Efficient C Code for ARM Devices

Satyaki Mukherjee

Efficient C Code for ARM Devices

Abstract

"You can make your C code better quickly, cheaply and easily. Simple techniques are capable of yielding surprising improvements in system performance, code size and power consumption. This session looks at how software applications can make most efficient use of the instruction set, memory systems, hardware accelerators and power-saving hardware features in order to deliver significantly lower power consumption. You will learn tricks which you can use the day you get back to your office!"

There are two sides to this coin

- Application
  - Minimal instruction count
  - Minimal memory access
  - Cache-friendly data accesses
    - Line length and boundary
  - Efficient use of stack
    - Parameter count
    - Return in regs
  - Task/thread partition
  - SIMD/Vectorization

- OS
  - System power management
    - Subsystem power
    - DVFS
    - Multicore power
  - Power-friendly spinlocks
  - Sensible task scheduling
  - Cache configuration and usage

Before we begin...

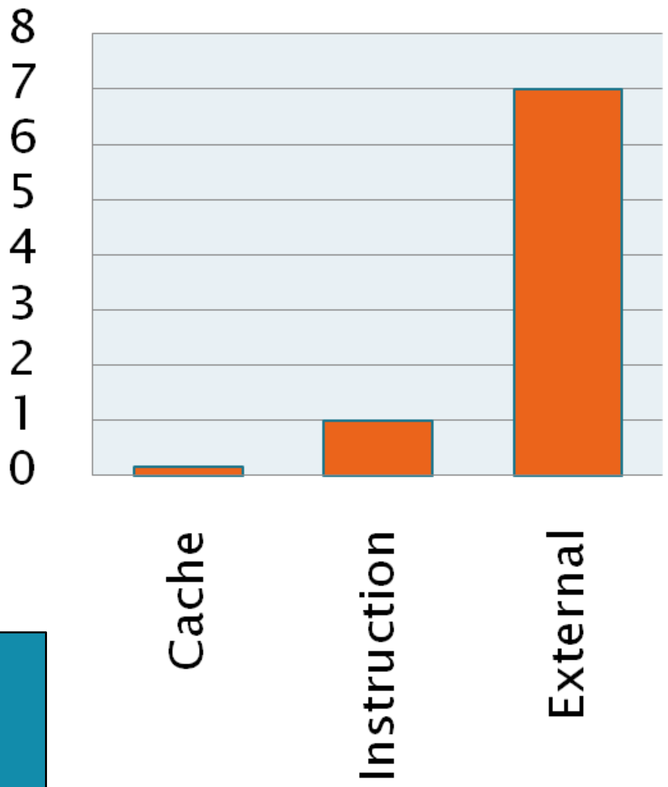
- Some things are so basic that we have to assume you are doing them before we can talk about anything else...

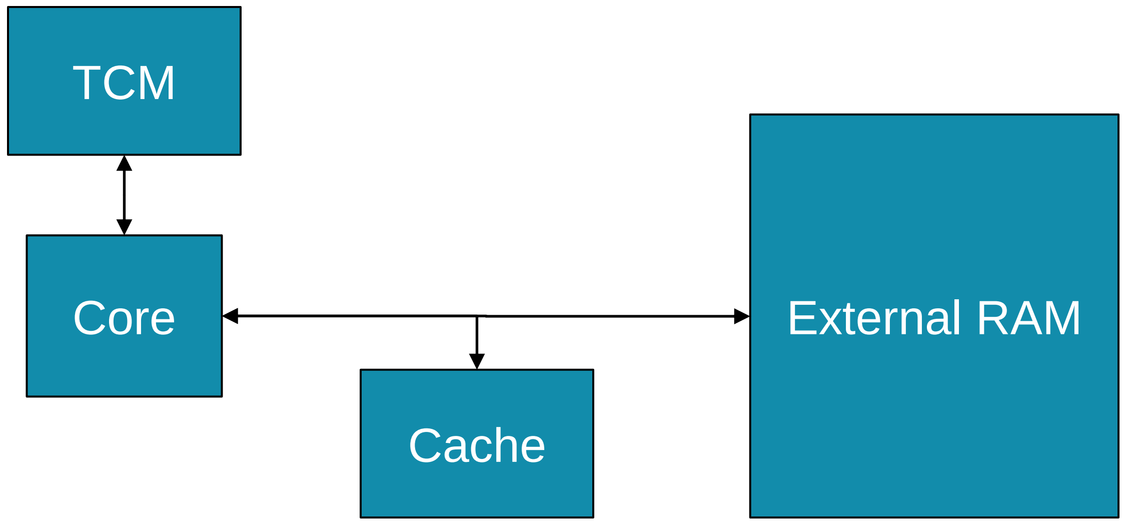
...some things are vital!

