1. Course overview
2. Intro to PICOBLAZE, C and Number systems and Boolean Algebra
3. Course overview with microprocessor MU0 (I)
4. Course overview with microprocessor MU0 (II)
5. Verilog HDL
6. Digital system components using schematics and Verilog
7. Combinational logic standard forms. Karnaugh maps
8. Combinational ccts and configurable logic devices
9. Simple Sequential circuits, flip flops
10. Sequential circuits, counters, registers, memories
11. Non-ideal effects in digital circuits
12. Finite State Machines
13. Design of FSMs
14. Design of FSMs
15. Datapaths
16. An introduction to Processor Design
17. The ARM Architecture
18. ARM Assembly Language Programming
19. ARM Assembly Language Programming
20. Programming in C
...

Follow Steve Furber 'ARM System on a Chip Architecture Lecture Notes
ARM Architecture

- Originally Advanced RISC Machines Limited
- Later Acorn RISC Machine

Acorn Computers Limited of Cambridge originally developed ARM from 1983-1985 based on a 1 year student project at Stanford/UC Berkeley that led to more cost effective high performance design to compete with CISC (viz PDP-11, VAX by DEC corporation)
ARM’s visible registers

- **r0**
- **r1**
- **r2**
- **r3**
- **r4**
- **r5**
- **r6**
- **r7**
- **r8**
- **r9**
- **r10**
- **r11**
- **r12**
- **r13**
- **r14**
- **r15 (PC)**

**usable in user mode**

**system modes only**

- **r6_fiq**
- **r9_fiq**
- **r10_fiq**
- **r11_fiq**
- **r12_fiq**
- **r13_fiq**
- **r14_fiq**

- **r13_svc**
- **r13_abt**
- **r15_irq**
- **r13_irq**

- **r14_svc**
- **r14_abt**
- **r14_irq**
- **r14_irq**

- **r0**
- **r1**
- **r2**
- **r3**
- **r4**
- **r5**
- **r6**
- **r7**
- **r8**
- **r9**
- **r10**
- **r11**
- **r12**
- **r13**
- **r14**
- **r15 (PC)**

**CPSR**

- **SPSR_fiq**
- **SPSR_svc**
- **SPSR_abt**
- **SPSR_irq**
- **SPSR_und**

**user mode**

**mode**

- **fiq**
- **svc**
- **abort**
- **irq**
- **undefined**

3213: Digital Systems & Microprocessors: L#18-19
ARM CPSR format

![ARM CPSR format diagram]
ARM Assembly Language Programming

Outline:

- the ARM instruction set
- writing simple programs
- examples

➢ hands-on: writing simple ARM assembly programs
ARM Instruction Set

ARM instructions fall into three categories:

• data processing instructions
  – operate on values in registers

• data transfer instructions
  – move values between memory and registers

• control flow instructions
  – change the program counter (PC)
Data processing instructions

• All operands are 32-bits wide and either:
  ➢ come from registers, or
  ➢ are literals (‘immediate’ values) specified in the instruction

• The result, if any, is 32-bits wide and goes into a register:
  ➢ except long multiplies generate 64-bit results

• All operand and result registers are independently specified
Data processing instructions

• Examples:

  \[
  \begin{align*}
  &\text{ADD } r0, r1, r2 \ ; \ r0 := r1 + r2 \\
  &\text{SUB } r0, r1, #2 \ ; \ r0 := r1 - 2
  \end{align*}
  \]

• Note:

 ➢ everything after the ‘;’ is a comment convenience
 ➢ the result register (r0) is listed first
Data processing instructions

- Arithmetic operations:

  \[
  \begin{align*}
  \text{ADD} & \quad r0, \ r1, \ r2 \ ; \ r0 := r1 + r2 \\
  \text{ADC} & \quad r0, \ r1, \ r2 \ ; \ r0 := r1 + r2 + C \\
  \text{SUB} & \quad r0, \ r1, \ r2 \ ; \ r0 := r1 - r2 \\
  \text{SBC} & \quad r0, \ r1, \ r2 \ ; \ r0 := r1 - r2 + C - 1 \\
  \text{RSB} & \quad r0, \ r1, \ r2 \ ; \ r0 := r2 - r1 \\
  \text{RSC} & \quad r0, \ r1, \ r2 \ ; \ r0 := r2 - r1 + C - 1
  \end{align*}
  \]

