</td>
</tr>
<tr>
<td>r8</td>
<td>v5</td>
</tr>
<tr>
<td>r9</td>
<td>v6</td>
</tr>
<tr>
<td>r10</td>
<td>v7</td>
</tr>
<tr>
<td>r11</td>
<td>v8</td>
</tr>
<tr>
<td>r12</td>
<td>IP</td>
</tr>
<tr>
<td>r13</td>
<td>SP</td>
</tr>
<tr>
<td>r14</td>
<td>LR</td>
</tr>
<tr>
<td>r15</td>
<td>PC</td>
</tr>
</tbody>
</table>
ARM register usage

- Function arguments
- Function results in r0, r1
- Scratch registers within a function
  - Caller save
ARM register usage

- Temporary registers
- Callee save
ARM register usage

- **Intra-Procedure-call scratch register**
- Reserved for use by linker during procedure call
  - May be trashed in the process
- Can be used as scratch register between procedure calls
ARM register usage

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>a1</td>
</tr>
<tr>
<td>r1</td>
<td>a2</td>
</tr>
<tr>
<td>r2</td>
<td>a3</td>
</tr>
<tr>
<td>r3</td>
<td>a4</td>
</tr>
<tr>
<td>r4</td>
<td>v1</td>
</tr>
<tr>
<td>r5</td>
<td>v2</td>
</tr>
<tr>
<td>r6</td>
<td>v3</td>
</tr>
<tr>
<td>r7</td>
<td>v4</td>
</tr>
<tr>
<td>r8</td>
<td>v5</td>
</tr>
<tr>
<td>r9</td>
<td>v6</td>
</tr>
<tr>
<td>r10</td>
<td>v7</td>
</tr>
<tr>
<td>r11</td>
<td>v8</td>
</tr>
<tr>
<td>r12</td>
<td>IP</td>
</tr>
<tr>
<td>r13</td>
<td>SP</td>
</tr>
<tr>
<td>r14</td>
<td>LR</td>
</tr>
<tr>
<td>r15</td>
<td>PC</td>
</tr>
</tbody>
</table>

- Stack pointer.
  - Used during procedure call to build stack frame
ARM register usage

- **Link register**
  - Return address for procedure call

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>a1</td>
</tr>
<tr>
<td>r1</td>
<td>a2</td>
</tr>
<tr>
<td>r2</td>
<td>a3</td>
</tr>
<tr>
<td>r3</td>
<td>a4</td>
</tr>
<tr>
<td>r4</td>
<td>v1</td>
</tr>
<tr>
<td>r5</td>
<td>v2</td>
</tr>
<tr>
<td>r6</td>
<td>v3</td>
</tr>
<tr>
<td>r7</td>
<td>v4</td>
</tr>
<tr>
<td>r8</td>
<td>v5</td>
</tr>
<tr>
<td>r9</td>
<td>v6</td>
</tr>
<tr>
<td>r10</td>
<td>v7</td>
</tr>
<tr>
<td>r11</td>
<td>v8</td>
</tr>
<tr>
<td>r12</td>
<td>IP</td>
</tr>
<tr>
<td>r13</td>
<td>SP</td>
</tr>
<tr>
<td>r14</td>
<td>LR</td>
</tr>
<tr>
<td>r15</td>
<td>PC</td>
</tr>
</tbody>
</table>
ARM register usage

• Program Counter
  – Can be treated a bit like a real register
  – You may see PC arithmetic instructions
Good luck!

• Milestone 1 starts tomorrow.

• Milestone 2: next week
  – Interrupts
  – Context switching

• We’ll start on research ideas:
  • Process dispatch and scheduler activations