- In our software optimization course we show how to improve performance of a common video codec suite by 200-250% using a combination of
  - Correct compiler configuration
  - Software optimization
  - Architectural optimization
  - System optimization
  - NEON vectorization
- This is impressive BUT it is dwarfed completely by the penalty of not configuring the cache correctly
  - Turning on the data cache can improves performance by up to 5000%!

Memory use

- Memory use is expensive
  - Takes longer than executing instructions
  - Consumes significantly more power
- Keep it close
- Access it as little as possible





Speed = Power

- In both senses
  - Increasing speed increases power consumption

  BUT
  - More efficient code completes more quickly
- Therefore – optimize for SPEED
- Favour computation over communication
- Only be as accurate as you need – is fixed point enough?

- BUT smaller code might cache more efficiently – avoiding memory accesses

Good coding practice

- Make sure things match

`Data types`	↔	`Architecture`
`Code`	↔	`Instruction Set`
`Memory use`	↔	`Platform`
`Conventions`	↔	`Tools`

- Write sensible code
- Make sure you know what the tools are doing

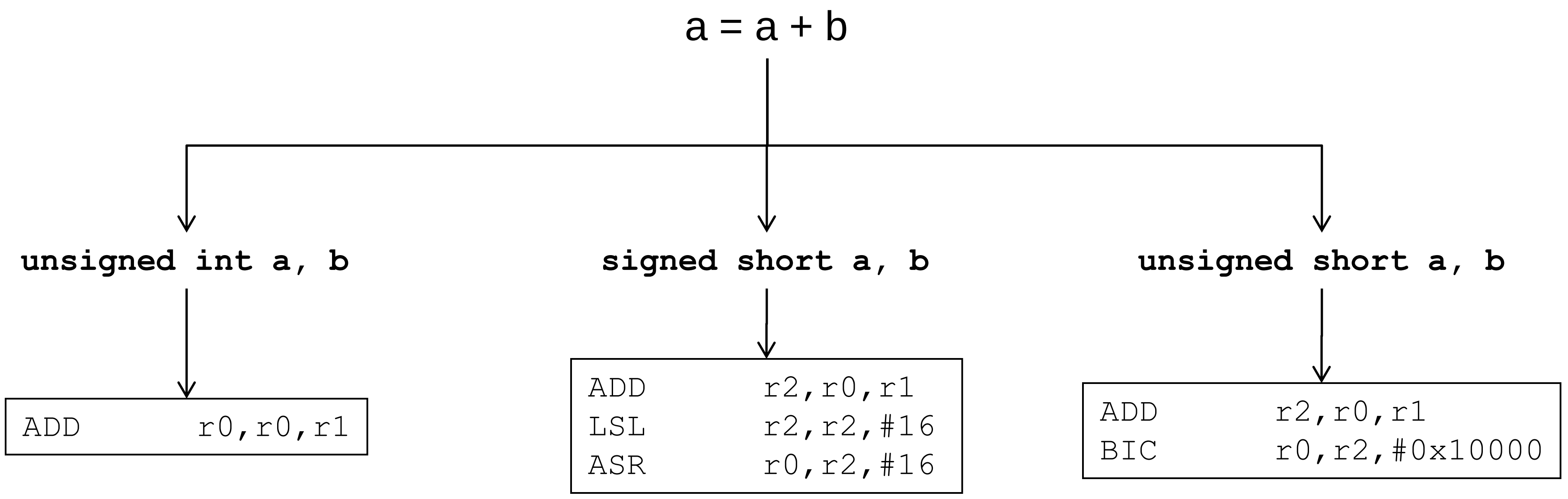
Data types

- In general, unsigned, word-sized integer types are best
  - Sub-word items require truncation or sign-extension
  - Doubleword types require higher alignment, especially when passed as parameters
  - Loading signed or unsigned halfwords and signed bytes takes longer on some cores
  - Loading unaligned items works but there can be a performance penalty
- The compiler can “hide” these effects in many cases ...

