HARVARD COMPUTER ARCHITECTURE

Чэнь Лэй (гр. 6085/1)

1 Two types of computer architectures

There are 2 computer architectures, which are different in the way of accessing memories: von Neumann Architecture (also names “Princeton Architecture”) and Harvard Architecture.

The von Neumann Architecture has following specialties [1]:

1. Instructions and Data are stored in the same memory.
2. Instructions and Data share one memory system. (The Length of bit, the same form of address)

And the Harvard Architecture has following factors [2]:

1. Physically separates storage and signal pathway for instructions and data.
2. Generally, the bit of Instructions is wider than Data.
3. For some computers, the Instruction memory is read-only.
4. In cases without caches, the Harvard Architecture is more efficient than von-Neumann.

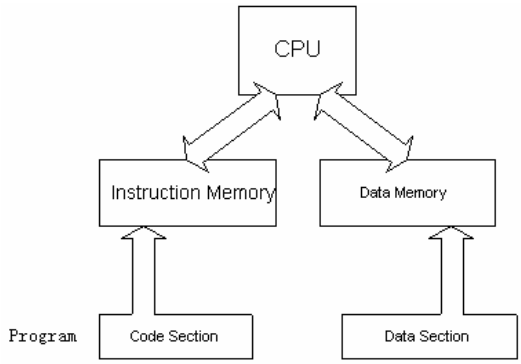
Bus structures of the two are also different: Harvard architecture has separate data and instruction busses, allowing transfers simultaneously on both busses. Von Neumann architecture has only one bus which is used for both data transfers and instruction fetches, and therefore data transfers and instruction fetches must be scheduled - they can not be performed at the same time.

1.1 Compare of the two in running programs

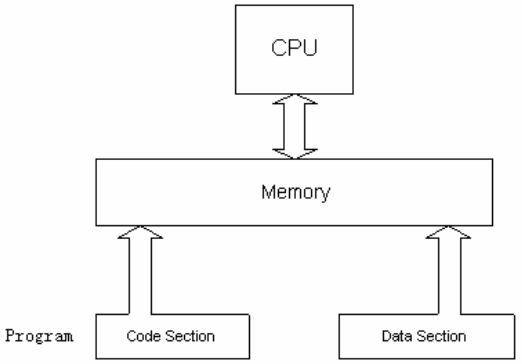
By reason of the wider bit of instructions, Harvard Architecture supports more instructions with less hardware requiring. For example, the ARM9 processor has 24-bit instruction length, so that it could have 2^{24}=16777216 instructions, which are much more than 16-bit processors have (65536). So, with uniform bus width which von-Neumann architecture has, the processor has to take more requirement of hardware in data length, if it wants to have 24-bit instruction width.

Secondly, two buses accessing memory synchronously provides more CPU time. The von Neumann processor has to perform a command in 2 steps (first read an instruction, and then read the data that the instruction requires. On picture-1 has shown the process of reading instructions and their data.). But the Harvard architecture can read both instruction and its data at the same time (On picture-2 shows that 2 buses

work synchronically). Evidently, the parallel method is faster and more efficiently, because it only takes one step for each command.



Pic. 1



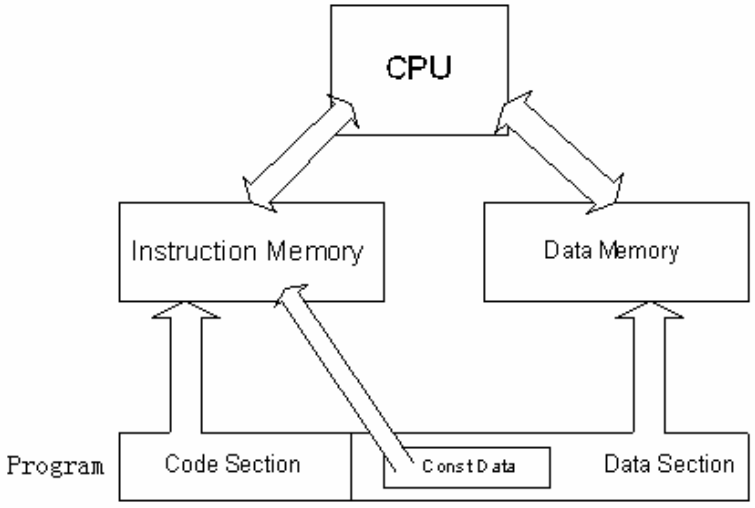
Pic.2

As a result, Harvard architecture is especially powerful in digital signal process. Because most commands in DSP require data memory access, the 2-bus-architecture saves much more CPU time.

