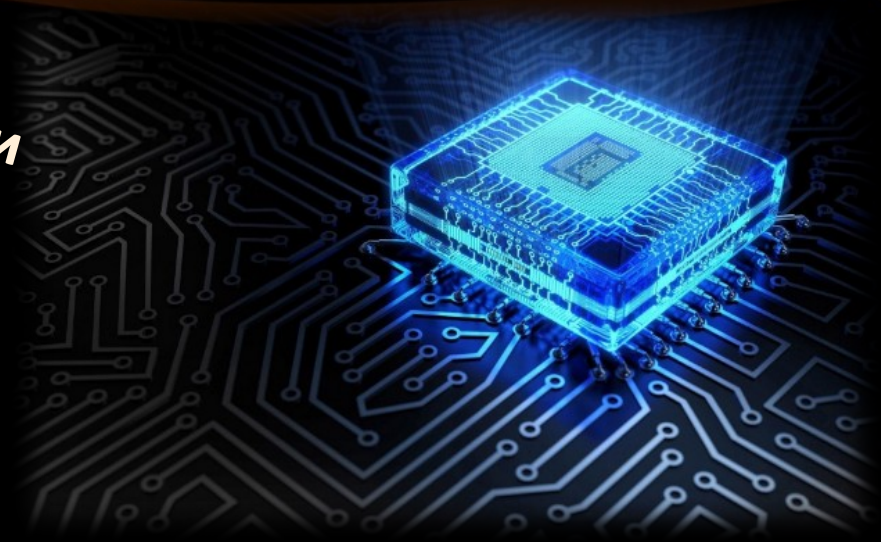
Организация на компютърната система



Инж. Венцеслав Кочанов

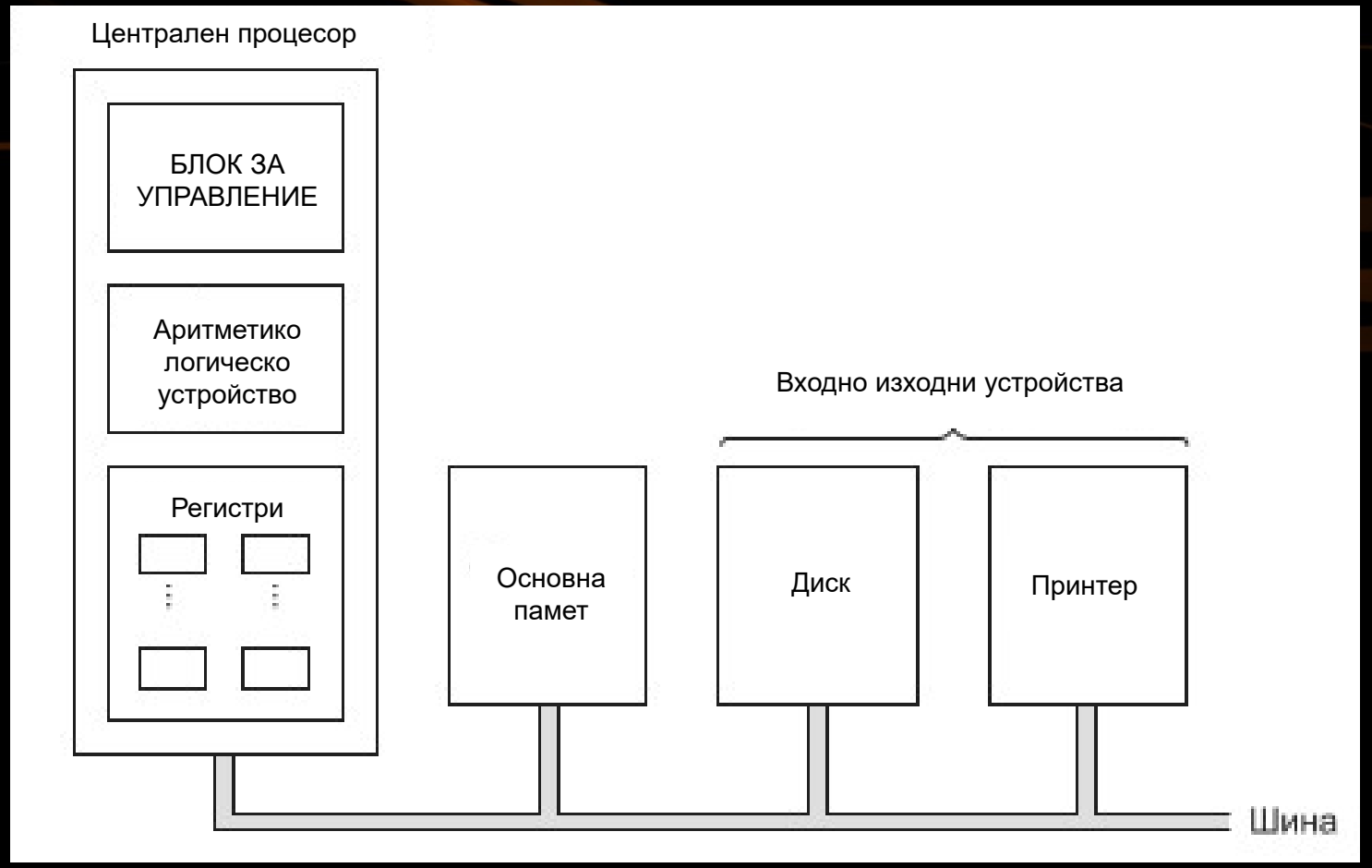
Организация на компютърните системи

Цифровият компютър се състои от взаимосвързани процесори, модули памет и входно/изходни устройства. Това са процесор, памет и входно/изходни устройства.

1.Процесор[10]

На фиг. Фигура 1.1 показва структурата на конвенционален компютър с шинна организация. Централният процесор е мозъкът на компютъра. Неговата задача е да изпълнява програми, намиращи се в основната памет. За да направи това, той извиква команди от паметта, определя техния тип и след това ги изпълнява една след друга. Компонентите са свързани с шина, която представлява набор от проводници, свързани паралелно за предаване на адреси, данни и контролни сигнали.

Шините могат да бъдат външни (свързващи процесора с памет и I/O устройства) и вътрешни. Съвременният компютър използва няколко шини.



Фиг. 1.1. Схема на компютър с един централен процесор и две входно-изходни устройства

Процесорът се състои от няколко части. ***Блокът за управление*** е отговорен за извикването на команди от паметта и определянето на техния тип. ***Аритметичното логическо устройство*** изпълнява аритметични операции (като събиране) и логически операции (като логическо И).

Регистрите са специални клетки на паметта, физически разположени вътре в процесора. За разлика от RAM, която изисква адресна шина за достъп до данни, процесорът има директен достъп до регистрите. Това значително ускорява работата с данни.