  ... but not always

Variable selection (size)

- The ARM is a 32-bit architecture, so optimal code is generated when working with 32-bit (word) sized variables



```
ADD         r0,r0,r1
```

```
ADD         r2,r0,r1
LSL         r2,r2,#16
ASR         r0,r2,#16
```

```
ADD         r2,r0,r1
BIC         r0,r2,#0x10000
```

Array element sizes

- To calculate the address of an element of an array, the compiler must multiply the size of the element by the index...

```
&(a[i]) ≡ a + i * sizeof(a)
```

- If the element size is a power of 2, this can be done with a simple inline shift

- For an array at [r3], to access the first word of element number r1
- Element size = 12:

```
ADD r1, r1, r1, LSL #1    ; r1 = 3 * r1
LDR r0, [r3, r1, LSL #2]  ; r0 = *(r1 + 4 * r1)
```

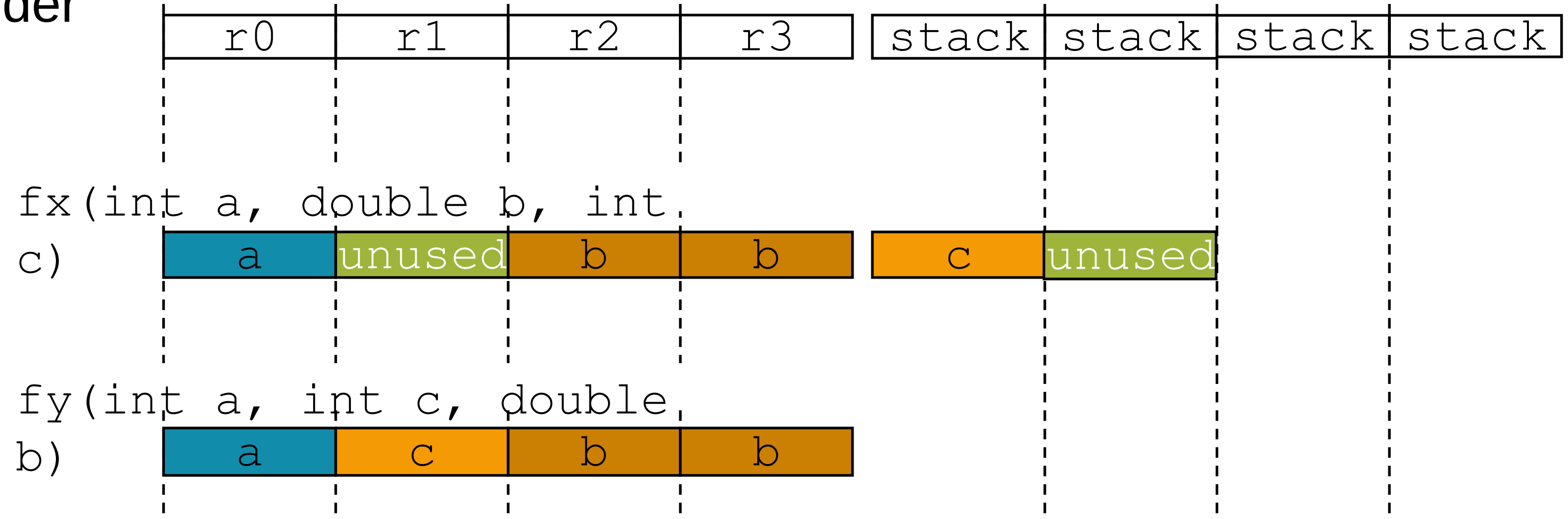
- Element size = 16:

```
LDR r0, [r3, r1, LSL #4]  ; r0 = *(r3 + 16 * r1)
```

Parameter Passing

- The AAPCS has rules about 64-bit types
  - 64-bit parameters must be 8-byte aligned in memory
  - 64-bit arguments to functions must be passed in an even + consecutive odd register