1.2 Modified Harvard Architecture

There is one type of modified Harvard Architecture, on which there is an addition pathway between CPU and the Instruction memory. It allows words in instruction memory be treated as “read-only data”, so that const data (e.g. text string) can be read into Instruction memory instead of Data memory [3]. This method preserves more data memory for read/write variables. On picture 3 shows how Modified Harvard Architecture works, when there is constant data in data memory.

Modified Harvard Architecture allows the contents of the Instruction memory to be treated as if it were data, but the high-level programming language standard C doesn’t support this architecture, so that need to add In-line assembly and non-standard extended C library.



Pic. 3

The principal historical advantage of the Harvard architecture (simultaneous access to more than one memory system) has been

nullified by modern cache systems, allowing the more flexible Von Neumann machine equal performance in most cases. The Modified Harvard architecture has thus been relegated to niche applications where the ease-of-programming / complexity / performance trade-off favors it.

Since the C Language was not designed for Harvard architectures, it was designed for Von Neumann architectures where code and data exist in the same address space, so that any compiler for a Harvard architecture processor, like the AVR, has to use other methods to operate with separate address spaces. Some compilers use non-standard C language keywords, or they extend the standard syntax in ways that are non-standard. The AVR toolset takes a different approach.

The development environment for AVR will be mentioned again in chapter 2.1.3.

Further discuss about this feature can be found on reference [15], in which the author had given out a example in C language to explain how realizes accessing strings in the code section in GCC.

1.3 Using caches in both architectures

Speed of CPU is faster than the speed of accessing main memory. So, modern high performance computers with caches have the "incorporate aspect" of both von-Neumann and Harvard architectures.

On von-Neumann architecture, cache on CPU is divided into instruction cache and data cache, and the main memory needn't to be separated into 2 sections. So that, the von-Neumann programmers can work on Harvard Architectures without knowing the hardware.

As a result, with the help of caches, both architectures gain efficiencies.

2 Harvard Architecture on embedded systems

2.1 An inchoate Harvard Architecture microcontroller

In 1996, Atmel developed a Harvard architecture 8-bit RISC single chip microcontroller, which was one of the first microcontroller families to use on-chip flash memory for program storage, as opposed to One-Time Programmable ROM, EPROM, or EEPROM used by other microcontrollers at the time. [9]

2.1.1 AVR Device Architecture

Data RAM: Flash, EEPROM, and SRAM are all integrated onto a single chip, removing the need for external memory (though still available on some devices).

Program Memory (Flash): Program instructions are stored in non-volatile Flash memory. Although they are 8-bit MCUs, each instruction

takes one or two 16-bit words. The size of the program memory is occasionally indicated in the naming of the device itself (e.g., the ATmega64x line has 64 kB of Flash).

Internal Data Memory: The data address space consists of the register file, I/O registers, and SRAM. The AVRs have 32 single-byte registers and are classified as 8-bit RISC devices.

Internal Registers: The working registers are mapped in as the first 32 memory addresses (0000_{16}-$001F_{16}$) followed by the 64 I/O registers (0020_{16}-$005F_{16}$). Actual SRAM starts after the register sections (address 006016). Even though there are separate addressing schemes and optimized opcodes for register file and I/O register access, all can still be addressed and manipulated as if they were in SRAM.

EEPROM: Some AVR microcontrollers have internal Electrically Erasable Programmable Read Only Memory (EEPROM) for semi-permanent data storage (as versus FLASH for semi-permanent program storage). Like Flash memory, EEPROM can maintain its contents when electrical power is removed. This internal EEPROM memory is not mapped into the MCU's addressable memory space. It can only be accessed the same way an external peripheral device is, using special pointer registers and read/write instructions which makes EEPROM access much slower than other internal RAM. Since the number of writes to EEPROM is not unlimited - Atmel specifies 100,000 write cycles in their datasheets - a well designed EEPROM write routine should compare the contents of an EEPROM address with desired contents and only perform an actual write if contents need to be changed.

Program Execution: Atmel's AVRs have a single level pipeline design. This means the next machine instruction is fetched as the current one is executing. Most instructions take just one or two clock cycles, making AVRs relatively fast among the eight-bit microcontrollers. The AVR family of processors was designed with the efficient execution of compiled C code in mind and has several built-in pointers for the task.