- C is the C bit in the CPSR

- the operation may be viewed as unsigned or 2’s complement signed
Data processing instructions

• Bit-wise logical operations:

AND r0, r1, r2 ; r0 := r1 and r2
ORR r0, r1, r2 ; r0 := r1 or r2
EOR r0, r1, r2 ; r0 := r1 xor r2
BIC r0, r1, r2 ; r0 := r1 and not r2

• the specified Boolean logic operation is performed on each bit from 0 to 31

• BIC stands for ‘bit clear’
  – each ‘1’ in r2 clears the corresponding bit in r1
Data processing instructions

• Register movement operations:

\[
\begin{align*}
\text{MOV} & \quad r0, \ r2 \ ; \ r0 := r2 \\
\text{MVN} & \quad r0, \ r2 \ ; \ r0 := \text{not} \ r2
\end{align*}
\]

• MVN stands for ‘move negated’

• there is no first operand (r1) specified as these are unary operations
Data processing instructions

• Comparison operations:

\[
\begin{align*}
\text{CMP} & \ r1, \ r2 \ ; \ \text{set cc on } r1 - r2 \\
\text{CMN} & \ r1, \ r2 \ ; \ \text{set cc on } r1 + r2 \\
\text{TST} & \ r1, \ r2 \ ; \ \text{set cc on } r1 \ \text{and} \ r2 \\
\text{TEQ} & \ r1, \ r2 \ ; \ \text{set cc on } r1 \ \text{xor} \ r2
\end{align*}
\]

• these instructions just affect the condition codes (N, Z, C, V) in the CPSR

– there is no result register (r0)
Data processing instructions

Immediate operands:

• the second source operand (r2) may be replaced by a constant:

```
ADD r3, r3, #1 ; r3 := r3 + 1
AND r8, r7, #&ff ; r8 := r7_{7:0}
```

• # indicates an immediate value
  – & indicates hexadecimal notation
  – C-style notation (#0xff) is also supported

• allowed immediate values are (in general): (0 → 255) x 2^n
Data processing instructions

Shifted register operands

- the second source operand may be shifted
  – by a constant number of bit positions:

  \[ \text{ADD} \ r3, \ r2, \ r1, \ \text{LSL} \ #3; \ r3 := r2+r1<<3 \]

- or by a register-specified number of bits:

  \[ \text{ADD} \ r5, \ r5, \ r3, \ \text{LSL} \ r2; \ r5 += r3<<(r2) \]

  – LSL, LSR mean ‘logical shift left’, ‘logical shift right’
  – ASL, ASR mean ‘arithmetic shift left’, ‘...right’
  – ROR means ‘rotate right’
  – RRX means ‘rotate right extended’ by 1 bit
ARM shift operations

- setting the ‘C’ flag is optional
Data processing instructions

Setting the condition codes

• all data processing instructions may set the condition codes.
  – the comparison operations always do so

• For example, here is code for a 64-bit add:

```
ADDS   r2, r2, r0 ; 32-bit carry-out -> C
ADC    r3, r3, r1 ; added into top 32 bits
```

  – S means ‘Set condition codes’

• The C flag comes from:
  ‣ the adder in arithmetic operations
  ‣ the shifter in logical operations

• the primary use of the condition codes is in control flow – see later
### Data processing instructions

#### Multiplication

- ARM has special multiply instructions

\[
MUL \quad r4, \ r3, \ r2 \quad ; \ r4 := (r3 \times r2)_{[31:0]} 
\]

- only the bottom 32 bits are returned
- immediate operands are not supported
- multiplication by a constant is usually best done with a short series of adds and subtracts with shifts

- there is also a multiply-accumulate form:

\[
MLA \quad r4, \ r3, \ r2, \ r1 \quad ; \ r4 := (r3 \times r2 + r1)_{[31:0]} 
\]

- 64-bit result forms are supported too
Arm Instruction Set

ARM instructions fall into three categories:

- **data processing instructions**
  - operate on values in registers

- **data transfer instructions**
  - move values between memory and registers

- **control flow instructions**
  - change the program counter (PC)
Data transfer instructions

The ARM has 3 types of data transfer instruction:

- **single register loads and stores**
  - flexible byte, half-word and word transfers

- **multiple register loads and stores**
  - less flexible, multiple words, higher transfer rate

- **single register - memory swap**
  - mainly for system use, so ignore for now
Data transfer instructions

Addressing memory

- all ARM data transfer instructions use register indirect addressing.