    (i.e. r0+r1 or r2+r3) or on the stack at an 8-byte aligned location
- Registers or stack will be 'wasted' if arguments are listed in a sub-optimal order



- Remember the hidden *this* argument in r0 for non-static C++ member functions

The C compiler doesn’t know everything

- The compiler is constrained in many areas
- Help the compiler out where you can

- Some things are not ARM-specific...
  - For instance, using `do-while` when termination condition will always pass on first iteration

  ...but some are

- The ARM compiler provides helpful things like

  __pure

  __restrict (equivalent to C99 “restrict”)

  __promise

Looks __promising

```
void f(int *x, int n)
{
    int i;

    __promise((n > 0) && ((n & 7) == 0));

    for (i = 0; i < n; i++)
     {
            x[i]++;
     }
}
```

Tells the compiler that the loop index is greater than 0 and divisible by 8

The C compiler can’t do everything

- Some instructions are never automatically generated by the C compiler
  - Q*, *SAT, many SIMD

- You may need to resort to hand-coding in assembler to get access to these
- You can get a long way with intrinsics

```
unsigned int SMUADop(unsigned int val1, unsigned int val2)
{
  return(__smuad(val1,val2));
}
```

Vectorization using NEON instrinsics

- The following function processes an array of data in memory

```
for(i = 0; i < 8; i++)
{
    dst[i] = src[i] * dst[i];
}
```

8x

```
:
MOV       r2,#0
LDR       r3,[r1,r2,LSL #2]
LDR       r4,[r0,r2,LSL #2]
MUL       r3,r3,r4
STR       r3,[r0,r2,LSL #2]
ADD       r2,r2,#1
CMP       r2,#8
BLT       {pc}-0x18 ; 0x38
:
```

- Vectorizing this using NEON™ C intrinsics

```
for(i = 0; i < 8; i+=4)
{
    n_dst = vld1q_s32(dst+i);
    n_src = vld1q_s32(src+i);
    n_dst = vmulq_s32(n_dst,n_src);
    vst1q_s32(dst+i,n_dst);
}
```

2x

```
:
MOV         r2,#0
ADD         r4,r1,r2,LSL #2
ADD         r3,r0,r2,LSL #2
ADD         r2,r2,#4
VLD1.32     {d0,d1},[r4]
CMP         r2,#8
VLD1.32     {d2,d3},[r3]
VMUL.I32    q0,q1,q0
VST1.32     {d0,d1},[r3]
BLT         {pc}-0x20 ; 0x8
:
```

Use your cache wisely

- Data which is aligned to cache line boundaries increases effectiveness of manual and automatic preload)

- Access data in a cache-friendly manner (i.e. sequentially)

```
int myarray[16] __attribute__((aligned(64)));
```

GOOD

BAD

```
for (i = 0; i < SIZE; i++)
{
  for (j = 0; j < SIZE; j++)
  {
    for (k = 0; k < SIZE; k++)
    {
      a[i][j] += b[i][k] * c[k][j];
    }
  }
}
```

- Make data structures match cache line length
- Use preload: PLD, PLE

Base Pointers

```
extern int a;
extern int b;

int func(void)
{
   return (a+b);
}
```

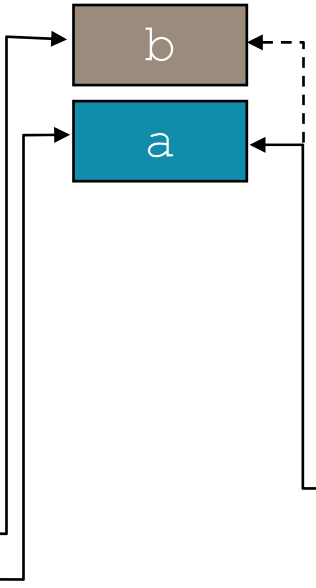
```
int a;
int b;

int func(void)
{
     return (a+b);
}
```

a and b defined externally

```
LDR r0,  [pc,#16]
LDR r0,  [r0,#0]
LDR r1,  [pc,#12]
LDR r1,  [r1,#0]
ADD r0,r0,r1
BX  lr

DCD “address of a”
DCD “address of b”
```



a and b defined within the module in which they are used

```
LDR r0,  [pc,#12]
LDR r1,  [r0,#0]
LDR r0,  [r0,#4]
ADD r0,r0,r1
BX  lr

DCD “base address of a and b”
```