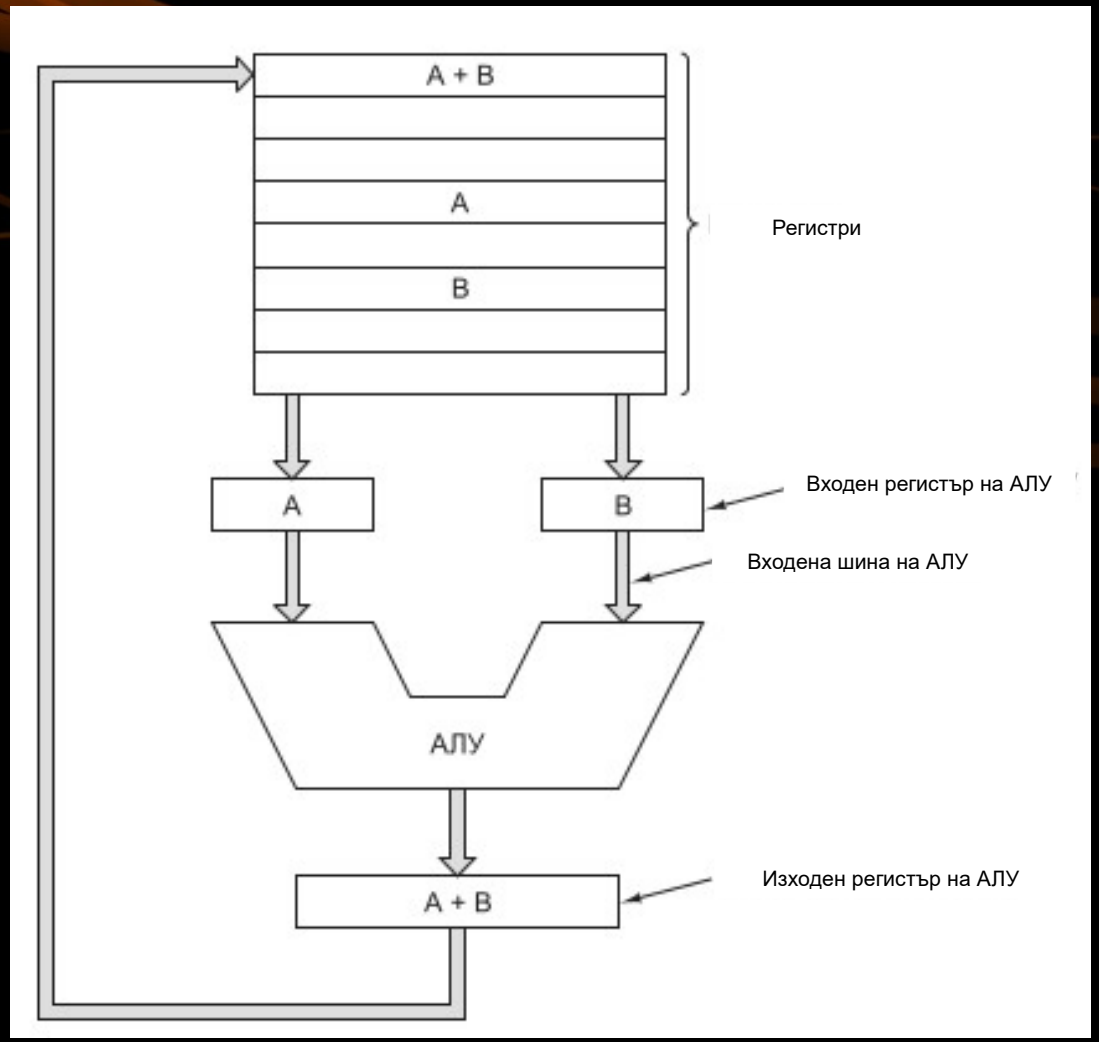
Вътре в централния процесор има малка, бърза памет за съхраняване на междинни резултати и някои контролни команди. Тази памет се състои от няколко регистъра, всеки от които изпълнява определена функция. Обикновено размерът на всички регистри е еднакъв.

Всеки регистър съдържа едно число, чиято горна граница зависи от размера на регистъра. Операциите за четене и запис на регистри са много бързи, защото те се намират вътре в процесора. Най-важният регистър е програмният брояч, който показва коя инструкция след коя следва. Името "програмен брояч" е малко неправилно , защото не брои нищо, но като термин се използва навсякъде.
Програмният брояч (PC) е специален регистър, присъстващ в компютрите, който съхранява адреса на следващата инструкция, която трябва да бъде изпълнена. Той се актуализира по време на цикъла на извличане на инструкции, когато процесорът извлича следващата инструкция от основната памет.
Друг регистър е и командния регистър, който съдържа текущо изпълняваната команда. Повечето компютри имат други регистри, някои от които са многофункционални, други служат само за конкретни цели. Други регистрите се използват от операционната система за управление на компютъра.

1.1. CPU устройство

Вътрешната структура на пътя на данните на типичен процесор на фон Нойман е илюстрирана на фиг. 1.2. Тази част се нарича път за данни и се състои от регистрите (обикновено от 1 до 32), ALU (аритметично логическо устройство) и няколко шини, свързващи частите. Съдържанието на регистрите се подава към входните регистри на ALU, които са показани на фиг. 1.2 са обозначени с A и B. Те съдържат входните данни на ALU, докато ALU извършва изчисления. ALU извършва събиране, изваждане и други прости операции върху входните данни и поставя резултата в изходен регистър. Съдържанието на този изходен регистър може да бъде записано обратно в един от регистрите или съхранено в паметта, ако е необходимо. Не всички архитектури имат A, B и изходни регистри.

На фиг. Фигура 1.2 показва операцията за добавяне, но ALU може да изпълнява други операции.



Фиг. 1.2. Път на данни на конвенционална машина на фон Нойман

Повечето инструкции могат да бъдат разделени на две групи: инструкции от регистър към памет и инструкции от регистър към регистър. Първият тип инструкция извиква думи от паметта, поставя ги в регистри, където се използват като вход към ALU (думите са елементи от данни, които се движат между паметта и регистрите). Една дума може да бъде цяло число. Други инструкции от този тип връщат регистрите обратно в паметта. Вторият тип инструкция извиква два операнда от регистри, поставя ги във входните регистри на ALU, извършва някаква аритметична или логическа операция върху тях и прехвърля резултата обратно в един от регистрите. Този процес се нарича цикъл на пътя на данните. До известна степен той определя какво може да направи машината. Съвременните компютри са оборудвани с няколко ALU, които работят паралелно и са специализирани в различни функции. Колкото по-бързи са циклите на пътя на данните, толкова по-бързо работи компютърът.

2. Регистри. Видове и структура

Процесорните регистри представляват блокът вътрешни регистри, който е един от съставящите основни блокове на централния процесор. По предназначение вътрешните регистри могат да бъдат разделени на две основни групи: регистри за данни и регистри за адреси. От своя страна регистрите за адреси се делят на две подгрупи: регистри указатели (SP, BP, IP) и индексни регистри (SI, DI).

По видове различаваме следните видове вътрешни регистри в основния блок:
1. Регистър файл: регистър за съхранение на файл. Файл наричаме съвкупност от данни с общо предназначение и тип, която притежава три атрибута: път, име и разширение.

2. Указател на стека (SP- Stack Pointer: stack-магазин), респективно ESP, представлява 16 битов регистър (SP), разширяем до 32 бита (ESP) на съвременните процесори, съдържащ адреса на следващата свободна клетка от стека. Той - показва (указва) началото на адресния стек и на стека за данни, които се намират в паметта и са необходими при работа с подпрограми. Стекът има организация „пръв влязъл-пръв излязъл”(FIFO), с достъп четене/запис. Това позволява съхранение на данните по време на прекъсванията и при изпълнение на подпрограми при тези прекъсвания. При запис на всеки нов байт в стека, SP автоматично се намалява, докато при четене SP се увеличава. Указателят на базата (BP-Base Pointer), респективно EBP, се използва както регистъра за данни BX за пресмятане на адрес при специални видове адресиране, свързан с регистър за стеков сегмент (SS).

3. Индексни регистри: Те включват:

3.1.Регистър за първични индекси (SI -Source Index), респ. ESI и Регистър за целеви индекси (DI -destination index), респ. EDI. Тези индексни регистри служат за пресмятане на адрес при инструкции за низове, ако например низ от знаци трябва да бъде преместен вътре в паметта. Тези регистри са ориентирани по X и по Y. SI по X се използва за индексна адресация, като подава 16 битова индексна стойност, която се сумира с 8-битово отместване в инструкцията, за да се изчисли действителния адрес. Този регистър може да бъде използван и като брояч или регистър за временно съхранение. SI по Y, подобно на този по X, се използва за индексна адресация. Неговото използване изисква един допълнителен байт в машинния код на инструкцията, тъй като всички инструкции с SI по Y са с двубайтови кодове.

3.2. Указателят на инструкциите (IP -Instruction Pointer), респ. EIP, показва адресите на следващите, наредени в опашка инструкции. Това отличава този регистър от брояча на инструкции (респ. от указателя на инструкциите в по-старите микропроцесори), който брояч постоянно показва следващата инструкция за изпълнение. Само при инструкция за преход, респ. за разклонение в програмата, указателят на инструкциите показва адреса на следващата за обработка инструкция.

4. Сегментни регистри: Това са два допълнителни сегментни регистъра:

Флагов сегмент (FS - Flag-Segment)
Кодов сегмент (CS - Code-Segment).

При тези допълнителни сегментни регистри, фирмата Intel е избрала следващите след „E” букви от латинската азбука. При този начин всеки адрес в паметта се получава чрез сумиране на един базов адрес и съответното отместване - Offset. Базовият адрес се запомня в един сегментен регистър, а отместването се намира в един от указателите или индексните регистри - например в указателя на инструкциите, ако дадена инструкция трябва да бъде прочетена от паметта. При директен достъп до данните в паметта, това отместване се задава при самата инструкция.

