به نام خدا

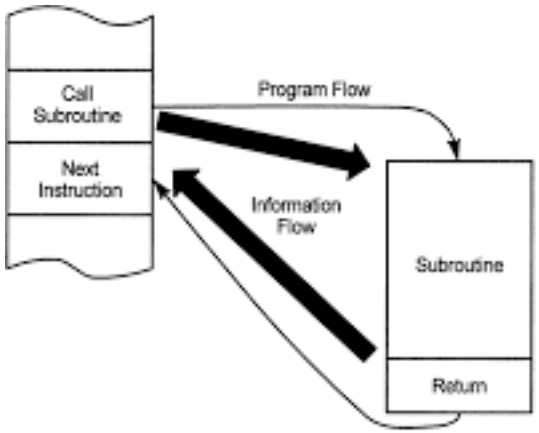
آشنایی با زبان اسمبلی AVR

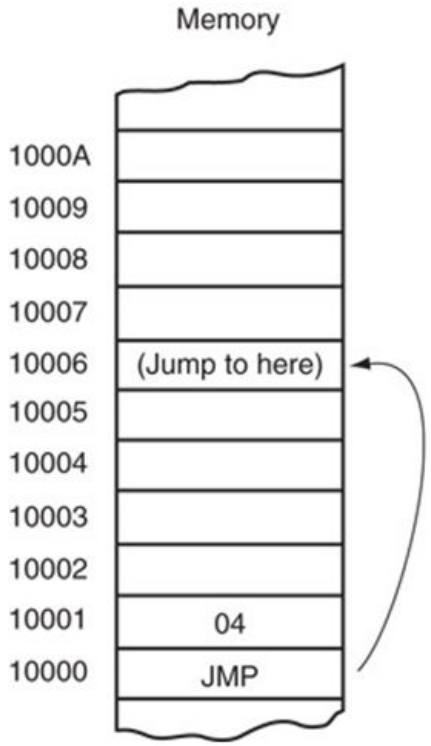
زیرروال‌ها

Dr. Aref Karimiafshar
A.karimiafshar@ec.iut.ac.ir

دستورات پرش

- In the sequence of instruction to be executed, it is often necessary to transfer program control to a different location.
- Instruction in AVR to achieve this:
  - Branches (JUMP)
  - CALL





اینو قبلا گفتیم
branches: این شامل دستوراش شرطی و غیر شرطی میشه که قبلا اینارو گفتیم
call: رو می خوایم الان بگیم

فراخوانی زیرروال (CALL)

- Is used to call a subroutine
- Subroutines
  - To perform tasks that need to be performed frequently
- Advantages
  - Makes a program more structured
  - Saving memory space

اتفاقی که اینجا رخ میده اینه که:
با فراخوانی call ما می ریم سراغ اجرای یک Subroutines
و Subroutines چیه؟ یک قطعه کدیه که یک کار خاصی رو برای ما انجام میده و این معمولا مکرر استفاده میشه مثلا یک تابع تاخیر که توی کارها زیاد ازش استفاده میشه
استفاده از دستور Call و Subroutines ها یکسری مزایایی واسه ما داره:
از جمله این که کمک میکنه کد ما ساختارمند بشه
و در حافظه کد ما صرفه جویی میشه
پس دستور مفیدی است با توجه به این مزایا

فراخوانی زیرروال (CALL)

- Four instructions for call subroutine:
  - CALL (long call)
  - RCALL (relative call)
  - ICALL (indirect call to Z)
  - EICALL (extended indirect call to Z)

- The choice of which one to use depends on the target address

دستور call رو به دسته تقسیم میکنیم: پس فراخوانیش با 4 دسته مختلف امکان پذیر است
به صورت عادی call به عنوان فراخوانی اصلی شناخته میشه و یک فراخوانی با پرش بلند توی فضای کد رو به ما میده
RCALL: به صورت نسبی پرش میکنه و یک مقدار دامنه پرش اون کوتاه تر است
ICALL:
EICALL: یک مقدار اون پرشی که توی ICALL داریم رو گسترش داده و یک پرش بلندتری داریم
چه زمانی از کدوم یکی از این دستورات استفاده بکنیم کاملا وابسته است به اون ادرس مقصد ما ینی اون Subroutines که ما می خوایم به اون پرش بکنیم در چه قسمتی از فضای برنامه ما قرار داره ینی نیازمند پرش کوتاه هستیم یا بلند پس بسته به نوع پرشی که داریم و اون جایی که اون Subroutines ما قرار گرفته یکی از این دستورات رو استفاده می کنیم

CALL – Long Call to a Subroutine

- Calls to a subroutine within the entire Program memory. The return address (to the instruction after the CALL) will be stored onto the Stack. The Stack Pointer uses a post-decrement scheme during CALL.