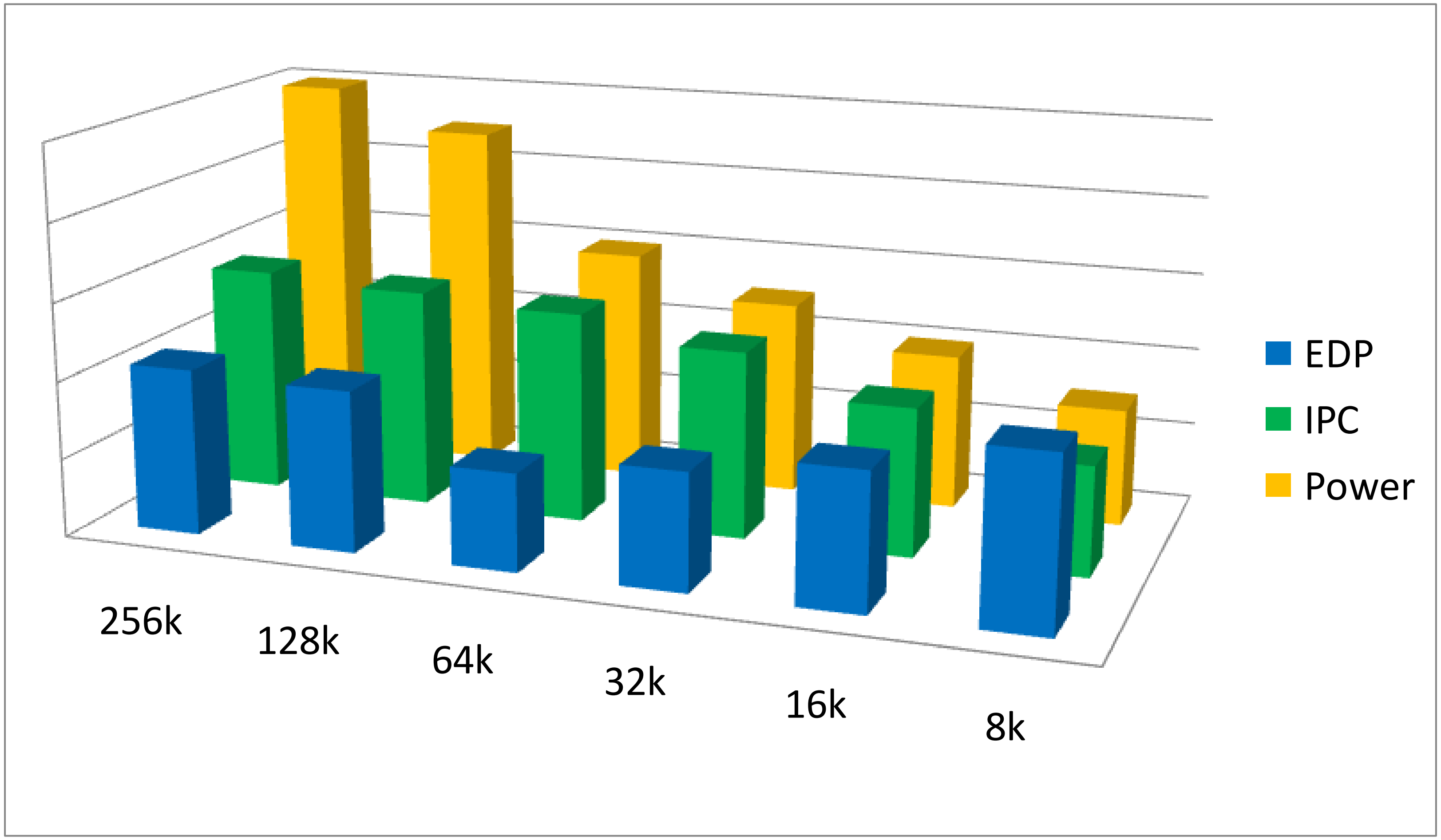
Note that this is not done at -O0

Power management

- The great debate

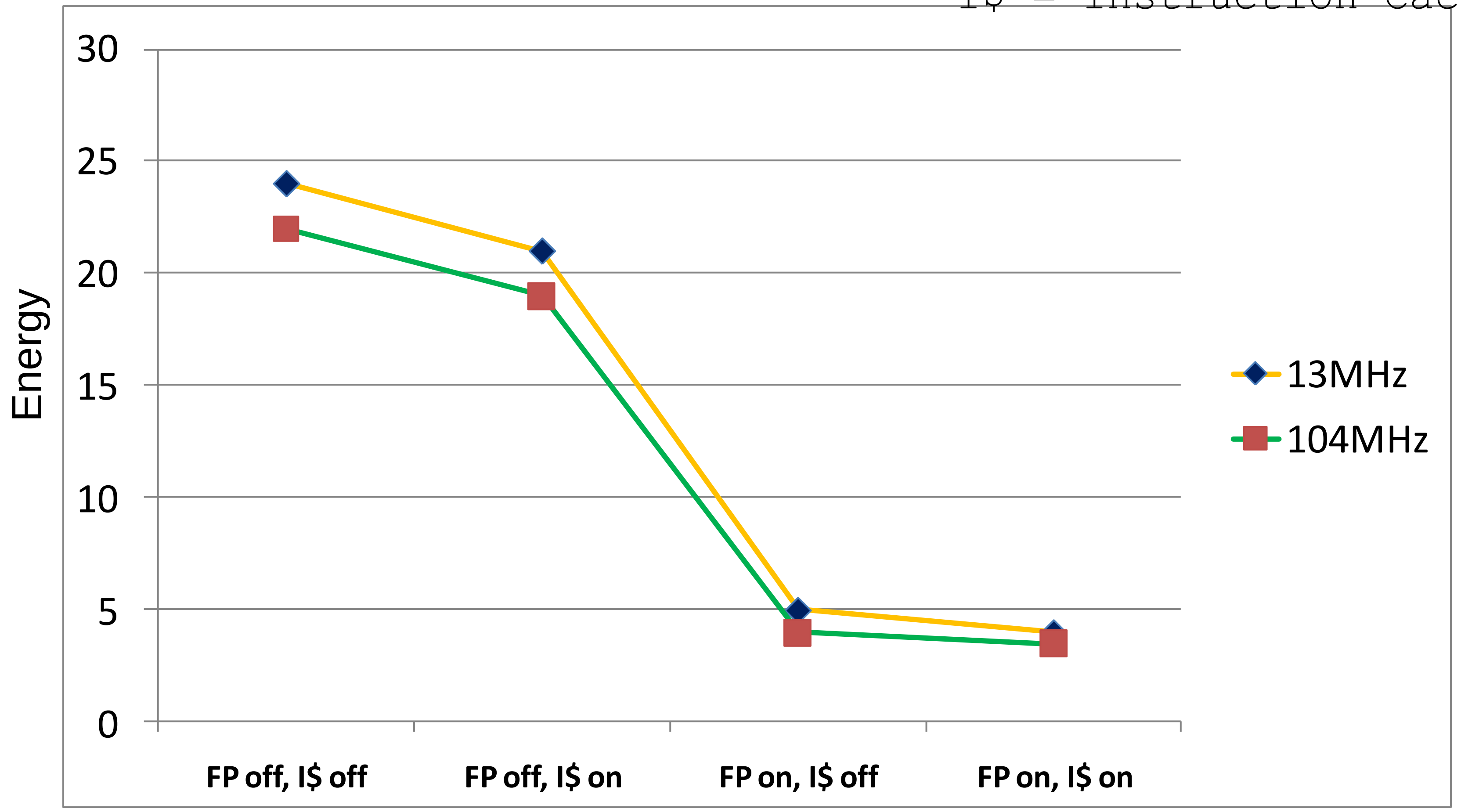
- Race to completion vs Just-in-time
- Static power vs. Dynamic power
  - As geometry gets smaller, static power becomes larger proportion. Actually larger at 65nm
- There are no standard answers!
- Energy Delay Product (EDP)
  - Metric which combines measure of energy consumption and timely completion
  - Goal is to minimize EDP