Сегментните регистри са четири типа: CS -кодов сегмент; DS - сегмент за данни; SS -стеков сегмент и ES - допълнителен сегмент за данни. Съществуването предимство на образуването на сегменти е възможността за преместване на програмите и данните в паметта. Това свойство се нарича преместваемост - relocatability.

5. Регистри за данни: Това са четири 16-разрядни регистъра: AX, BX, CX и DX, респективно четирите 32-разрядни регистъра при процесорите на фирмата Intel: EAX, EBX, ECX и EDX . тези 32-разрядни регистри са предназначени преди всичко за съхраняване на данните. Тъй като често се работи с еднобайтов операнд, 16-разрядните регистри са разделени на два 8-разрядни: старши - H-Higher и младши - L-Lower, регистри. Например регистърът AX се състои от AH и AL. Заедно с предназначението си като регистри за данни те изпълняват още и специални функции:

5.1 Регистърът AX - респ. EAX, наричан също така акумулатор, е централен регистър за умножение и деление. За извеждане към входно-изходен канал се използват регистрите AL, AX или EAX;

5.2. Регистърът BX, наричан още базов регистър (Base Register) е необходим за пресмятане на адреса при специален вид адресиране;

5.3 Регистърът CX, наричан също броячен регистър (Count Register), служи като броячен регистър при цикли, в инструкции за преместване надясно, респ. наляво, както и при повторяеми инструкции за низове. При последните може да се използва и регистърът ECX;

5.4 Регистърът DX, респ. EDX, наричан също регистър за данни (Data Register), се използва за пресмятане на адреса при специален вид адресиране. Тези регистри DX, свързани с AX (респ. EDX, свързан и с EAX), се използват при умножение, респ. при деление, както беше посочено по-горе при AX или EAX.

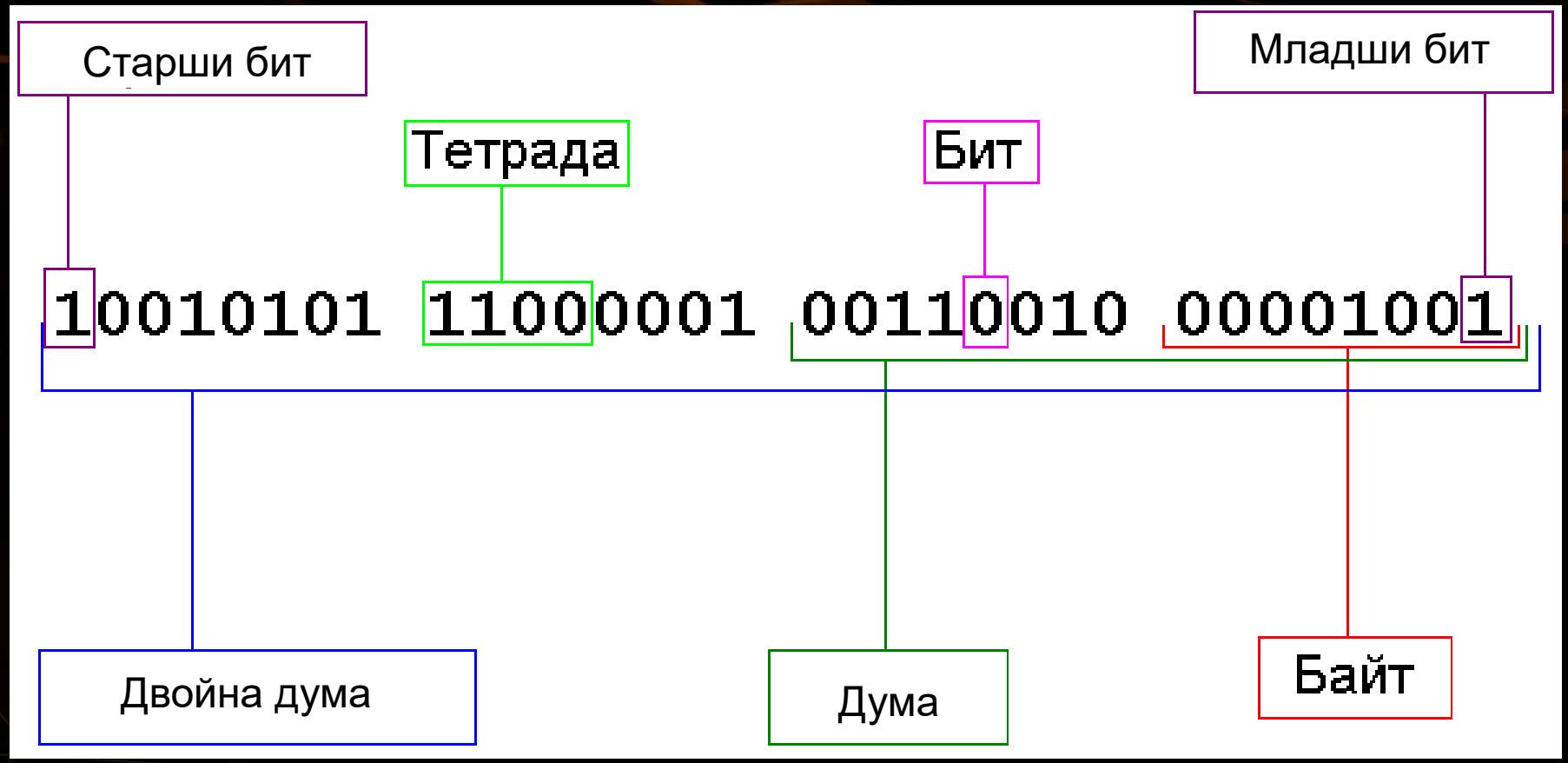
6. Програмен брояч(PC -Program Counter): Това е най-често 16-разряден регистър, който съдържа адреса на следващата инструкция, чието изпълнение предстои.
7. Регистър за код на условията (CCR -Code Condition Register). Това е най-често 8-разряден регистър, в който всеки бит указва определено условие от резултата на току-що изпълнената инструкция. Тези битове могат да бъдат тествани индивидуално от програмата и в резултат от тези тестове могат да бъдат предприети различни действия.
8. Дешифратор на инструкции: Този регистър дешифрира кода на операцията (КОП) и определя коя е заявената за изпълнение операция. Това представлява обработка на текущата инструкция.

Изпълнение на команди

Централният процесор изпълнява всяка команда на няколко стъпки. Той прави следното:

1. Извиква следващата инструкция от паметта и я прехвърля в регистъра на инструкциите.
2. Променя позицията на програмния брояч, който след това сочи към следващата команда.
3. Определя типа на извиканата команда.
4. Ако командата използва дума от паметта, определя къде се намира тази дума.
5. Прехвърля думата, ако е необходимо, в регистър на процесора.
6. Изпълнява командата.
7. Преминава на стъпка 1, за да започне изпълнението на следващата команда.

Забележка: Един байт е 256 комбинации на числа от 0 до 255. На долната фигура е показано визуално представяне на двоично число (двойна дума).



3. Микропроцесорни системни шини и цикли на обмен

Обменът на информация в микропроцесорните системи се извършва в цикли на обмен на информация. Цикълът на обмен на информация се разбира като интервал от време, през който се извършва една елементарна операция на обмен в шината. Например прехвърляне на код на данни от процесора към паметта или прехвърляне на код на данни от входно/изходно устройство към процесора. Няколко кода на данни, дори цял масив от данни, също могат да бъдат предадени в рамките на един цикъл, но това е по-рядко.

Циклите на обмен на информация се разделят на два основни вида:

1. Цикъл на запис (извеждане), в който процесорът записва (извежда) информация;
2. Цикъл на четене (въвеждане), в който процесорът чете (въвежда) информация.

Някои микропроцесорни системи също имат цикъл четене-модифициране-запис или вход-пауза-изход. В тези цикли процесорът първо чете информация от паметта или I/O устройство, след което по някакъв начин я трансформира и я записва отново на същия адрес.
Например по този начин процесорът може да чете код от клетка с памет, да го увеличава с единица и да го записва обратно в същата клетка с памет. Наличието или отсъствието на този тип цикъл е свързано с характеристиките на използвания процесор.
Специално място заемат циклите на директен достъп до паметта (ако в системата е осигурен режим DMA) и циклите на искане и предоставяне на прекъсвания (ако има прекъсвания в системата).
По време на всеки цикъл устройствата, участващи в обмена на информация, предават информация и управляващи сигнали един на друг в строго установен ред или, както се казва, в съответствие с приетия протокол за обмен на информация.