Operation:

(i) PC ← k Devices with 16-bit PC, 128KB Program memory maximum.

(ii) PC ← k Devices with 22-bit PC, 8MB Program memory maximum.

	Syntax:	Operands:	Program Counter:	Stack:
(i)	CALL k	0 ≤ k < 64K	PC ← k	STACK ← PC+2 SP ← SP-2, (2 bytes, 16 bits)
(ii)	CALL k	0 ≤ k < 4M	PC ← k	STACK ← PC+2 SP ← SP-3 (3 bytes, 22 bits)

CALL: با این دستور می تونیم در تمام فضای حافظه پرش بکنیم ینی هر جایی که Subroutines قرار گرفته می تونیم با دستور CALL به اون دستیابی پیدا بکنیم
اتفاقی که رخ میده اینه که با اجرای دستور CALL ادرس دستور بعدی میاد و توی استک قرار میگیره و حافظه استک یک بخشی از SRAM ما است
وقتی که ادرس بازگشت در استک قرار گرفت اون استک پوینتر ما که به انتهای حافظه استک داره اشاره میکنه یک واحد کاهش پیدا میکنه ینی به سمت پایین رشد خواهد کرد
از اونجایی که میکروکنترلرهای AVR بعضا PC هاشون 16 بیتی و بعضی هاشون 22 بیتی این باعث میشه دستور CALL ما به فضاهای متفاوتی دسترسی پیدا بکنه:
اگر از یک میکروکنترلی استفاده میکنیم که PC ان 16 بیتی است ما می تونیم در تمام اون 128KB که اون میکرو در اختیار ما قرار میده با استفاده از این دستور پرش بکنیم
و اگر از میکرویی استفاده میکنیم که PC اون 22 بیت داره ما می تونیم در 8KB فضای حافظه اون جابه جا بشیم
به هر حال بعد از اجرای دستور CALL ادرس اون خونه ینی K در PC قرار میگیره
و بسته به نوع میکرو این K می تونه فرق داشته باشه همون 16 بیت و 22 بیت که بالا گفتیم باعث میشه K ما متفاوت باشه
اتفاقی که بعدش می افته اینه که:
اگر در یک میکروی 16 بیتی هستیم استک پوینتر ما 2 واحد کم میشه و اگر در یک میکروی 22 بیتی هستیم استک پوینتر ما 3 واحد کم میشه

CALL – Long Call to a Subroutine

32-bit Opcode:

1001	010k	kkkk	111k
kkkk	kkkk	kkkk	kkkk

Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

Words 2 (4 bytes)

Cycles 4 devices with 16-bit PC

5 devices with 22-bit PC

اپکدش 32 بیتی است و 22 بیت اون می تونه متعلق به اپرند اون اجرا باشه

RCALL – Relative Call to Subroutine

- Relative call to an address within PC - 2K + 1 and PC + 2K (words). The return address (the instruction after the RCALL) is stored onto the Stack. For AVR microcontrollers with Program memory not exceeding 4K words (8KB) this instruction can address the entire memory from every address location. The Stack Pointer uses a post-decrement scheme during RCALL.

	Operation:	Comment:
(i)	PC ← PC + k + 1	Devices with 16-bit PC, 128KB Program memory maximum.
(ii)	PC ← PC + k + 1	Devices with 22-bit PC, 8MB Program memory maximum.

	Syntax:	Operands:	Program Counter:	Stack:
(i)	RCALL k	-2K ≤ k < 2K	PC ← PC + k + 1	STACK ← PC + 1 SP ← SP - 2 (2 bytes, 16 bits)
(ii)	RCALL k	-2K ≤ k < 2K	PC ← PC + k + 1	STACK ← PC + 1 SP ← SP - 3 (3 bytes, 22 bits)

RCALL:
تفاوتی که این دستور با قبلی داره اینه که در اینجا پرش ما به اندازه 2K می تونه باشه ینی می تونیم 2K به جلو پرش بکنیم یا 2K به عقب
K اونجایی است که می خوایم بهش پرش بکنیم

RCALL – Relative Call to Subroutine

16-bit Opcode:

1101	kkkk	kkkk	kkkk

Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

Words	1 (2 bytes)
Cycles	3 devices with 16-bit PC
	4 devices with 22-bit PC

ICALL – Indirect Call to Subroutine

- Calls to a subroutine within the entire 4M (words) Program memory. The return address (to the instruction after the CALL) will be stored onto the Stack. The Stack Pointer uses a post-decrement scheme during CALL.