- Examples of load and store instructions:

  \[
  \text{LDR} \quad r0, \ [r1] \quad ; \ r0 := \text{mem}[r1] \\
  \text{STR} \quad r0, \ [r1] \quad ; \ \text{mem}[r1] := \ r0
  \]

- therefore before any data transfer is possible:

a register must be initialized with the target address
Data transfer instructions

Initializing an address pointer

- any register can be used for an address

- the assembler has special ‘pseudo instructions’ to initialise address registers:

  \texttt{COPY ADR \ r1, TABLE1} \ ; \ r1 \ points \ to \ TABLE1

  \texttt{ADR \ \ r2, \ TABLE2} \ ; \ r1 \ points \ to \ TABLE2

  ..

  \texttt{TABLE1} \ ; \ \texttt{LABEL}

  ..

  \texttt{TABLE2} \ ; \ \texttt{LABEL}
Data transfer instructions

Single register load and store

- the simplest form is just register indirect:

  \[ LDR \quad r0, \ [r1] \quad ; \quad r0 := \text{mem}[r1] \]

- this is a special form of base plus offset:

  \[ LDR \quad r0, \ [r1,#4] \quad ; \quad r0 := \text{mem}[r1+4] \]

- The offset is within 4kBytes.
Data transfer instructions

More...

• another form uses post-indexing

\[ LDR \quad r0, [r1], #4 \quad ; \quad r0 := \text{mem}[r1] \]
\[ ; \quad r1 := r1 + 4 \]

• stores (STR) have the same forms

• Many more types of memory and register load/store modes... stack and even block copy
Arm Instruction Set

ARM instructions fall into three categories:

• data processing instructions
  – operate on values in registers

• data transfer instructions
  – move values between memory and registers

• control flow instructions
  – change the program counter (PC)
Control Flow Instructions

Control flow instructions just switch execution around the program:

\[ B \quad \text{LABEL} \]
\[ .. \quad ; \text{these instructions are skipped} \]
\[ \text{LABEL} \quad .. \]

- normal execution is sequential
- branches are used to change this
  - to move forwards or backwards
- Note: data ops and loads can also change the PC!
Conditional branches

sometimes whether or not a branch is taken depends on the condition codes:

```
MOV      r0, #0 ; initialise counter
LOOP...
ADD      r0, r0, #1 ; increment counter
CMP      r0, #10 ; compare with limit
BNE      LOOP    ; repeat if not equal
         ; else continue
```

– here the branch depends on how CMP sets Z
## Branch conditions

<table>
<thead>
<tr>
<th>Branch</th>
<th>Interpretation</th>
<th>Normal uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Unconditional</td>
<td>Always take this branch</td>
</tr>
<tr>
<td>BAL</td>
<td>Always</td>
<td>Always take this branch</td>
</tr>
<tr>
<td>BEQ</td>
<td>Equal</td>
<td>Comparison equal or zero result</td>
</tr>
<tr>
<td>BNE</td>
<td>Not equal</td>
<td>Comparison not equal or non-zero result</td>
</tr>
<tr>
<td>BPL</td>
<td>Plus</td>
<td>Result positive or zero</td>
</tr>
<tr>
<td>BMI</td>
<td>Minus</td>
<td>Result minus or negative</td>
</tr>
<tr>
<td>BCC</td>
<td>Carry clear</td>
<td>Arithmetic operation did not give carry-out</td>
</tr>
<tr>
<td>BLO</td>
<td>Lower</td>
<td>Unsigned comparison gave lower</td>
</tr>
<tr>
<td>BCS</td>
<td>Carry set</td>
<td>Arithmetic operation gave carry-out</td>
</tr>
<tr>
<td>BHS</td>
<td>Higher or same</td>
<td>Unsigned comparison gave higher or same</td>
</tr>
<tr>
<td>BVC</td>
<td>Overflow clear</td>
<td>Signed integer operation; no overflow occurred</td>
</tr>
<tr>
<td>BVS</td>
<td>Overflow set</td>
<td>Signed integer operation; overflow occurred</td>
</tr>
<tr>
<td>BGT</td>
<td>Greater than</td>
<td>Signed integer comparison gave greater than</td>
</tr>
<tr>
<td>BGE</td>
<td>Greater or equal</td>
<td>Signed integer comparison gave greater or equal</td>
</tr>
<tr>
<td>BLT</td>
<td>Less than</td>
<td>Signed integer comparison gave less than</td>
</tr>
<tr>
<td>BLE</td>
<td>Less or equal</td>
<td>Signed integer comparison gave less than or equal</td>
</tr>
<tr>
<td>BHI</td>
<td>Higher</td>
<td>Unsigned comparison gave higher</td>
</tr>
<tr>
<td>BLS</td>
<td>Lower or same</td>
<td>Unsigned comparison gave lower or same</td>
</tr>
</tbody>
</table>
Control Flow Instructions