2.1.2 AVR Instruction Set

The AVR Instruction Set is more orthogonal(*An instruction set is said to be orthogonal if any instruction can use any register in any addressing mode.*) than most eight-bit microcontrollers; however, it is not completely regular: Pointer registers X, Y, and Z have addressing capabilities that are different from each other. [10]

Register locations R0 to R15 have different addressing capabilities than register locations R16 to R31. I/O ports 0 to 31 have different addressing capabilities than I/O ports 32 to 63.

CLR affects flags, while SER does not, even though they are complementary instructions. CLR set all bits to zero and SER sets them

to one. (Note that CLR is pseudo-op for EOR R, R; and SER is short for LDI R, $FF. Math operations such as EOR modify flags while moves/loads/stores/branches such as LDI do not.)

Arithmetic operations work on registers R0-R31 but not directly on RAM and take one clock cycle, except for multiplication and word-wide addition (ADIW and SBIW) which take two cycles.

RAM and I/O space can be accessed only by copying to or from registers. Indirect access (including optional postincrement, predecrement or constant displacement) is possible through registers X, Y, and Z. All accesses to RAM takes two clock cycles. Moving between registers and I/O is one cycle. Moving eight-bit or sixteen-between registers or constant to register is also one cycle. Reading program memory (LPM) takes three cycles.

2.1.3 Development Environment for Atmel AVR

The GNU Compiler Collection (usually shortened to GCC) is a set of compilers produced for various programming languages by the GNU Project. GCC is a key component of the GNU toolchain. As well as being the official compiler of the GNU system, GCC has been adopted as the standard compiler by most other modern Unix-like computer operating systems, including Linux, the BSD family and Mac OS X. GCC has been ported to a wide variety of computer architectures, and is widely deployed as a tool in commercial, proprietary and closed source software development environments. GCC is also used in popular embedded platforms like Symbian, Playstation and Sega Dreamcast. [8]

Originally named the GNU C Compiler, because it only handled the C programming language, GCC 1.0 was released in 1987, and the compiler was extended to compile C++ in December of that year. Front ends were later developed for Fortran, Pascal, Objective C, Java, and Ada, among others.

Distributed by the Free Software Foundation (FSF) under the terms of the GNU General Public License (GNU GPL) and the GNU Lesser General Public License (GNU LGPL), GCC is free software.

Many AVRs have limited amount of RAM in which to store data, but may have more Flash space available. The AVR is a Harvard architecture processor, where Flash is used for the program, RAM is used for data, and they each have separate address spaces. It is a challenge to get constant data to be stored in the Program Space, and to retrieve that data to use it in the AVR application.

The problem is exacerbated by the fact that the C Language was not designed for Harvard architectures, it was designed for Von Neumann architectures where code and data exist in the same address space. This

means that any compiler for a Harvard architecture processor, like the AVR, has to use other means to operate with separate address spaces.

Some compilers use non-standard C language keywords, or they extend the standard syntax in ways that are non-standard. The AVR toolset takes a different approach. GCC has a special keyword, "__attribute__" that is used to attach different attributes to things such as function declarations, variables, and types. This keyword is followed by an attribute specification in double parentheses. In AVR GCC, there is a special attribute called "progmem". This attribute is use on data declarations, and tells the compiler to place the data in the Program Memory (Flash).

AVR-Libc provides a simple macro PROGMEM that is defined as the attribute syntax of GCC with the progmem attribute. This macro was created as a convenience to the end user, as we will see below. The PROGMEM macro is defined in the <avr/pgmspace.h> system header file. It is difficult to modify GCC to create new extensions to the C language syntax, so instead, avr-libc has created macros to retrieve the data from the Program Space. These macros are also found in the <avr/pgmspace.h> system header file.

2.2 ARM Family

The ARM architecture (previously, the Advanced RISC Machine, and prior to that Acorn RISC Machine) is a 32-bit RISC processor architecture developed by ARM Limited that is widely used in a number of embedded designs. Because of their power saving features, ARM CPUs are dominant in the mobile electronics market, where low power consumption is a critical design goal [5].

It is well known, that ARM7 is widely used in on-chip systems and embedded systems instead of old 8-bit microcontrollers. But ARM7 is still in von Neumann bus architecture, that restricts improve the efficiency of the processors. Only simple OS are able to run on ARM7 chips (such as uCOS-II and uCLinux). But ARM9, which adopts Harvard architecture, supports more embedded operation systems (such as WinCE). A simple ARM9 system even can work as a PC, if a monitor connects to its AGP port. It's all because of its better performance. ARM9 realizes the same (v4T) instruction set that ARM7 and is thus binary compatible.