Микропроцесорни системни шини

Шината за данни е основната шина, за която е създадена цялата система.

Броят на неговите битове (комуникационни линии) определя скоростта и ефективността на обмена на информация, както и максимално възможния брой команди.

Шината за данни винаги е двупосочна, тъй като включва пренос на информация в двете посоки. Най-често срещаният тип изходно стъпало за линии на тази шина е изход с три състояния. Обикновено шината за данни има 8, 16, 32 или 64 бита. Ширината на шината за данни определя ширината на цялата шина.

Адресната шина е втората по важност шина, която определя максималната възможна сложност на микропроцесорната система, т.е. допустимото количество памет и следователно максималния възможен размер на програмата и максималното възможно количество съхранявани данни. Броят на адресите, предоставени от адресната шина, се определя като 2N, където N е броят на битовете. Например 16-битова адресна шина осигурява 65 536 адреса. Ширината на адресната шина обикновено е кратна на 4 и може да достигне 32 или дори 64. Адресната шина може да бъде еднопосочна (когато шината винаги е управлява само процесора) или двупосочен (когато процесорът може временно да прехвърли управлението на магистралата на друго устройство, например DMA контролер). Най-често използваните видове изходни етапи са три състояния или конвенционален TTL (две състояния).

Контролната шина е спомагателна шина, управляващите сигнали на която определят вида на текущия цикъл и фиксират времената, съответстващи на различни части или етапи от цикъла. Освен това управляващите сигнали осигуряват координация на работата на процесора (или друг главен магистрален, главен, master) с работата на паметта или входно-изходното устройство (изпълнително устройство, подчинено).

4. Обиколка на компютърните системи

Компютърната система е съвкупност от хардуерни и софтуерни компоненти, които работят заедно, за да изпълняват компютърни програми. Конкретните реализации на системите се променят с времето, но основните концепции не.
Всички системи имат подобни хардуерни и софтуерни компоненти, които изпълняват подобни функции.
В техния класически текст на почти всички езици за програмиране използваме програмата hello world, като в нашият случай ще използвам езика C. Кодът е показан по-долу:

```
1 #include <stdio.h>
2
3 int main()
4 {
5       printf("hello, world\n");
6 }
```

Въпреки че hello е много проста програма, всяка голяма част от системата трябва да работи съгласувано, за да може да работи докрай. Целта ние тук да разбереме какво се случва и защо, когато стартирате hello на нашата система. Ще започнем нашето изследване на системите, като проследим живота на програмата hello от момента, в който е създадена, отпечата простото си съобщение и се прекрати. Докато проследяваме жизнения цикъл на програмата, ще представим накратко основните концепции,

Информацията е битове в контекста

Нашата програма hello започва живота си като програма източник (или файл източник), който програмистът създава с редактор и записва в текстов файл, наречен hello.c. Изходната програма е поредица от битове, всеки със стойност от 0 или 1, организирани в 8-битови части, наречени байтове. Всеки байт представлява някакъв текстов знак в програмата. Повечето съвременни системи представят текстови знаци, като използват стандарта ASCII, който представя всеки знак с уникална целочислена стойност с размер на байт. Например Фигура 4.1 показва ASCII представянето на програмата hello.c.

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46

h	>	\n	\n	i	n	t	<sp>	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123

\n	<sp>	<sp>	<sp>	<sp>	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108

l	o	,	<sp>	w	o	r	l	d	\	n	"	)	;	\n	}
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

Фигура 4.1.ASCII текстово представяне на hello.c.

Програмата hello.c се съхранява във файл като поредица от байтове. Всеки байт има цяло число, което съответства на някакъв знак. Например, първият байт има цяло число 35, което съответства на символът '#'. Вторият байт има цяло число 105, което съответства на знака „i“ и т.н. Забележете, че всеки текстов ред завършва с невидимия знак за нов ред '\n', който е представен от целочислената стойност 10. Файлове като hello.c, които се състоят изключително от ASCII знаци, са известни като текстови файлове. Всички други файлове са известни като двоични файлове. Представянето на hello.c илюстрира фундаментална идея: Цялата информация в системата - включително дискови файлове, програми, съхранени в паметта, потребителски данни, съхранени в паметта, и данни, прехвърлени през мрежа - се представя като куп битове. Единственото нещо, което отличава различните обекти с данни, е контекстът, в който ги разглеждаме.

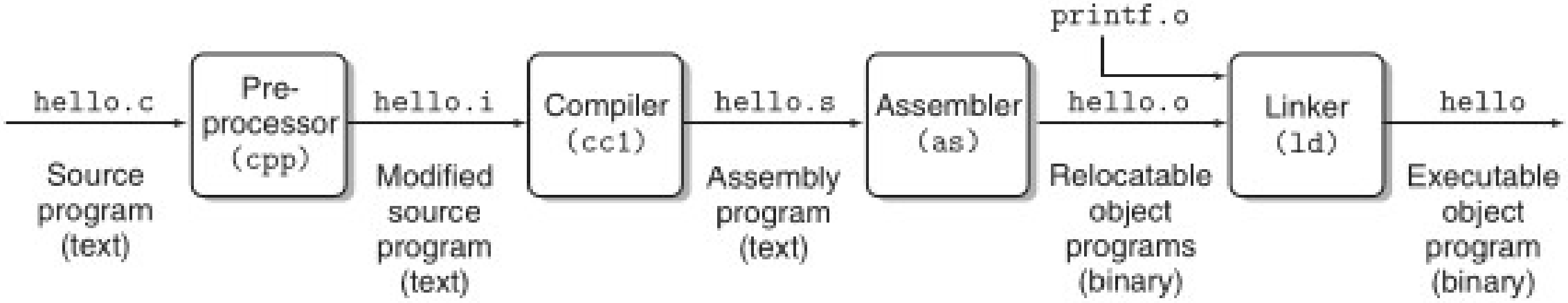
4.1. Програмите се превеждат от други програми в различни форми

Програмата hello започва живота си като С програма от високо ниво, защото може да бъде прочетена и разбрана от хората в тази форма. Въпреки това, за да стартирате hello.c в системата, отделните С изрази трябва да бъдат преведени от други програми в последователност от инструкции на машинен език на ниско ниво. След това тези инструкции се пакетират във форма, наречена изпълнима обектна програма, и се съхраняват като двоичен дисков файл. Обектните програми също се наричат изпълними обектни файлове.

В Unix система преводът от изходен файл към обектен файл се извършва от драйвер на компилатор:

linux> gcc -o hello hello.c

Тук драйверът на компилатора на GCC чете изходния файл hello.c и го превежда в изпълним обектен файл hello. Преводът се извършва в последователност от четири фази, показана на фигура 4.2. Програмите, които изпълняват четирите фази (препроцесор, компилатор, асемблер и линкер), са известни като система за компилиране.



Фигура 4.2 Системата за компилиране.

Фаза на предварителна обработка.
Препроцесорът (cpp) модифицира оригиналната C програма според директиви, които започват със знака "#". Например командата #include <stdio.h> в ред 1 на hello.c казва на препроцесора да прочете съдържанието на системния заглавен файл stdio.h и да го вмъкне директно в текста на програмата. Резултатът е друга C програма, обикновено със суфикс .i.

Фаза на компилация.
Компилаторът (cc1) превежда текстовия файл hello.i в текстовия файл hello.s, който съдържа *програма на асемблиращ език*. Тази програма включва следната дефиниция на функция main:

```
1  main:
2          subq    $8, %rsp
3          movl    $.LC0, %edi
4          call    puts
5          movl    $0, %eax
6          addq    $8, %rsp
7          ret
```

Този код изглежда е за x86 архитектура, е написан на асемблер и е генериран от компилатор.
Ето разбивка на това, което прави всеки ред:

main:: Това маркира началото на основната функция. Това е етикет, указващ началото на кодовия сегмент за основната функция.

subq $8, %rsp: Тази инструкция изважда 8 байта от указателя на стека %rsp. Това обикновено означава, че в стека се разпределя място за локални променливи или други данни.

movl $.LC0, %edi: Това премества адреса на мястото в паметта, обозначено с .LC0, в регистъра %edi. .LC0 вероятно съдържа низов литерал.

call puts: Това извиква функцията puts, която се използва за извеждане на низ към стандартния изход. Низът за отпечатване се определя от съдържанието на %edi, което беше зададено в предишната инструкция.

movl $0, %eax: Това премества стойността 0 в регистъра %eax. Това обикновено се използва за обозначаване на успешно изпълнение на програмата (връщането на 0 обикновено означава успех в C програмирането).

addq $8, %rsp: Това добавя 8 байта обратно към указателя на стека %rsp, ефективно освобождавайки пространството, което е разпределено в ред 2. Това е за почистване на стека преди връщане.

ret: Това е инструкцията за връщане, която прекратява функцията и връща контролния поток към извикващата функция.