	Operation:	Comment:
(i)	PC(15:0) ← Z(15:0)	Devices with 16-bit PC, 128KB Program memory maximum.
(ii)	PC(15:0) ← Z(15:0) PC(21:16) ← 0	Devices with 22-bit PC, 8MB Program memory maximum.

	Syntax:	Operands:	Program Counter:	Stack:
(i)	ICALL	None	See Operation	STACK ← PC + 1 SP ← SP - 2 (2 bytes, 16 bits)
(ii)	ICALL	None	See Operation	STACK ← PC + 1 SP ← SP - 3 (3 bytes, 22 bits)

:ICALL
ما میتونیم یک Subroutines در فضای 4 مگابایتی ادرس AVR فراخوانی بکنیم
با اجرای این دستور اون ادرس بازگشت که درس بعد از RCALL هستش میاد و در استک ذخیره میشه
با ذخیره شدن اون ادرس در استک، استک پوینتر ما کاهش پیدا میکنه
این دستور هیچ اپرندی نداره
در این دستور برای PC که 22 بیت است بیت های بالای 15 با صفر پر میشن چون رجیستر Z ما 16 بیت داره که این باعث میشه محدودیت داشته باشیم و نتونیم از 4 مگابایت فضای ادرسمون استفاده بکنیم پس این مشکل رو توی EICALL حل میکنیم
پس اتفاقی که توی این دستور می افته اینه که ادرس پرشمون یا ادرس Subroutines مون رو توی Z داریم و این پرش میکنه به اون ادرس

ICALL – Indirect Call to Subroutine

16-bit Opcode:

1001	0101	0000	1001

Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

Words	1 (2 bytes)
Cycles	3 devices with 16-bit PC
	4 devices with 22-bit PC

EICALL – Extended Indirect Call to Subroutine

- Indirect call of a subroutine pointed to by the Z (16 bits) Pointer Register in the Register File and the EIND Register in the I/O space. This instruction allows for indirect calls to the entire 4M (words) Program memory space. The Stack Pointer uses a post-decrement scheme during EICALL.

Operation:

(i) PC(15:0) ← Z(15:0)

PC(21:16) ← EIND

	Syntax:	Operands:	Program Counter:	Stack:
(i)	EICALL	None	See Operation	STACK ← PC + 1 SP ← SP - 3 (3 bytes, 22 bits)

EICALL:
این مشکل بیت های بالای 15 رو از طریق یک رجیستر دیگری در فضای I/O تکمیل و حل میکنه و ما می تونیم به کل فضای ادرسمون که 4 مگابایت هستش دسترسی داشته باشیم
این دستور هم هیچ اپرندی نداره
ما اینجا دوتا رجیستر Z و EIND داریم
این دستور فقط مخصوص اون رجیسترهایی هستش که PCشون 22 بیتی است

EICALL – Extended Indirect Call to Subroutine

16-bit Opcode:

1001	0101	0001	1001

Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

Words 1 (2 bytes)

Cycles 4 (only implemented in devices with 22-bit PC)

RET – Return from Subroutine

- Every subroutine needs RET as the last instruction.
- Returns from subroutine. The return address is loaded from the STACK. The Stack Pointer uses a pre-increment scheme during RET.

	Operation:	Comment:
(i)	PC(15:0) ← STACK	Devices with 16-bit PC, 128KB Program memory maximum.
(ii)	PC(21:0) ← STACK	Devices with 22-bit PC, 8MB Program memory maximum.