Pipeline length is 5 stages instead of ARM7 3 stages. This allows for faster clocking, and TDMI extensions are available too. With these factors, efficiency of processor enhanced greatly, especially in field of DSP.

Now let's have a look at a table [5][11], in which compares the members in ARM family by their different features.

Family	Architecture Version	Core	Cache(I/D) /MMU	Pipeline	FPU&DSP Support	SMP
ARM7TDMI	ARMv4T	ARM710T	8kB unified, MMU	3-stage	Enhanced DSP, FPA10	
		ARM720T	8kB unified, MMU	5-stage	Enhanced DSP	
StrongARM	ARMv4	SA-110	16KB/16KB, MMU			
ARM8	ARMv4	ARM810	8kB unified, MMU	5-stage		
ARM9TDMI	ARMv4T	ARM920T	16kB/16kB, MMU			
		ARM922T	8kB/8kB, MMU		Enhanced DSP	
ARM9E	ARMv5TE	ARM946E-S	Variable, TCMs		Enhanced DSP	
	ARMv5TEJ	ARM926EJ	Variable, TCMs, MMU		Enhanced DSP, VFP	
ARM10E	ARMv5TE	ARM1020E	32kB/32kB, MMU	6-stage	Enhanced DSP, VFP	
		ARM1020E	16kB/16kB, MMU	6-stage	Enhanced DSP, VFP	
XScale	ARMv5TE	IOP34x	32kB/32kB, MMU		Enhanced DSP	
		PXA255	32kB/32kB, MMU	7-stage	Enhanced DSP	
ARM11	ARMv6	ARM1136JF	Variable, MMU	8-stage	VFP	
	ARMv6K	ARM11MP Core	Variable, MMU	9-stage	VFP	1-4 core SMP
Cortex	ARMv7-A	Cortex-A8	Variable(L1+L2) MMU+ TrustZone	13stage	VFP	
		Cortex-A9	MMU+ TrustZone	13stage	VFP	
		Cortex-A9 MPCore	MMU+ TrustZone	13stage	VFP	1-4 core SMP

In this table, we can perceive their memory architecture from their cache structures. Unified Cache means that it is von-Neumann processor, and divided cache is the character of Harvard processors.

In addition, let's have some explanations about a few key-concepts in the table.

MMU: memory management unit (MMU), sometimes called paged memory management unit (PMMU), is a computer hardware component

responsible for handling accesses to memory requested by the central processing unit (CPU). Its functions include translation of virtual addresses to physical addresses (i.e., virtual memory management), memory protection, cache control, bus arbitration, and, in simpler computer architectures (especially 8-bit systems), bank switching.

Modern MMUs typically divide the virtual address space (the range of addresses used by the processor) into pages, whose size is 2^n, usually a few kilobytes. The bottom n bits of the address (the offset within a page) are left unchanged. The upper address bits are the (virtual) page number. The MMU normally translates virtual page numbers to physical page numbers via an associative cache called a Translation Lookaside Buffer (TLB). When the TLB lacks a translation, a slower mechanism involving hardware-specific data structures or software assistance is used. The data found in such data structures are typically called page table entries (PTEs), and the data structure itself is typically called a page table. The physical page number is combined with the page offset to give the complete physical address.

An MMU also reduces the problem of fragmentation of memory. After blocks of memory have been allocated and freed, the free memory may become fragmented (discontinuous) so that the largest contiguous block of free memory may be much smaller than the total amount. With virtual memory, a contiguous range of virtual addresses can be mapped to several non-contiguous blocks of physical memory. [7]

FPU: floating point unit (FPU) is a part of a computer system specially designed to carry out operations on floating point numbers. Typical operations are addition, subtraction, multiplication, division, and square root. Some systems (particularly older, microcode-based architectures) can also perform various "transcendental" functions such as exponential or trigonometric calculations, though in most modern processors these are done with software library routines.

In most modern general purpose computer architectures, one or more FPUs are integrated with the CPU; however many embedded processors, especially older designs, do not have hardware support for floating point operations.

Not all computer architectures have a hardware FPU. In the absence of an FPU, many FPU functions can be emulated, which saves the added hardware cost of an FPU but is significantly slower. Emulation can be implemented on any of several levels - in the CPU as microcode, as an operating system function, or in user space code.