Отделя се малко място в стека, отпечатва низ, задава връщаната стойност на 0, почиства стека и се връща от функцията.
Всеки от редовете 2–7 в тази дефиниция описва една инструкция на машинен език от ниско ниво в текстова форма. Асемблерният език е полезен, защото осигурява общ изходен език за различни компилатори за различни езици на високо ниво. Например C компилаторите и Fortran компилаторите генерират изходни файлове на един и същи асемблер.

Фаза на асемблиране.

След това асемблерът (as) превежда hello.s в инструкции на машинен език, пакетира ги във форма, известна като *преместваема обектна програма,* и съхранява резултата в обектния файл hello.o. Този файл е двоичен файл, съдържащ 17 байта за кодиране на инструкциите за функция main. Ако разглеждаме hello.o с текстов редактор, ще изглежда като безсмислица.

Фаза на свързване.

Забележете, че нашата hello програма извиква функцията printf, която е част от стандартната C библиотека, предоставена от всеки C компилатор. Функцията printf се намира в отделен предварително компилиран обектен файл, наречен printf.o, който по някакъв начин трябва да бъде обединен с нашата програма hello.o. Линкерът (ld) обработва това сливане.

Резултатът е hello файл, който е изпълним обектен файл (или просто изпълним), който е готов да бъде зареден в паметта и изпълнен от системата.

В този момент нашата програма източник hello.c е преведена от системата за компилиране в изпълним обектен файл, наречен hello, който се съхранява на диска. За да стартираме изпълнимия файл на Unix система, въвеждаме името му в приложна програма, известна като shell:

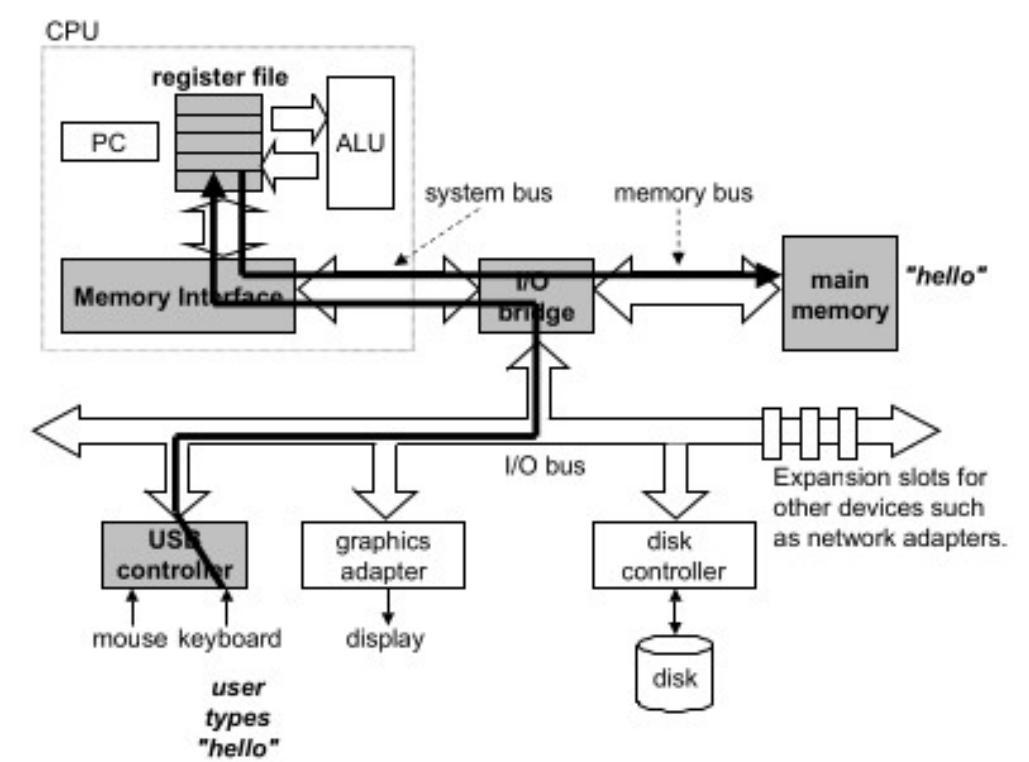
```
linux> ./hello
hello, world
linux>
```

Обвивката /shell/ е интерпретатор на команден ред, който отпечатва подкана, изчаква да въведете командния ред и след това изпълнява командата. Ако първата дума от командния ред не съответства на вградена команда в shell, тогава shell приема, че това е името на изпълним файл, който трябва да зареди и изпълни. Така че в този случай обвивката зарежда и изпълнява програмата hello и след това чака тя да приключи. Програмата hello отпечатва своето съобщение на екрана и след това прекратява работа. След това shell отпечатва подкана и изчаква следващия команден ред за въвеждане.

4.2. Изпълнение на програмата hello

Имайки предвид този прост изглед на хардуерната организация и работа на системата, можем да започнем да разбираме какво се случва, когато стартираме нашата примерна програма.

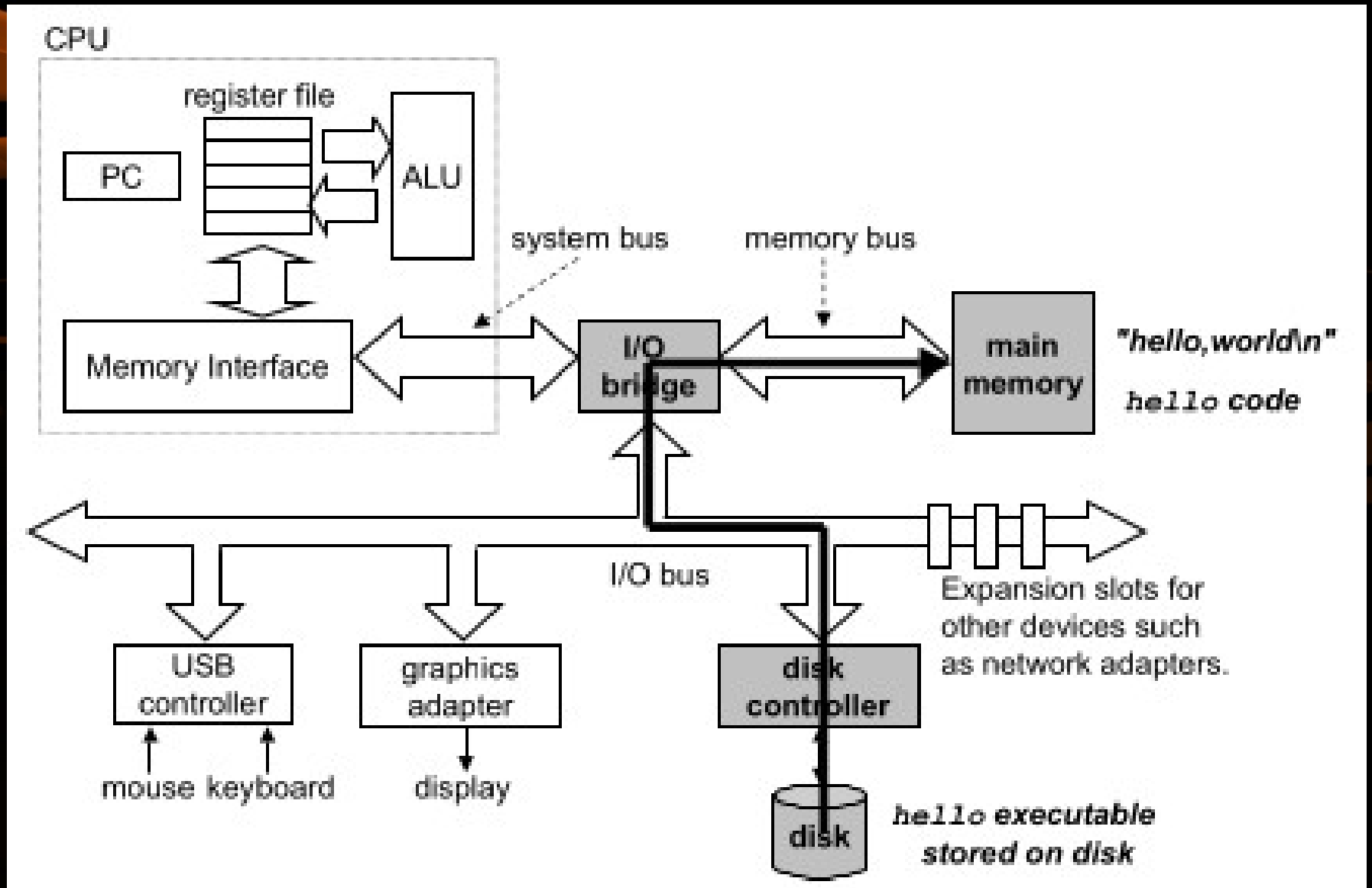
Първоначално програмата shell изпълнява своите инструкции, чакайки да напишем команда. Докато въвеждаме символите hello на клавиатурата, програмата на shell чете всеки един в регистър и след това го съхранява в паметта, както е показано на Фигура 4.2.