	Syntax:	Operands:	Program Counter:	Stack:
(i)	RET	None	See Operation	SP ← SP + 2, (2 bytes, 16 bits)
(ii)	RET	None	See Operation	SP ← SP + 3, (3 bytes, 22 bits)

:RET
در هر Subroutines که می نویسیم ما نیاز داریم که اجرای برنامه رو برگردونیم به اون نقطه اولیه
انتهای هر Subroutines باید یک دستوری رو تحت عنوان ریترن قرار بدیم
این دستور انتهای Subroutines قرار میگیره و مارو از انتهای اون Subroutines به اون نقطه ای که باید برگرده برمی گردونه پس انتهای هر Subroutines ما این دستور رو داریم که ما رو برمیگردونه به اون محلی که قراره اجرای برنامه ادامه پیدا بکنه
با اجرای این دستور اون ادرس از استک برداشته میشه و استک پوینتر ما افزایش پیدا میکنه
پس با اجرای این دستور اون محتوای خانه بالای استک میاد توی PC قرار میگیره
این دستور هیچ اپرندی نداره

RET – Return from Subroutine

16-bit Opcode:

1001	0101	0000	1000

Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

Words	1 (2 bytes)
Cycles	4 devices with 16-bit PC
	5 devices with 22-bit PC

Stack

- Stack is a section of RAM used by CPU to store information temporarily
  - Data
  - Address
- CPU needs this storage area because there are only a limited number of registers
- There is a register in CPU to point to stack
  - Called Stack Pointer (SP) register
    - Two register in I/O memory space
      - SPL: Low byte of SP
      - SPH: High byte of SP

استک:
یک قسمتی از رم ما است که CPU میاد به صورت موقت یکسری اطلاعاتی رو اونجا ذخیره میکنه این اطلاعات میتونه دیتا یا ادرس باشه
چرا به استک نیاز داریم؟ CPU تعداد رجیسترهاش محدوده و برای اینکه ما بتونیم فراخوانی های متعددی داشته باشیم ممکنه این رجیسترها کافی نباشه بنابراین نیاز به فضایی داریم که یکسری اطلاعاتی مثل ادرس های بازگشت رو توی اون ها ذخیره بکنیم و یک مقدار فراتر از اون حجم حافظه ای باشه که رجیسترها در اختیار ما قرار میده پس این دلیل استفاده ما استک است
نحوه استفاده CPU از استک به این صورتکه ما یک رجیستر داریم تحت عنوان استک پوینتر یا SP و SP به اون خونه بالای حافظه استک داره اشاره میکنه و توی AVR این از سمت بالا به سمت پایین رشد خواهد کرد
خود استک پوینتر یک رجیستر 16 بیتی هستش و این از دوتا رجیستر 8 بیتی تشکیل شده به نام های SPH , SPL و SPL قسمت های پایین حافظه رجیستر استک پوینتر رو واسه ما فراهم میکنه و SPH قسمت بالای استک پوینتر رجیستر رو واسه ما فراهم میکنه

Operation on the Stack

- PUSH
  - Storing information on the Stack
    - SP points to top of the stack (TOS)
    - As we push data, the data are saved where SP points
    - SP is decrement by one

	Syntax:	Operands:	Program Counter:	Stack:
(i)	PUSH Rr	$0 \leq r \leq 31$	PC ← PC + 1	SP ← SP - 1

- POP
  - Loading stack contents back into a CPU register
    - POP is the opposite process of PUSH
    - SP is incremented
    - Top location of stack is copied back to the register

	Syntax:	Operands:	Program Counter:	Stack:
(i)	POP Rd	$0 \leq d \leq 31$	PC ← PC + 1	SP ← SP + 1

چجوری از استک استفاده بکنیم:
استک دوتا دستور داره برای اینکه بتونیم با اون کار بکنیم:
برای اینکه بتونیم یک اطلاعاتی رو درون استک قرار بدیم از دستور PUSH استفاده می کنیم و پوش باعث میشه یکسری اطلاعات درون استک ذخیره بشه
وقتی که دستور پوش رو استفاده میکنیم چه اتفاقی می افته؟ استک پوینتر ما داره به خانه بالای استک اشاره میکنه و وقتی که ما از پوش استفاده می کنیم میاد دیتا ما توی اون خونه ای که الان استک پوینتر داره بهش اشاره میکنه ذخیره میشه و بعد از این اتفاق استک پوینتر ما یک واحد زیاد میشه
وقتی که پوش میکنیم PC یک واحد زیاد میشه و SP یک واحد کم میشه و دلیل کم شدنش هم اینه که استک ما داره به سمت پایین رشد می کنه

POP: باعث میشه که ما بتونیم یک اطلاعاتی رو از استک برداریم و در یک رجیستری ذخیره بکنیم که CPU بتونه ازش استفاده بکنه