In most modern computer architectures, there is some division of floating point operations from integer operations. This division varies significantly by architecture; some, like the Intel x86 have dedicated

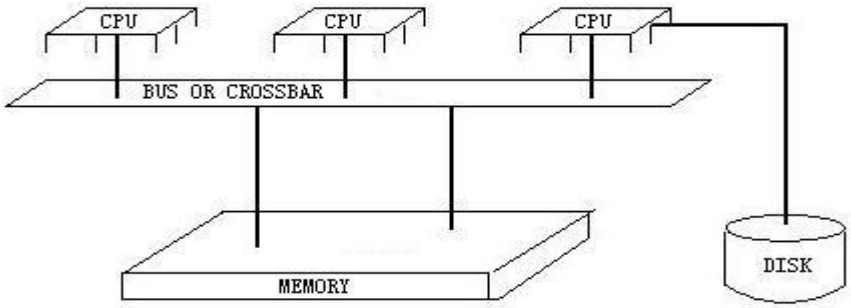
floating point registers, while some take it as far as independent clocking schemes. [6]

Add-on FPUs(Coprocessor): In addition to the Intel architectures, FPUs as coprocessors were available for the Motorola 680x0 line. These FPUS, the 68881 and 68882, were common in 68020/68030-based workstations like the Sun 3 series. They were also commonly added to higher-end models of Apple Macintosh and Commodore Amiga series, but unlike IBM PC-compatible systems, sockets for adding the coprocessor were not as common in lower end systems. With the 68040, Motorola integrated the FPU and CPU, but like the x86 series, a lower cost 68LC040 without an integrated FPU was also available. [6]

SMP: Symmetric multiprocessing, or SMP, is a multiprocessor computer architecture where two or more identical processors are connected to a single shared main memory. Most common multiprocessor systems today use an SMP architecture. In case of multi-core processors, the SMP architecture applies to the cores, treating them as separate processors.

SMP systems allow any processor to work on any task no matter where the data for that task are located in memory; with proper operating system support, SMP systems can easily move tasks between processors to balance the workload efficiently.

Picture-4 shows a diagram of a typical SMP system. Three processors are connected to the same memory module through a bus or crossbar switch.



Pic. 4

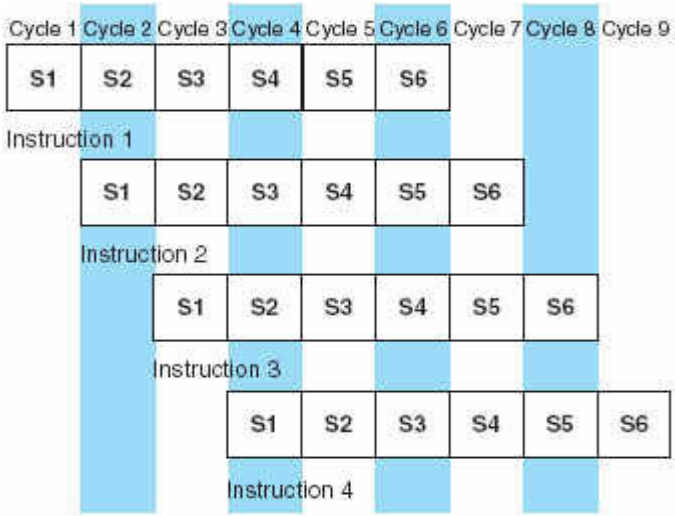
SMP is one of the earliest styles of multiprocessor machine achitectures, typically used for building smaller computers with up to 8 processors. Larger computer systems might use newer architectures such as NUMA (Non-Uniform Memory Access), which dedicates different memory banks to different processors. In a NUMA architecture, processors may access local memory quickly and remote memory more slowly. This can dramatically improve memory throughput as long as the data is localized to specific processes (and thus processors). On the

downside, NUMA makes the cost of moving data from one processor to another, as in workload balancing, more expensive. The benefits of NUMA are limited to particular workloads, notably on servers where the data is often associated strongly with certain tasks or users. [4]

Instruction Pipeline: Conceptually, each pulse of the computer's clock is used to control one step in the sequence, but sometimes additional pulses can be used to control smaller details within one step. Some CPUs break the fetch-decode-execute cycle down into smaller steps, where some of these smaller steps can be performed in parallel. This overlapping speeds up execution. This method, used by all current CPUs, is known as pipelining.

Suppose the fetch-decode-execute cycle were broken into the following "mini-steps": Fetch instruction, Decode op-code, Calculate effective address of operands, Fetch operands, Execute instruction, and Store result.