Energy benefits of cache



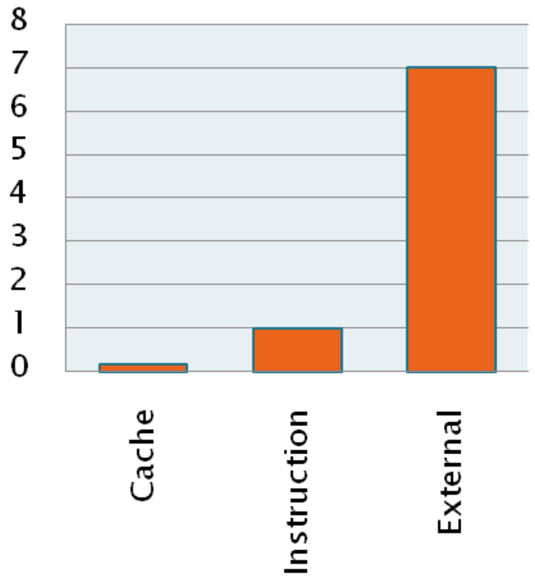
Should we turn it on?

```
FP = Floating Point Unit
I$ = Instruction Cache
```



Minimizing memory access

- Memory access is expensive
- Use registers as much as possible
  - Minimize live variables, don't take address of automatics
  - Inlining can reduce stack usage
- Make best use of cache
  - WB/WA for stack/heap/globals, WB/RA for buffers
  - Multi-level L1 WT/RA, L2 WB/WA
  - Beware of false-sharing in multicore systems
    - MESI is a "write-invalidate coherency protocol"
    - Two processes modifying same cache line will bounce it back and forth – solution is put volatile shared data in separate cache lines



The Architecture for the Digital World® ARM®

Systems-wide power management

- Implementation-dependent power controllers
- Standard components e.g. NEON
- Multicore

Power modes in ARM cores

- Some variant of the following
- Run
  - Normal operation, fully powered (but NEON could be off)
- Standby
  - Clock stopped, still powered, restart from next instruction, transparent to program
- Dormant
  - Powered down, RAM retained, context save required, restart from reset vector
- Power-Off
  - Core, cache, RAM all powered down, full restart from reset vector required

SMP Linux support for power-saving

- Current versions of SMP Linux support
  - CPU hotplug
  - Load-balancing and dynamic re-prioritization
  - Intelligent scheduling to minimize cache migration/thrashing (strong affinity between processes and cores)
  - DVFS per CPU
  - Individual CPU power state management
- Handles interface with external power management controller

Example – Freescale i.MX51

- Freescale i.MX51 contains HW NEON monitor which automatically powers down NEON when unused
  - Countdown timer set by program
  - Decrements automatically
  - Reset when NEON instruction is executed
  - Raises interrupt on timeout, triggering state retention powerdown
- Subsequent NEON instruction causes an Undef exception
  - Undef handler powers up and restores NEON
  - Return to re-execute NEON instruction
- MP3/WMA decode power consumption down by 20%*

*Freescale figures

Example – TI OMAP 4

- Dual-core Cortex™-A9 MPCore™ up to 1GHz
- Supports
  - DVFS – Dynamic Voltage and Frequency Scaling
  - AVS – Adaptive Voltage Scaling
  - DPS – Dynamic Power Switching
  - SLM – Static Leakage Management
- OS can select from a number of OPPs (Operating Performance Points)
- OS can drive DVFS depending on current load
- Power consumption from 600uW to 600mW depending on load

In multicore systems

- System efficiency
  - Intelligent/dynamic task prioritization
  - Load balancing
  - Power-conscious spinlocks
- Computational efficiency
  - Data, task and functional parallelism
  - Low synchronization overhead
- Data efficiency
  - Efficient use of memory system
  - Efficient use of SMP caches (minimize trashing, false sharing)

Conclusions

- Configure properly
  - Tools and platform
- Memory
  - Minimize expensive external accesses
- Instruction count
  - Optimize for speed
- Subsystems
  - Take every opportunity to power down as far and as often as possible

Thank You

Please visit www.arm.com for ARM related technical details

For any queries contact < Salesinfo-IN@arm.com >