Initializing the Stack Pointer

- The SP register contains the value 0
  - When the AVR is powered on
- We must initialize the SP at the beginning of the program
  - Stack grows from higher memory Location to the lower memory location
  - It is common to initialize SP to uppermost memory location
    - Different AVRs have different amounts of RAM
    - In the AVR assembler, RAMEND represents the address of the last RAM location
    - To initialize SP
      - Load RAMEND into SP

نکته:
توی AVR وقتی که روشن می کنیم AVR رو و شروع میکنه به کار کردن استک پوینتر به خونه صفر اشاره میکنه و این اون چیزی که ما انتظار داریم نیست پس باید بیایم اونو اماده سازی بکنیم پس در شروع برنامه ما میایم استک پوینتر رو متناسب به اون چیزی که میخوایم داشته باشیم ینی گفتیم از بالای حافظه شروع بشه میایم و اینو مقدار دهی میکنیم
پس وقتی که مقدار دهی کردیم این از سمت بالا به سمت پایین رشد میکنه
اما از اونجایی که AVR های که در اختیار داریم میزان حافظه رم متفاوتی دارند این نیاز به یک مقدار دقت داره اسمبلر Avr توی اون فایل های تعریف خودش یک سری تمدیداتی برای این داره ما RAMEND رو داریم که به خونه انتهایی حافظه رم اشاره میکنه و برای اینکه بیایم initialize بکنیم این sp در ابتدای برنامه کافیه که RAMEND رو داخل استک پوینتر قرار بدیم و این می تونه اون عملکردی که داریم به صورت رشد از بالا به سمت پایین رو برای ما داشته باشه

CALL\RET instructions and the role of Stack

- When a subroutine is called
  - CPU saves the address of the instruction just bellow the CALL instruction on the stack
    - To know where to resume when it returns from called subroutine

PUSH (instruction bellow CALL) onto the stack

- When RET instruction at the end of the subroutine is executed
  - The top location of stack copied back to PC
  - SP is incremented

POP (instruction bellow CALL) into the PC

نقش استک در اجرای دستورات CALL, RET و فراخوانی subroutine ها مرور بکنیم:
وقتی که ما میخوایم یک subroutine رو فراخوانی بکنیم میایم از دستور CALL استفاده میکنیم و اتفاقی که می افته اینه که CPU میاد ادرس بازگشت که میشه دقیقا اون ادرس بعدی بعد از دستور CALL رو در استک ذخیره میکنه و دلیل این کار اینه که بدونه بعد از اجرای اون subroutine باید از کجا اجرای برنامه رو ادامه بده و اینجا مثل این می مونه که ما داریم ادرس بازگشت رو پوش میکنیم توی حافظه و وقتی که می خوایم بازگشت کنیم از اون subroutine از دستور RET استفاده میکنیم و توی این دستور استک پوینتر ما داره به بالاترین خونه اشاره میکنه و یک واحد کمش می کنیم و محتوای اون رو PC منتقل می کنیم و این مثل این می مونه که ما داریم دستور POP رو اجرا می کنیم

مثال

Toggle all the bits of Port B by sending to it the values $55 and $AA continuously. Put a time delay between each issuing of data to Port B.

Initialization of SP

```
.INCLUDE "M32DEF.INC"
.ORG 0
      LDI R16,HIGH(RAMEND)     ;load SPH
      OUT SPH,R16
      LDI R16,LOW(RAMEND)      ;load SPL
      OUT SPL,R16
```

Main program

```
BACK:
      LDI   R16,0x55      ;load R16 with 0x55
      OUT   PORTB,R16     ;send 55H to port B
      CALL  DELAY         ;time delay
      LDI   R16,0xAA      ;load R16 with 0xAA
      OUT   PORTB,R16     ;send 0xAA to port B
      CALL  DELAY         ;time delay
      RJMP  BACK          ;keep doing this indefinitely
```

Delay subroutine

```
;-------  this is the delay subroutine
      .ORG 0x300          ;put time delay at address 0x300
DELAY:
      LDI   R20,0xFF      ;R20 = 255,the counter
AGAIN:
      NOP                 ;no operation wastes clock cycles
      NOP
      DEC   R20
      BRNE  AGAIN         ;repeat until R20 becomes 0
      RET                 ;return to caller
```