Picture-5 provides an illustration of computer pipelining with overlapping stages. We see each clock cycle and each stage for each instruction (where S1 represents the fetch, S2 represents the decode, S3 is the calculate state, S4 is the operand fetch, S5 is the execution, and S6 is the store).



Pic. 5

We see from Picture-5 that once instruction 1 has been fetched and is in the process of being decoded, we can start the fetch on instruction 2. When instruction 1 is fetching operands, and instruction 2 is being decoded, we can start the fetch on instruction 3. Notice these events can occur in parallel, very much like an automobile assembly line. [13]

At last we have a talk about an Example of application ARM7 – Picotux. Picotux is the smallest computer running Linux in the world. There are several different kinds of Picotux available, but the main one is

the Picotux 100. It is 35 mm × 19 mm × 19 mm and just barely larger than a RJ45 connector (picture 6). Two communication interfaces are provided, 10/100 Mbit/s half/full duplex Ethernet and a serial port with up to 230.400 bit/s. Five additional lines can be used for either general input/output or serial handshaking.

Pic. 6

The Picotux 100 operates a 55 MHz 32-bit ARM 7 Netsilicon NS7520 processor, with 2 MB of Flash Memory (750 KB of which contains the OS) and 8 MB SDRAM Memory. The operating system of the picotux is μClinux 2.4.27 Big Endian. BusyBox 1.0 is used as main shell. The picotux system runs at 250 mA only and 3.3 V +/- 5%. [16]

2.2 Problem of Harvard Architecture on ARM9 system

On ARM9, Instruction memory is read-only. If it used completely separated memory and isolated instruction-data memory, the On-line Debug and Self Modify would be impossible [11]. Compilers generally embed data (literal pools) within the code, and it is often also necessary to be able to write to the instruction memory space, for example in the case of self modifying code, or, if an ARM debugger is used, to set software breakpoints in memory. If there are two completely separate, isolated memory systems, this is not possible.

As result, using a simple, unified memory system on embedded system is inefficient. (Unless make data into both buses. But this way is no better than a von-Neumann architecture computer.)

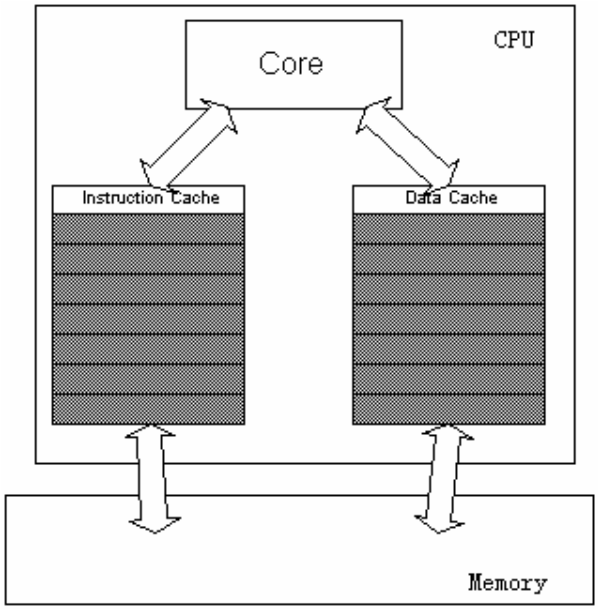
2.3 Resolve the problem

There are 2 main methods towards the problem: use caches inside CPU and adopt “uniform addressing” method [11]. The caches will improve the calculation speed remarkably, and the method of “uniform addressing” will allow using a simple & unified memory system on embedded systems. It is very significative in high speed clock CPU and pipelines.

2.3.1 Using Caches

At higher clock speeds, caches are useful as the memory speed is proportionally slower. Harvard architectures tend to be targeted at higher

performance systems, and so caches are nearly always used in such systems.