Branch and link

ARM’s subroutine call mechanism saves the return address in r14

- note the use of a data processing instruction for return
r14 is often called the link register (lr)
- the only special register use other than the PC

```
BL SUBR ; branch to SUBR
.. ; return to here
SUBR .. ; subroutine entry point
MOV pc, r14 ; return
```
Control Flow Instructions

Conditional execution

• an unusual ARM feature is that all instructions may be conditional:

```assembly
CMP     r0, #5           ; if (r0 != 5) { 
ADDNE   r1, r1, r0       ; r1 := r1 + r0 - r2 
SUBNE   r1, r1, r2       ; }
```

• this removes the need for some short branches  
  – improving performance and code density
Control Flow Instructions

Supervisor calls

• these are calls to operating system functions such as input and output:

  SWI PPT_io ; call an access routine
  ; on PPT

• the range of available calls is system dependent

  – PPT_io uses a single SWI number with the system function specified in r0
Missing Instructions??

Some familiar mnemonics are not present in the ARM instructions:

- **NOT**
  - MVN
  - $R0, R0$
  - ; $R0 := \text{not}(R0)$

- **NEG**
  - RSB
  - $R0, R0, \#0$
  - ; $R0 := 0 - R0$

- **RET**
  - MOV
  - PC, LR
  - ; ‘Leaf’ return

- **PUSH & POP**
  - STMF D SP!, {register list}; Push
  - LDMFD SP!, {register list}; Pop

- **LSL, etc.**
  - MOV
  - $R0, R0, LSL \#5$
  - ;
  - MOV
  - $R0, R0, ASR R1$
  - ; etc.

– Note that the shift can be combined with other operations too.
Programming in assembly

• Compiling
• Linking
• Debugging
• Loading
The Cross-Compiler Toolchain

The Assembler / compiler (gcc, as)
- Produces ARM object format output that can be linked with other assembler or C-language object code
- Assembly language is near machine level with most assembly language commands translating to single ARM (or Thumb) instructions

The Linker (ld)
- Combiners one or more object modules and combines them into an executable program
- Resolves symbolic references between the object files and extracts any required object modules from libraries

The Debugger (gdb)
- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.
The structure of the ARM
cross-development toolkit

C source  C libraries
\rightarrow asm source
C compiler  asembler
\rightarrow .aof
linker
\rightarrow .axf  debug
ARMSd
system model
\rightarrow ARMulator
\rightarrow development board

The structure of the GNU-ARM
cross-development toolkit

C source  C libraries
\rightarrow asm source
C compiler  asembler
\rightarrow .o
linker
\rightarrow .elf  debug
\rightarrow object libraries
\rightarrow development board

Debugger
GDB
\rightarrow HEX converter
FlashMagic
\rightarrow development board

3213: Digital Systems & Microprocessors: L#18-19
Many ARM Toolchains

* Crossworks for ARM (GCC + Commercial IDE) Commercial

* Keil microVision (Commercial)

* IAR Embedded Workbench (Commercial)

* ARM Realview (Commercial)

GNU-ARM Toolchains...