Фигура 4.2 Четене на командата hello от клавиатурата.

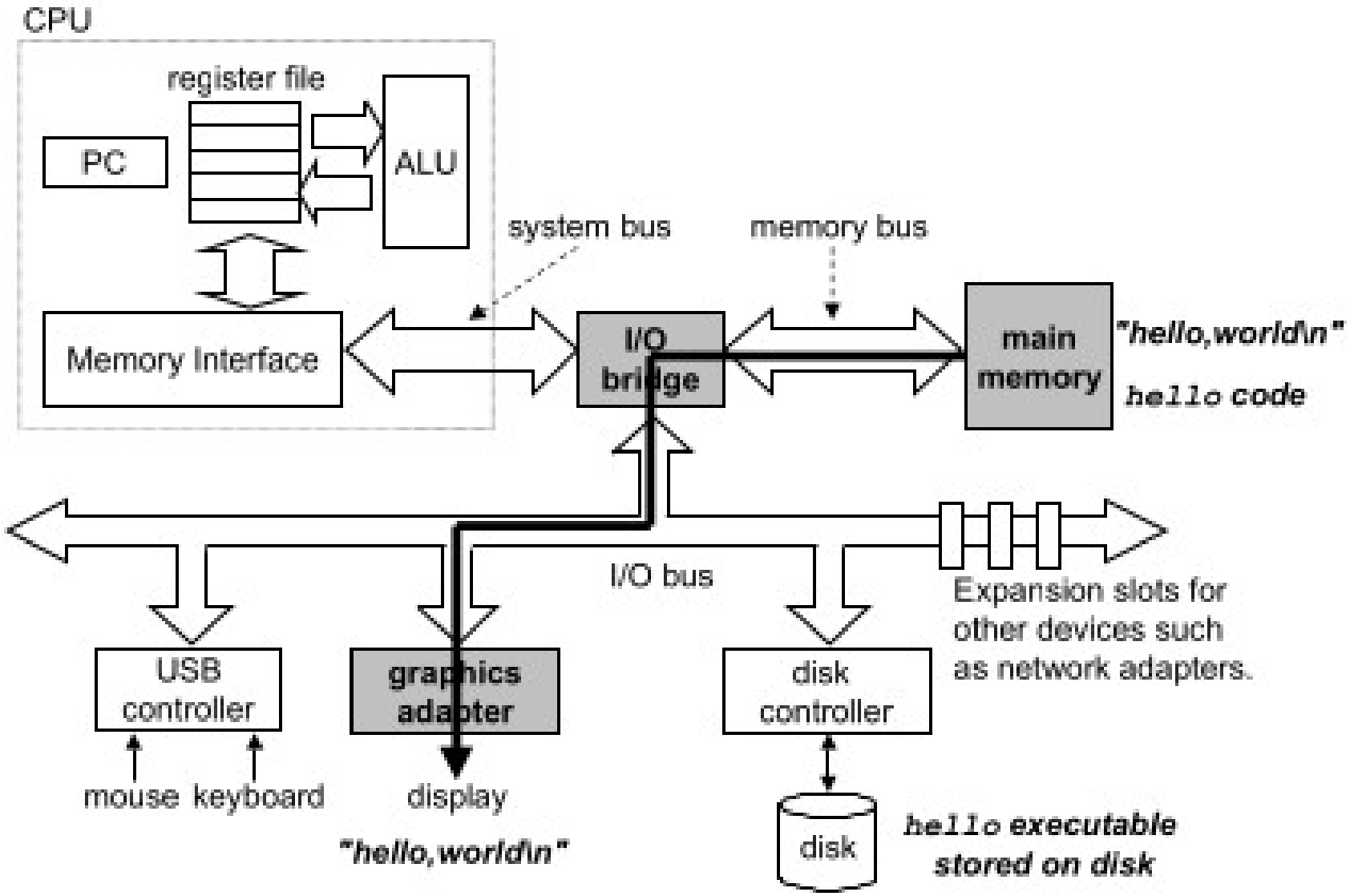
Когато натиснем клавиша enter на клавиатурата, обвивката знае, че сме приключили с въвеждането на командата. След това обвивката зарежда изпълнимия hello файл чрез изпълнение на последователност от инструкции, които копират кода и данните в hello обектния файл от диска в основната памет. Данните включват низа от знаци hello, world\n, който в крайна сметка ще бъде отпечатан.

Използвайки техника, известна като директен достъп до паметта, данните пътуват директно от диска към основната памет, без да преминават през процесора. Тази стъпка е показана на фигура 4.3.



Фигура 4.3 Зареждане на изпълнимия файл от диска в основната памет.

След като кодът и данните в обектния файл hello се заредят в паметта, проце-сорът започва да изпълнява инструкциите на машинния език в основната рутина на програмата hello. Тези инструкции копират байтовете в низа hello, world\n от паметта в регистърния файл и оттам в устройството за показване, където се показват на екрана. Тази стъпка е показана на фигура 4.3.



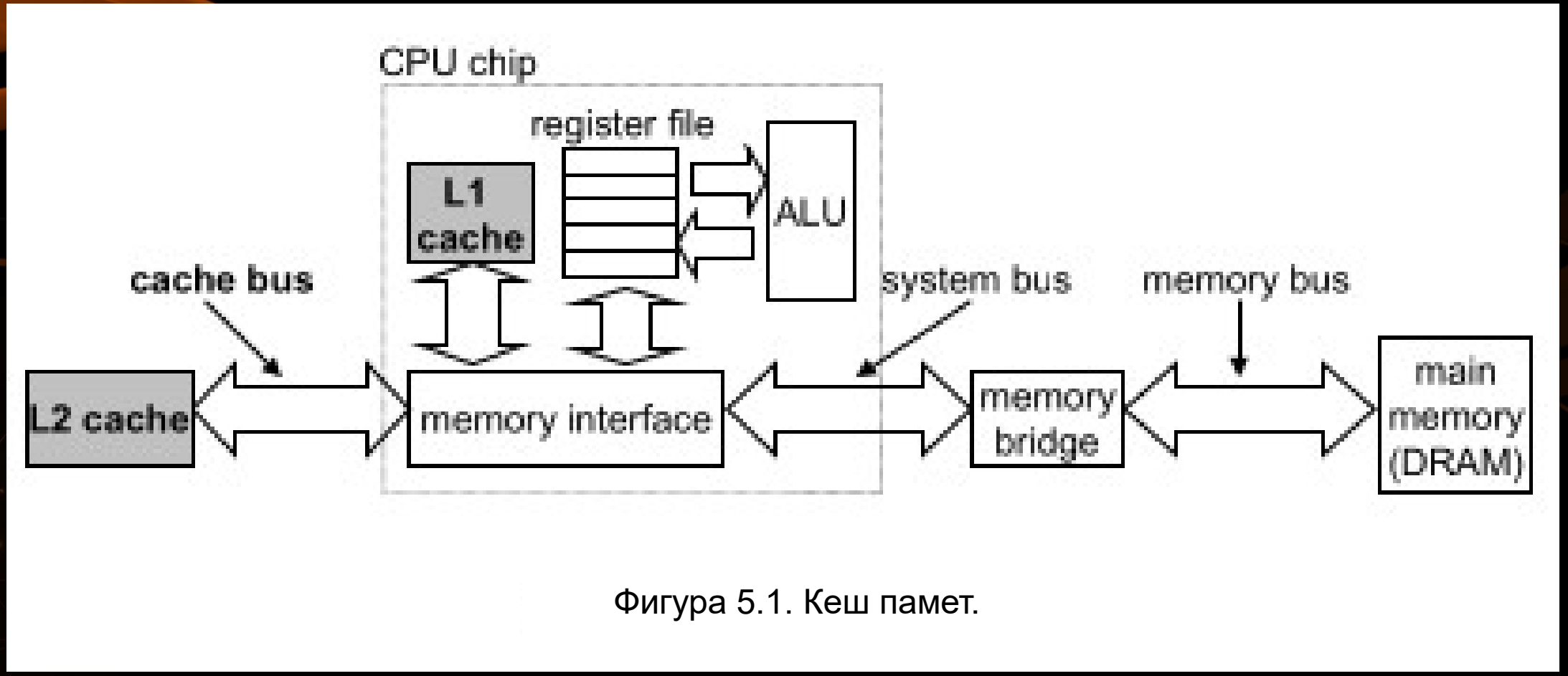
Фигура 4.3 Записване на изходния низ от паметта на дисплея.