Pic. 7

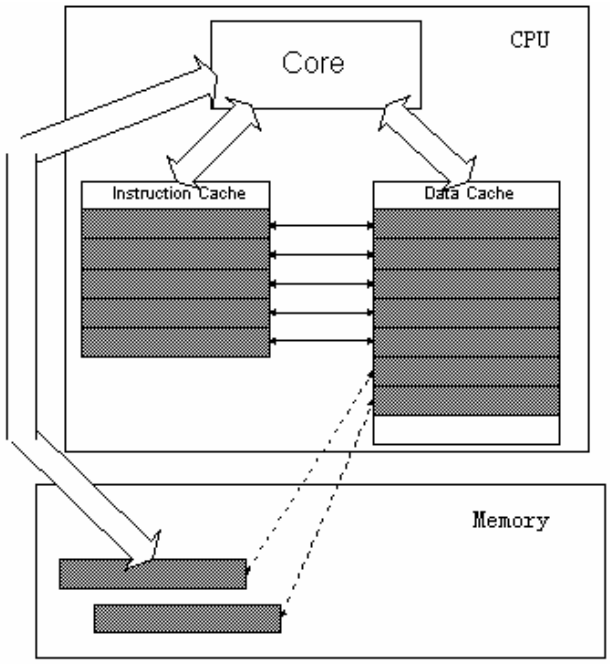
Such a system would have separated caches for each bus. Trying to use a shared cache on Harvard architecture would be very inefficient since then only one bus can be fed at a time. Having two caches means it is possible to feed both buses simultaneously, which is exactly necessary for Harvard architecture. In other words, it makes Harvard Architecture Computers be able to use Single Unified memory system (picture 7), and, in addition, allows the method of Uniform Addressing to be realized.

2.3.2 Uniform Addressing

"Uniform Addressing" means using the same address space for both instructions and with a very simple unified memory system. On picture 6 shows the structure of method "uniform addressing".

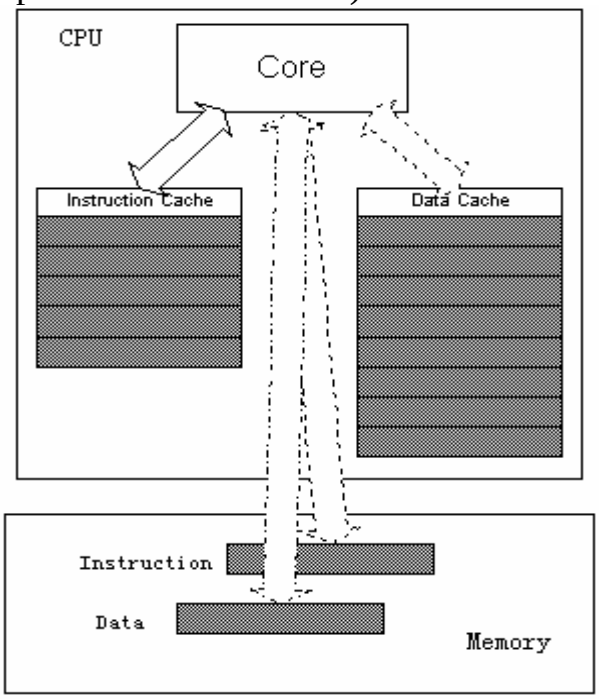
It is this method of addressing that gets around the problem of literal pools and self modifying code. What it does mean, however, is that when starting with empty caches, it is necessary to fetch instructions and data from the single memory system, at the same time. Obviously, two memory accesses are needed therefore before the core has all the data needed. This performance will be no better than von Neumann architecture. However, as the caches fill up, it is much more likely that the instruction or data value has already been cached, and so only one of the two has to be fetched from memory. The other can be supplied directly from the cache with no additional delay. The best performance is achieved when both instructions and data are supplied by the caches, with no need to access external memory at all. [11][12]

Having using “Uniform Addressing”, the Harvard Architecture computers only need to access the main memory once when one of caches is full.



Pic. 8

On picture 8 we can see that after instruction cache is full, the blocks corresponding to the ones in data cache will be located in the main memory. So CPU accesses memory only once, but the von-Neumann Architecture computer needs to read the main memory twice (picture 9). (Speed of accessing the main memory is much slower than caches, so von-Neumann computer takes more time.)



Pic. 9

This method is the most sensible compromise and the architecture used by ARM's Harvard processor cores. Two separate memory systems can perform better, but would be difficult to implement.

2.4 µClinux

µClinux was originally created by D. Jeff Dionne and Kenneth Albanowski in 1998. Initially they targeted the Motorola DragonBall family of embedded 68k processors (specifically the 68328 series) on a 2.0.33 Linux kernel. After releasing their initial work a developer community soon sprung up to extend their work to newer kernels and other microprocessor architectures. In early 1999 support was added for the Motorola (now Freescale) ColdFire family of embedded microprocessors. ARM processor support also became available later that year.

Although originally targeting 2.0 series Linux kernels, it now has ports based on Linux 2.4 and Linux 2.6. There were never any µClinux extensions applied to the 2.2 series kernels.

Since version 2.5.46 of the Linux kernel the major parts of µClinux have been integrated with the main line kernel for a number of processor architectures. Greg Ungerer (who originally ported µClinux to the Motorola ColdFire family of processors) continues to maintain and actively push core µClinux support into 2.6 series Linux kernels. In this regard µClinux is essentially no longer a separate fork of Linux.