مثال:
برای اینکه از استک به صورت صحیح استفاده بکنیم باید اونو مقداردهی اولیه بکنیم ینی initialized بکنیم
توی این مثال قراره که پورت B رو به صورت یکی در میان ست و ریست بکنیم با یک تاخیر خاصی
اون فرایند ایجاد تاخیر توی یک subroutine قرار داره به اسم subroutine DELAY
استفاده از subroutine باعث میشه که کد ما خیلی ساختارمند بشه

Macros vs. Subroutine

Macros and subroutines are useful in writing assembly programs, but each has limitations. Macros increase code size every time they are invoked. For example, if you call a 10-instruction macro 10 times, the code size is increased by 100 instructions; whereas, if you call the same subroutine 10 times, the code size is only that of the subroutine instructions. On the other hand, a function call takes 3 or 4 clocks and the RET instruction takes 4 clocks to get executed. So, using functions adds around 8 clock cycles. The subroutines use stack space as well when called, while the macros do not.

تفاوت ماکرو با subroutine:
ماکروها به نوعی کمک می کرد که از کد زدن تکراری جلوگیری بکنیم
توی ماکرو فقط توی نوشتار جایگزین خطوط کد ما می شد و این باعث میشد اگر بخوایم مرتبا اونو تکرار بکنیم حجم کد ما زیاد بشه ولی این در مورد subroutine صدق نمی کنه ینی
subroutine یکبار نوشته میشه و بعد فراخوانی به اونجا منتقل میشه ینی از طریق فراخوانی اجرای برنامه و کنترل برنامه به اون نقطه منتقل خواهد شد پس subroutine توی فضای حافظه کد ما صرفه جویی میکنه ولی در مقابل یک مقداری باعث میشه که تاخیر ایجاد بکنه ینی ما یکسری سیکل ساعت اضافی داریم توی این حالت
subroutine از اونجایی که ادرس بازگشت رو توی استک ذخیره میکنه یک فضایی از حافظه رم رو هم به این صورت داره مصرف میکنه

AVR Timer Delay

- Delay subroutine
  - How to generate various time delays
  - How to calculate exact delays for AVR

- Two factors that can affect the accuracy of the delay
  - The crystal frequency
  - The AVR design

subroutine ها قطعه کدهای تکراری هستند ینی ممکنه زیاد در کد استفاده بشن
یکی از subroutine های مهمی که در عمل ازش استفاده میکنیم delay subroutine هستش
ینی اینکه ما یک subroutine داشته باشیم که کمک بکنه که فواصل زمانی ایجاد بکنیم بین رویدادهایی رو که قراره با هم فاصله زمانی داشته باشند
درباره ایجاد زیروال یک تاخیر چند نکته وجود داره:
1- چجوری یک تاخیر مشخصی رو ایجاد بکنیم و تاخیر دلخواه خودمونو با زیروال های مناسبی ایجاد بکنیم
2- یک زیروال که ما نوشتیم دقیقا چقدر تاخیر در یک تراشه avr خاص می تونه ایجاد بکنه

عوامل موثر در یک تراشه avr که می تونه روی یک subroutine تاثیرگذار باشه:
1- فرکانس کاری اون میکرو
2- طراحی اون avr

AVR Timer Delay

- Machine cycle
  - Certain amount of time for the CPU to execute an instruction
  - More instructions take no more than one or two machine cycles
  - The length of machine cycle depends on the frequency of the oscillator
  - One machine cycle consist of one oscillator period
    - To calculate the machine cycle
      - We take the inverse of the oscillator frequency

هر دستوری برای اینکه بتونه اجرا بشه در cpu ما نیاز به یک زمانی داره
به اون میزان زمان خاصی که یک cpu میاد یک دستور رو اجرا بکنه می گن machine cycle
و این تعیین می کنه که الان این دستور ما چقدر زمان برای اجرای خودش نیاز داره
در avr به صورت خاص و با توجه به اینکه دستورات عموما در یک سیکل ساعت و بعضا در دو سیکل ساعت انجام میشن این ها در واقع به اندازه یک یا دو Machine cycle زمان نیاز دارند که اجرا بشن
مدتی که یک Machine cycle داره این وابسته به فرکانس اون oscillator هستش که در avr استفاده شده
یک Machine cycle برابر با یک دوره oscillator هستش و برای اینکه بیایم این دوره رو حساب بکنیم کافیه که معکوس فرکانس oscillator رو حساب بکنیم

مثال

The following shows the crystal frequency for four different AVR-based systems. Find the period of the instruction cycle in each case.
(a) 8 MHz (b) 16 MHz (c) 10 MHz (d) 1 MHz