5. Кеш памет

Машинните инструкции в програмата Hello първоначално се съхраняват на диск. Когато програмата се зареди, те се копират в основната памет. Докато процесорът изпълнява програмата, инструкциите се копират от основната памет в процесора. По подобен начин низът с данни hello,world\n, първоначално от диска, се копира в основната памет и след това се копира от основната памет в устройството за показване. От гледна точка на програмиста, голяма част от това копиране е излишно, което забавя „истинската работа“ на програмата. По този начин основната цел на системните дизайнери е да направят тези копиращи операции възможно най-бързи.

Поради физическите закони по-големите устройства за съхранение са по-бавни от по-малките устройства за съхранение. А по-бързите устройства са по-скъпи за изграждане от техните колеги. Например, дисковото устройство на типична система може да е 1000 пъти по-голямо от основната памет, но може да отнеме на процесора 10 000 000 пъти повече време, за да прочете дума от диска, отколкото от паметта. По същия начин типичният регистърен файл съхранява само няколкостотин байта информация, за разлика от милиарди байтове в основната памет. Процесорът обаче може да чете данни от регистърния файл почти 100 пъти по-бързо, отколкото от паметта. Още по-неприятно е, че с напредването на полупроводниковата технология през годините тази разлика между процесор и памет продължава да се увеличава. По-лесно и евтино е да се направи процесорите работят по-бързо, отколкото е, за да може основната памет да работи по-бързо.

За да се справят с празнината между процесор и памет, системните дизайнери включват по-малки, по-бързи устройства за съхранение, наречени кеш памети (или просто кешове), които служат като временни зони за съхранение на информация, от която процесорът вероятно ще се нуждае в близко бъдеще. Фигура 5.1 показва кеш паметта в типична система. Кеш L1 на процесорния чип съдържа десетки хиляди байтове и може да бъде достъпен почти толкова бързо, колкото регистрационния файл.

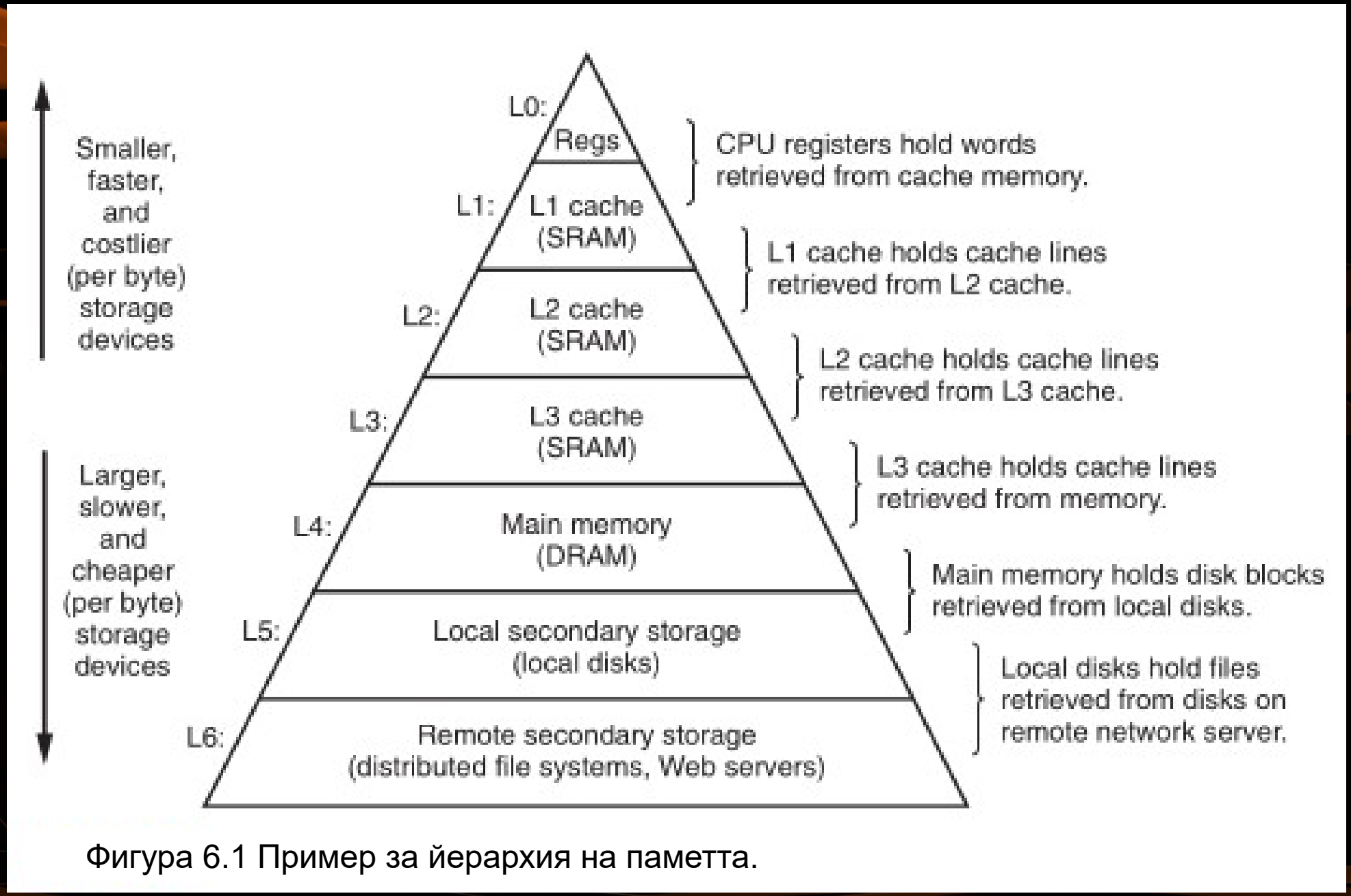


Фигура 5.1. Кеш памет.

По-голям L2 кеш със стотици хиляди до милиони байтове е свързан към процесора чрез специална шина. Може да отнеме 5 пъти повече време на процесора за достъп до L2 кеша отколкото L1 кеша, но това все пак е 5 до 10 пъти по-бързо от достъпа до основната памет. Кешовете L1 и L2 са реализирани с хардуерна технология, известна като статична памет с произволен достъп (SRAM). По-новите и по-мощни системи дори имат три нива на кеш: L1, L2 и L3. Идеята зад кеширането е, че системата може да получи ефекта както на много голяма памет, така и на много бърза, като използва локалността, тенденцията програмите да имат достъп до данни и код в локализирани региони. Като настроим кешове за съхранение на данни, които е вероятно да бъдат достъпни често, можем да извършваме повечето операции с паметта използване на бързи кеши.

6. Йерархия на устройствата за съхранение

Идеята за вмъкване на по-малко, по-бързо устройство за съхранение (напр. кеш памет) между процесора и по-голямо, по-бавно устройство (напр. основна памет) се оказва обща идея. Всъщност устройствата за съхранение във всяка компютърна система са организирани като йерархия на паметта, подобна на фигура 6.1.