The project continues to develop patches and supporting tools for using Linux on microcontrollers. µClinux has support for many architectures, and forms the basis of many products, like network routers, security cameras, DVD or MP3 players, VoIP phone or Gateways, scanners, and card readers.

Currently, it supports following processors: Freescale DragonBall, Freescale ColdFire, ADI Blackfin, ETRAX, Freescale QUICC, ARM7TDMI / MC68EN302, Sigma Design DVD system, Intel i960, PRISMA, Atari 68k, Xilinx MicroBlaze, NEC V850E, Hitachi H8.

Like any part of the Linux kernel, the extensions (in the form of patches) are licensed under the GPL.

Although strictly speaking µClinux is only the operating system kernel, the µClinux project also produced a C standard library called uClibc (now separately maintained) and a "userland" which can be used as a complete operating system for embedded systems called "µClinux-dist".

The "µClinux-dist" software package contains libraries, applications and tools. It can be configured and built into a kernel with root file system. It was first released by Greg Ungerer in 1999 as the µClinux-coldfire package. In the following years it came to support many

architecture families, and now can even build standard Linux architectures (such as x86) as well.

The "μClinux-dist" userland utilities contain tiny http servers, a small 'sh like' shell, and even a fun ascii art Star Wars film. It also contains many other well known Open Source packages, like Samba and freeswan, all of which run on μClinux systems.

For example, the iPodLinux project uses μClinux for its kernel, and so does Mattel's Juice Box, as well as well-known Nintendo DS Linux port, DSLinux, the lesser-known PlayStation port Runix, and the PlayStation Portable port.

It has also been used in the Picotux, advertised as the smallest computer running Linux in the world. The Pictotux 100 is 35 mm × 19 mm × 19 mm, but the Blackfin based Minotaur BF537 is smaller, at 26.5mm x 26.5mm x 4.2mm. [14]

Conclusions

First of all, the Harvard architecture is more powerful due to its parallelity. With the help of caches and multi-stage pipeline structure, the Harvard architecture increases efficiency observably.

Secondly, the Modified Harvard architecture preserves more data memory for read/write variables by considering constant data as if they were instruments.

For modern high speed embedded ARM systems, separated caches are set in CPU, instead of separate the main memory into 2 sections. It allows system use a unified memory system, so that hardware design will be easier, and so do software programmers.

The method “uniform addressing” will cost a little CPU efficiency, but solved the conflict problem of ARM systems.

In addition, we have discussed about differences among the members in ARM family, so that we can see the development of embedded system and its applications. Apparently, the improvement of architecture enhanced capabilities of ARM processors.

The AVR microprocessor is also mentioned as an inchoate Harvard processor, because of its representative development environment.

At last we talked about μClinux, although it runs on both Harvard and von-Neumann architecture. It evidently and observably improves the development efficiency of embedded system. As a result, newer and more powerful systems can be developed easily and efficiently, because designers can insert program sections, which are well running on former Linux system, into their subjects.

Reference

[1] En.wikipedia.org, “von Neumann architectures”.
[2] En.wikipedia.org, “Harvard architectures”.
[3] En.wikipedia.org, “Modified Harvard architectures”.
[4] En.wikipedia.org, “Symmetric multiprocessing”.
[5] En.wikipedia.org, “ARM architecture”.
[6] En.wikipedia.org, “Floating point unit”.
[7] En.wikipedia.org, “Memory management unit”.
[8] En.wikipedia.org, “GNU Compiler Collection”.
[9] En.wikipedia.org, “Atmel AVR”.
[10] En.wikipedia.org, “Atmel AVR instruction set”.
[11] www.arm.com, “What is the difference between a von Neumann architecture and a Harvard architecture”,
http://www.arm.com/support/faqip/3738.html
[12] ARM Ltd. Co., ARM9TDMI Technical Reference Manual (Rev 3), 2000, chapter 3.
[13] L. Null and J. Lobur, Computer Organization and Architecture, 1999, chapter 1.6-1.8; chapter 5, Jones and Bartlett Publishers © 2003 (ebook: http://mugur.roz.md/computer-science/computer-science/__toc.htm)
[14] http://www.uclinux.org
[15] “Data in Program Space”, http://www.nongnu.org/avr-libc/user-manual/pgmspace.html
[16] “Technical Data of picotux”, http://www.picotux.com/techdatae.html

Contents