* Yagarto (Yet another GNU ARM toolchain) (GCC + Eclipse) Free

* WINARM (http://winarm.scienceprog.com/)

* Code sourcery G++ (http://www.codesourcery.com/sgpp)
**CodeSourcery G++** ([http://www.codesourcery.com/sgpp](http://www.codesourcery.com/sgpp))

**Sourcery G++ Platforms**

Sourcery G++ generates code for the following target CPU architectures and operating systems:

- ARM EABI
- ARM uClinux
- ARM GNU/Linux
- ColdFire ELF
- ColdFire uClinux
- ColdFire GNU/Linux
- fido ELF
- IA32 GNU/Linux
- IA32 Windows
- MIPS ELF
- MIPS GNU/Linux
- Power EABI
- Power GNU/Linux
- Sparc EABI
- SuperH GNU/Linux

Click on one of the links above for detailed information about the latest release of Sourcery G++ for your target system.

**Selecting the Right Sourcery G++ for Your Target**

Sourcery G++ supports multiple target operating systems as well as multiple CPU architectures. Use the following table to choose the configuration that is right for you.

*This table applies to the target system on which your applications will run, not to the host system on which you run Sourcery G++.*

<table>
<thead>
<tr>
<th>Target Platform</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EABI/ELF</td>
<td>RTOS systems or &quot;bare metal&quot; systems where no operating system is present. These configurations should not be used to build Linux kernels or applications.</td>
</tr>
<tr>
<td>uClinux®</td>
<td>Systems running uClinux, i.e. Linux on CPUs without an MMU. Use Sourcery G++ to build both the uClinux kernel and applications.</td>
</tr>
<tr>
<td>GNU/Linux®</td>
<td>Systems running &quot;full&quot; Linux, i.e., Linux on CPUs with an MMU. Use Sourcery G++ to build both the Linux kernel and applications.</td>
</tr>
<tr>
<td>Microsoft Windows®</td>
<td>Systems running Microsoft Windows 2000, or later.</td>
</tr>
</tbody>
</table>

Sourcery G++ includes C and C++ runtime libraries that you can use on your target system, but Sourcery G++ does not include a GNU/Linux, uClinux, or RTOS kernel.
**Using CodeSourcery G++**

**MUST DO...**

**Assemble...**

```
arm-none-linux-gnueabi-gcc -mcpu=arm7tdmi -c LedBlink.s
```

or

```
```

**Link...**

```
arm-none-linux-gnueabi-ld -Ttext=0 LedBlink.o -o LedBlink.elf
```

**HEX conversion...**

```
arm-none-linux-gnueabi-objcopy -O ihex LedBlink.elf LedBlink.hex
```

#########################################################

**EXTRA...**

**Disassembly....**

```
arm-none-linux-gnueabi-objdump -d LedBlink.elf
```

**View size report....**

```
arm-none-linux-gnueabi-size LedBlink.elf
```

3213: Digital Systems & Microprocessors: L#18-19
Applications of LED Display

LED Display circuit of Board will be connected as Sink Current type and it is used with +3.3V Power Supply. In this case, it runs with Logic “0” (0V) and stops running with Logic “1” (+3.3V). It is controlled by 2 GPIOs that are P3[25] and P3[26]. This circuit is used to test operation of Output.

If we want to use this function, we must set function of P3[25] and P3[26] to be Output Port first and then control the desired Logic for P3[25] and P3[26] as example below.
// Config Pin GPIO = P3[26:25] Drive LED
PINSEL7 &= 0xFFC3FFFF;  // Config P3[26:25] = GPIO Function
PINMODE7 &= 0xFFC3FFFF; // Enable Pull-Up on P3[26:25]
FIO3DIR |= 0x02000000;  // Set P3[25] = Output
FIO3DIR |= 0x04000000;  // Set P3[26] = Output

FIO3CLR  = 0x02000000;  // LED(P3[25]) = ON
FIO3CLR  = 0x04000000;  // LED(P3[26]) = ON

FIO3SET  = 0x02000000;  // LED(P3[25]) = OFF
FIO3SET  = 0x04000000;  // LED(P3[26]) = OFF
Table 133. GPIO register map (local bus accessible registers - enhanced GPIO features)