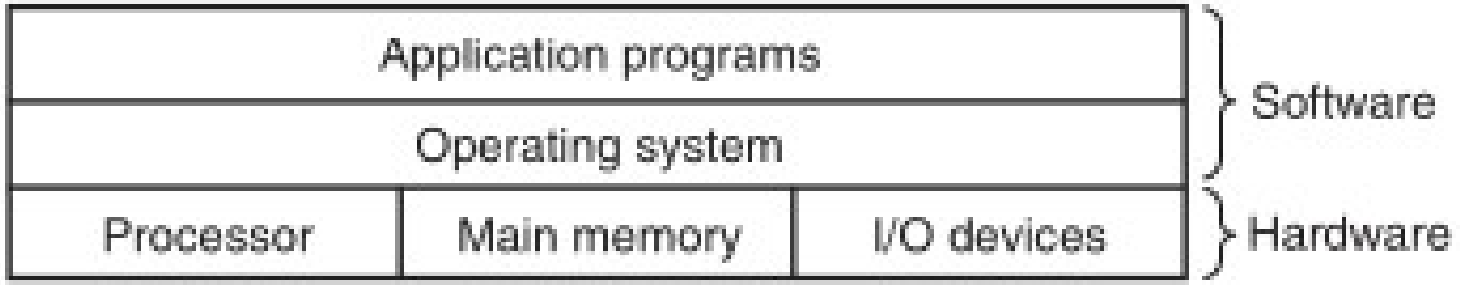
Фигура 6.1 Пример за йерархия на паметта.

Докато се придвижваме от върха на йерархията към дъното, устройствата стават по-бавни, по-големи и по-евтини на байт. Регистърният файл заема най-високото ниво в йерархията, което е известно като ниво 0 или L0. Показваме три нива на кеширане от L1 до L3, заемащи паметта нива на йерархия от 1 до 3.
Основната памет заема ниво 4 и т.н.
Основната идея на йерархията на паметта е, че съхранението на едно ниво служи като кеш за съхранение на следващото по-ниско ниво. По този начин регистрационният файл е кеш за L1 кеша. Кешовете L1 и L2 са кешове съответно за L2 и L3. Кешът L3 е кеш за основната памет, който е кеш за диска. В някои мрежови системи с разпределени файлови системи локалният диск служи като кеш за данни, съхранявани на дисковете на други системи.
Програмистите могат да използват знанията за различните кешове, за да подобрят производителността, така те могат да използват своето разбиране за цялата йерархия на паметта.

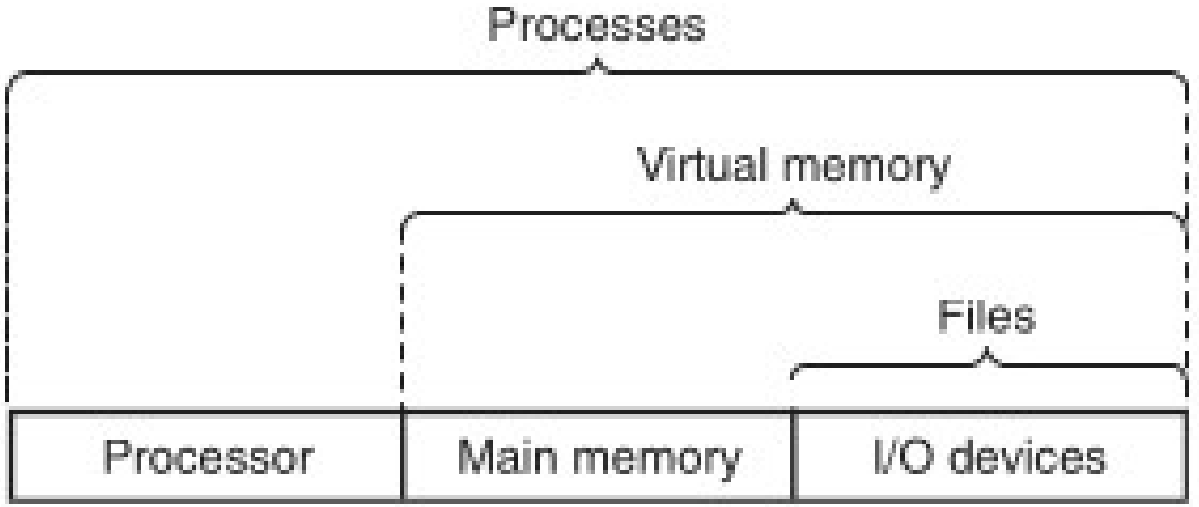
7. Операционната система управлява хардуера(p.22)

Да се върнем към нашия пример за hello. Когато обвивката зареди и изпълни програмата hello и когато програмата hello отпечата съобщението си, в този момент нито една програма няма директен достъп до клавиатурата, дисплея, диска или основната памет. Тук се разчита на услугите, предоставяни от операционната система. Можем да мислим за операционната система като слой от софтуер, разположен между приложната програма и хардуера, както е показано на фигура 7.1. Всички опити на приложната програма да манипулира хардуерът трябва да преминат през операционната система. Операционната система има две основни цели: (1) да защитава хардуера от злоупотреба от неработещи приложения и (2) да предоставя на приложенията прости и единни механизми за манипулиране на сложни и често изключително различни хардуерни устройства от ниско ниво.

Фигура 7.1 Послоен изглед на компютърна система.



Фигура 7.2 Абстракции, предоставени от операционна система.



Операционната система постига и двете цели чрез фундаменталните абстракции, показани на фигура 7.2: процеси, виртуална памет и файлове. Както подсказва тази фигура, файловете са абстракции за I/O устройства, виртуалната памет е абстракция както за основната памет, така и за дисковите I/O устройства, а процесите са абстракции за процесора, основната памет и I/O устройствата.

8. RISC и CISC технологии

RISC и CISC са два различни вида компютърни архитектури, които се използват за проектиране на микропроцесорите, които се намират в компютрите. Основната разлика между RISC и CISC е, че RISC (компютър с намален набор от инструкции) включва прости инструкции и отнема един цикъл, докато CISC (компютър с комплексен набор от инструкции) включва сложни инструкции и отнема множество цикъла.

Какво е RISC?

В RISC архитектурата наборът от инструкции на компютърната система е опростен, за да се намали времето за изпълнение. RISC архитектурата има малък набор от инструкции, които обикновено включват операции от регистър към регистър.

RISC архитектурата използва сравнително прост формат на инструкции, който е лесен за декодиране. Дължината на инструкциите може да бъде фиксирана и подравнена към границите на думите. RISC процесорите могат да изпълнят само една инструкция на тактов цикъл.

Ето някои важни характеристики на RISC процесора:

- RISC процесорът има няколко инструкции.
- RISC процесорът има няколко режима на адресиране.
- В RISC процесора всички операции се извършват в регистрите на процесора.
- RISC процесорът може да бъде с фиксирана дължина.
- RISC може да бъде твърдо свързан, а не микропрограмиран контрол.
- RISC се използва за изпълнение на инструкции в един цикъл.

• RISC процесорът има лесно декодируем формат на инструкции.
RISC архитектурите се характеризират с малък, прост набор от инструкции и високоефективен конвейер за изпълнение. Това позволява на RISC процесорите да изпълняват инструкции бързо, но също така означава, че те могат да изпълняват само ограничен брой задачи.

Какво е CISC?
CISC архитектурата се състои от сложен набор от инструкции. CISC процесорът има формат на инструкции с променлива дължина. В тази процесорна архитектура инструкциите, които изискват регистърни операнди, могат да заемат само два байта.

В CISC процесорна архитектура инструкциите, които изискват два адреса на паметта, могат да отнемат пет байта, за да съставят пълния код на инструкцията. Следователно, в CISC процесор, изпълнението на инструкции може да отнеме различен брой тактови цикли. CISC процесорът също така осигурява директно манипулиране на операндите, които се съхраняват в паметта.
Основната цел на процесорната архитектура на CISC е да поддържа една машинна инструкция за всеки израз, който е написан на език за програмиране от високо ниво.

Ето по - важните характеристики на CISC процесорната архитектура:

- CISC може да има формати на инструкции с променлива дължина.
- Той поддържа набор от голям брой инструкции, обикновено от 100 до 250 инструкции.

- Има голямо разнообразие от режими на адресиране, обикновено от 5 до 20 различни режима.
- CISC има някои инструкции, които изпълняват специализирани задачи и се използват рядко.

CISC архитектурите имат голям, сложен набор от инструкции и по-малко ефективен конвейер за изпълнение. Това позволява на CISC процесорите да изпълняват по-широк набор от задачи, но те не са толкова бързи, колкото RISC процесорите, когато изпълняват инструкции.

Въпроси?