(a) instruction cycle is 1/8 MHz = 0.125 µs (microsecond) = 125 ns (nanosecond)
(b) instruction cycle = 1/16 MHz = 0.0625 µs = 62.5 ns (nanosecond)
(c) instruction cycle = 1/10 MHz = 0.1 µs = 100 ns
(d) instruction cycle = 1/1 MHz = 1 µs

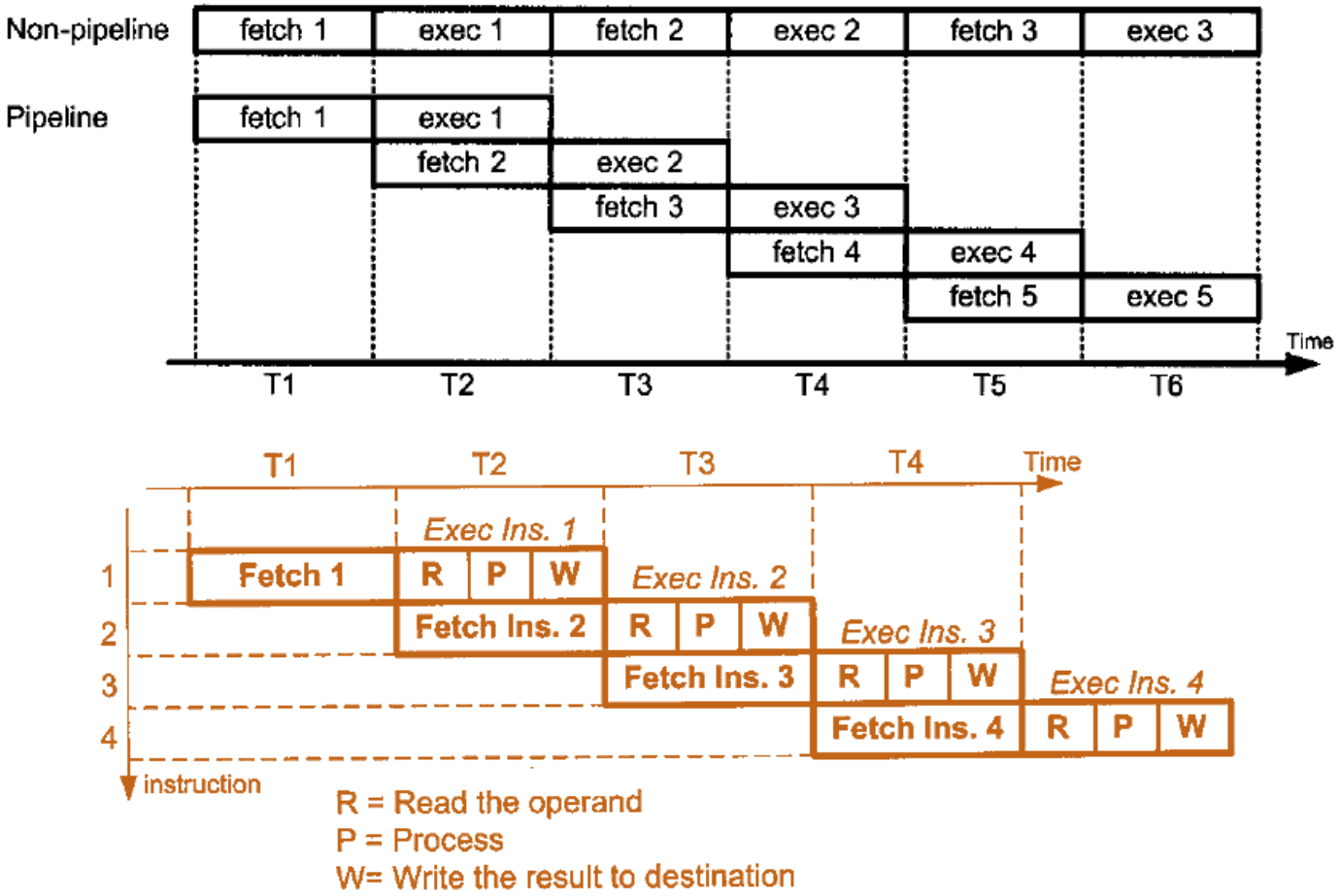
این مقادیر به دست امده برای یک دستور یک سیکلی هستن