<table>
<thead>
<tr>
<th>Generic Name</th>
<th>Description</th>
<th>Access</th>
<th>Reset value</th>
<th>PORTn Register Address &amp; Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIODIR</td>
<td>Fast GPIO Port Direction control register. This register individually controls the direction of each port pin.</td>
<td>R/W</td>
<td>0x0</td>
<td>FIO0DIR - 0x3FFF C000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO1DIR - 0x3FFF C020</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO2DIR - 0x3FFF C040</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO3DIR - 0x3FFF C060</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO4DIR - 0x3FFF C080</td>
</tr>
<tr>
<td>FIMASK</td>
<td>Fast Mask register for port. Writes, sets, clears, and reads to port (done via writes to FIOPIN, FIOSET, and FIOCLR, and reads of FIOPIN) alter or return only the bits enabled by zeros in this register.</td>
<td>R/W</td>
<td>0x0</td>
<td>FIO0MASK - 0x3FFF C010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO1MASK - 0x3FFF C030</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO2MASK - 0x3FFF C050</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO3MASK - 0x3FFF C070</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO4MASK - 0x3FFF C090</td>
</tr>
<tr>
<td>FIOPIN</td>
<td>Fast Port Pin value register using FIMASK. The current state of digital port pins can be read from this register, regardless of pin direction or alternate function selection (as long as pins are not configured as an input to ADC). The value read is masked by ANDing with inverted FIMASK. Writing to this register places corresponding values in all bits enabled by zeros in FIMASK. Important: if a FIOPIN register is read, its bit(s) masked with 1 in the FIMASK register will be set to 0 regardless of the physical pin state.</td>
<td>R/W</td>
<td>0x0</td>
<td>FIO0PIN - 0x3FFF C014</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO1PIN - 0x3FFF C034</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO2PIN - 0x3FFF C054</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO3PIN - 0x3FFF C074</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO4PIN - 0x3FFF C094</td>
</tr>
<tr>
<td>FIOSET</td>
<td>Fast Port Output Set register using FIMASK. This register controls the state of output pins. Writing 1s produces highs at the corresponding port pins. Writing 0s has no effect. Reading this register returns the current contents of the port output register. Only bits enabled by 0 in FIMASK can be altered.</td>
<td>R/W</td>
<td>0x0</td>
<td>FIO0SET - 0x3FFF C018</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO1SET - 0x3FFF C038</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO2SET - 0x3FFF C058</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO3SET - 0x3FFF C078</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO4SET - 0x3FFF C098</td>
</tr>
<tr>
<td>FIOCLR</td>
<td>Fast Port Output Clear register using FIMASK0. This register controls the state of output pins. Writing 1s produces lows at the corresponding port pins. Writing 0s has no effect. Only bits enabled by 0 in FIMASK0 can be altered.</td>
<td>WO</td>
<td>0x0</td>
<td>FIO0CLR - 0x3FFF C01C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO1CLR - 0x3FFF C03C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO2CLR - 0x3FFF C05C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO3CLR - 0x3FFF C07C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FIO4CLR - 0x3FFF C09C</td>
</tr>
</tbody>
</table>
5.12 Pin Mode select register 0 (PINMODE0 - 0xE002 C040)
This register controls pull-up/pull-down resistor configuration for PORT0 pins 0 to 15.

### Table 115. Pin function select register 7 (PINSEL7 - address 0xE002 C01C) bit description (LPC2364/65/66/67/68 and LPC2387)

<table>
<thead>
<tr>
<th>PINSEL7</th>
<th>Pin name</th>
<th>Function when 00</th>
<th>Function when 01</th>
<th>Function when 10</th>
<th>Function when 11</th>
<th>Reset value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:0</td>
<td>P3.16</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>00</td>
</tr>
<tr>
<td>3:2</td>
<td>P3.17</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>00</td>
</tr>
<tr>
<td>5:4</td>
<td>P3.18</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>00</td>
</tr>
<tr>
<td>7:6</td>
<td>P3.19</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>00</td>
</tr>
<tr>
<td>9:8</td>
<td>P3.20</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>00</td>
</tr>
<tr>
<td>11:10</td>
<td>P3.21</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>00</td>
</tr>
<tr>
<td>13:12</td>
<td>P3.22</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>00</td>
</tr>
<tr>
<td>15:14</td>
<td>P3.23</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>00</td>
</tr>
<tr>
<td>17:16</td>
<td>P3.24</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>00</td>
</tr>
<tr>
<td>19:18</td>
<td>P3.25</td>
<td>GPIO Port 3.25</td>
<td>Reserved</td>
<td>MAT0.0</td>
<td>PWM1.2</td>
<td>00</td>
</tr>
<tr>
<td>21:20</td>
<td>P3.26</td>
<td>GPIO Port 3.26</td>
<td>Reserved</td>
<td>MAT0.1</td>
<td>PWM1.3</td>
<td>00</td>
</tr>
<tr>
<td>23:22</td>
<td>P3.27</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>00</td>
</tr>
<tr>
<td>25:24</td>
<td>P3.28</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>00</td>
</tr>
<tr>
<td>27:26</td>
<td>P3.29</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>00</td>
</tr>
<tr>
<td>29:28</td>
<td>P3.30</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>00</td>
</tr>
<tr>
<td>31:30</td>
<td>P3.31</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>00</td>
</tr>
</tbody>
</table>
PINSEL7 = 0xE002C01C
PINMODE7 = 0xE002C05C

FIO3DIR = 0x3FFFC060
FIO3SET = 0x3FFFC078
FIO3CLR = 0x3FFFC07C

.text
@; Executable code follows
.align 2
    .align 2
    .global _start @; "_start" is required by the linker
_start:

.... You code goes here

    .align 2
```assembly
ldr    r1,=PINSEL7
ldr    r2,=0xFFC3FFFF
str    r2, [r1]

ldr    r1,=PINMODE7
ldr    r2,=0xFFC3FFFF
str    r2, [r1]

ldr    r1,=FIO3DIR
ldr    r2,=0x02000000
str    r2, [r1]

loop:
	@; ON LED (GPIO-12) (GPIO LOW-ON)
    ldr    r0,=FIO3CLR
    ldr    r1,=0x02000000
    str    r1, [r0,#0x00]
    ldr    r4,=100000
    1:      nop
    subs   r4, r4, #1
    bne    1b
    @; OFF LED (GPIO-12) (GPIO HIGH-OFF)
    ldr    r0,=FIO3SET
    ldr    r1,=0x02000000
    str    r1, [r0,#0x00]
    ldr    r4,=100000
    1:      nop
    subs   r4, r4, #1
    bne    1b
    bal    loop
```
ARM memory organization

```
   bit 31          bit 0
   23  22  21  20
  19  18  17  16
   ______          ______
word16             word16
  15  14  13  12
  half-word14 half-word12
  11  10  9  8
   ______          ______
word8             word8
  7  6  5  4
  byte6  half-word4
  3  2  1  0
byte3  byte2  byte1  byte0
```

byte address
Multiple register transfer addressing modes

STMIA r9!, {r0, r1, r5}
STMIB r9!, {r0, r1, r5}
STMAD r9!, {r0, r1, r5}
STMDB r9!, {r0, r1, r5}
The mapping between the stack and block copy views of the load and store multiple instructions

<table>
<thead>
<tr>
<th>Increment</th>
<th>Ascending</th>
<th>Descending</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Full</td>
<td>Empty</td>
</tr>
<tr>
<td><strong>Before</strong></td>
<td>STMIB</td>
<td>STMFA</td>
</tr>
<tr>
<td><strong>After</strong></td>
<td>STMIA</td>
<td>STMEA</td>
</tr>
<tr>
<td></td>
<td>LDMDB</td>
<td>LDMEA</td>
</tr>
<tr>
<td><strong>Before</strong></td>
<td>LDMDA</td>
<td>LDMFA</td>
</tr>
<tr>
<td><strong>After</strong></td>
<td>LDMDA</td>
<td>LDMFA</td>
</tr>
<tr>
<td></td>
<td>STMDA</td>
<td>STMED</td>
</tr>
</tbody>
</table>
Naming ARM

• ARMxyzTDMIEJFS
  - x: series
  - y: MMU
  - z: cache
  - T: Thumb
  - D: debugger
  - M: Multiplier
  - I: Interrupt
  - E: Enhanced
  - J: Jazelle
  - F: Floating-point
  - S: Source
Popular ARM architecture

• ARM7TDMI
  - 3 pipeline stages
  - One of the most used ARM-version (for low-end systems)

• ARM9TDMI
  - Compatible with ARM7
  - 5 pipeline stages
  - Separate instruction and data cache

• ARM11
ARM architecture

- Load/store architecture
- A large array of uniform registers
- Fixed-length 32-bit instructions
- 3-address instructions
## Processor modes

<table>
<thead>
<tr>
<th>Processor mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>usr</td>
</tr>
<tr>
<td></td>
<td>Normal program execution mode</td>
</tr>
<tr>
<td>FIQ</td>
<td>fiq</td>
</tr>
<tr>
<td></td>
<td>Supports a high-speed data transfer or channel process</td>
</tr>
<tr>
<td>IRQ</td>
<td>irq</td>
</tr>
<tr>
<td></td>
<td>Used for general-purpose interrupt handling</td>
</tr>
<tr>
<td>Supervisor</td>
<td>svc</td>
</tr>
<tr>
<td></td>
<td>A protected mode for the operating system</td>
</tr>
<tr>
<td>Abort</td>
<td>abt</td>
</tr>
<tr>
<td></td>
<td>Implements virtual memory and/or memory protection</td>
</tr>
<tr>
<td>Undefined</td>
<td>und</td>
</tr>
<tr>
<td></td>
<td>Supports software emulation of hardware coprocessors</td>
</tr>
<tr>
<td>System</td>
<td>sys</td>
</tr>
<tr>
<td></td>
<td>Runs privileged operating system tasks</td>
</tr>
</tbody>
</table>
ARM architecture

- 37 registers
  - 1 Program counter
  - 1 current program status registers
  - 5 saved program status registers
  - 30 general purpose registers
Registers

- Only 16 registers are visible to a specific mode. A mode could access
  - A particular set of r0-r12
  - r13 (sp, stack pointer)
  - r14 (lr, link register)
  - r15 (pc, program counter)
  - Current program status register (cpsr)
Register organization

<table>
<thead>
<tr>
<th>User</th>
<th>FIQ</th>
<th>IRQ</th>
<th>SVC</th>
<th>Undef</th>
<th>Abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r8</td>
<td>r8</td>
<td>r13 (sp)</td>
<td>r13 (sp)</td>
<td>r13 (sp)</td>
<td>r13 (sp)</td>
</tr>
<tr>
<td>r9</td>
<td>r9</td>
<td>r14 (lr)</td>
<td>r14 (lr)</td>
<td>r14 (lr)</td>
<td>r14 (lr)</td>
</tr>
<tr>
<td>r10</td>
<td>r10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r11</td>
<td>r11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r12</td>
<td>r12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r13 (sp)</td>
<td>r13 (sp)</td>
<td>r13 (sp)</td>
<td>r13 (sp)</td>
<td>r13 (sp)</td>
<td>r13 (sp)</td>
</tr>
<tr>
<td>r14 (lr)</td>
<td>r14 (lr)</td>
<td>r14 (lr)</td>
<td>r14 (lr)</td>
<td>r14 (lr)</td>
<td>r14 (lr)</td>
</tr>
<tr>
<td>r15 (pc)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cpsr</td>
<td>spsr</td>
<td>spsr</td>
<td>spsr</td>
<td>spsr</td>
<td>spsr</td>
</tr>
</tbody>
</table>

3213: Digital Systems & Microprocessors: L#22_23
General-purpose registers

- 6 data types (signed/unsigned)
- All ARM operations are 32-bit. Shorter data types are only supported by data transfer operations.
Program counter

- Store the address of the instruction to be executed
- All instructions are 32-bit wide and word-aligned
- Thus, the last two bits of pc are undefined.
Program status register (CPSR)

- **Mode bits:** M4, M3, M2, M1, M0
- **State bit:** I, F, T
- **Overflow:** N
- **Carry/borrow:** Z
- **Zero:** C
- **Negative:** V

3213: Digital Systems & Microprocessors: L#22_23
Summary

- Load/store architecture
- Most instructions are RISCy, operate in single cycle.
  - Some multi-register operations take longer.
- All instructions can be executed conditionally.
Linux usage on Intel (4 level Von Neuman)

Figure 1-1. A split view of the kernel

3213: Digital Systems & Microprocessors: L#18