مثال:
فرض کنید که ما 4 تا سیستم داریم که فرکانسشون روبه رو اومده و میخوایم ببینم طول یک Machine cycle که اینجا داریم چقدر است؟

نکته: هر چه فرکانس کمتر باشه ما زمان بیشتری نیاز داریم که یک دستور یک سیکلی رو انجام بدیم

Pipelining

- Overlapping of fetch and execution



در حالت عادی اگر ما بخوایم دستورات رو اجرا بکنیم اول اون هارو فچ می کنیم و بعد اجرا میکنیم ینی برای دستور اول ابتدا فچ و بعد اجرا - و بعد دستور دوم فچ و بعد اجرا و به همین ترتیب..
اما در رابطه ما با avr ما یک Pipelining داریم ینی دستور اول فچ می شه و در زمانی که می خواد دستور اول اجرا بشه دستور دوم همزمان میاد و فچ میشه و این روند همینطوری پیش می ره...
و این کار باعث میشه که ما بتونیم هر دستور رو در یک سیکل ساعت انجام بدیم

در پایین نمودار زمانیشو نشون داده:
اجرا شدن در سه فاز هست: اینکه اپرندها خونده بشه ینی R و فرایند اجرا بشه ینی P و نتایج در اون جایی که باید نوشته بشه وارد بشه ینی W

Branch Penalty

- For the concept of pipelining to work
  - We need a buffer or queue
    - Which an instruction is prefetched and ready to be executed
- When a branch instruction is executed
  - CPU starts to fetch codes from new location
    - The code in the queue that was fetched previously is discarded
    - In this case, the execution unit must wait until the fetch unit fetches the new instruction
      - Called **branch penalty**

با توجه به Pipelining که داریم اینجا یک اتفاق می تونه این وسط بیوفته که این روند رو از اون حالت عادی خودش خارج بکنه و اون هم branch هستش ینی اگر ما اگر در دستورات یک پرش داشته باشیم این می تونه روال عادی این Pipelining رو بهم بزنه
اتفاقی که توی سیستم هایی که Pipelining دارن اینه که: اینه که ما یک بافر و یک صف نیاز داریم که اون دستوری که قراره همزمان با اجرای دستور قبلیش فچ بشه در اون ذخیره میشه ینی یک بافر و یک صفی داریم که دستورات اونجا prefetched میشن و اونجا اماده اجرا خواهند بود وقتی که با یک دستور پرش برخورد می کنیم و اون میخواد اجرا بشه cpu مجبوره بره کدش رو از یک جای جدیدی بره برداره و این باعث میشه که اون دستور قبلیه که رفته prefetched شده و داخل صف قرار گرفته دیگه بدرد نخوره و ما مجبوریم اون دستور رو دور بزنیم و بریم از اون محل جدید دستور دیگه بیاریم و این باعث میشه که ما یک مقداری اون روال عادیمون دچار خدشه بشه ینی توی این حالت که داریم پرش میکنیم به اون دستور جدید واحد اجرای ما دیگه چیزی واسه اجرا شدن نداره پس یک پنالتی اینجا داریم که بهش میگن branch penalty ینی دستوری که قرار بود prefetched بشه الان دیگه بدرد ما نمی خوره و ما دستور دیگه رو فچ بکنیم در نتیجه اون زمانی که واحد فچ ما داره دستور مناسب رو میاره توی اون رجیستر که بعدا اجراش بکنه اون قسمت اجرا شدن چیزی واسه اجرا نداره و بیکار خواهد بود که بهش میگن branch penalty

مثال

For an AVR system of 1 MHz, find how long it takes to execute each of the following instructions:

(a) LDI	(b) DEC	(c) LD
(d) ADD	(e) NOP	(f) JMP
(g) CALL	(h) BRNE	(i) .DEF

The machine cycle for a system of 1 MHz is 1 µs,

Instruction	*Instruction cycles*	*Time to execute*
(a) LDI	1	1 × 1 µs = 1 µs
(b) DEC	1	1 × 1 µs = 1 µs
(c) OUT	1	1 × 1 µs = 1 µs
(d) ADD	1	1 × 1 µs = 1 µs
(e) NOP	1	1 × 1 µs = 1 µs
(f) JMP	3	3 × 1 µs = 2 µs
(g) CALL	4	4 × 1 µs = 4 µs
(h) BRNE	2/1	(2 µs taken, 1 µs if it falls through)
(i) .DEF	0	(directive instructions do not produce machine instructions)

مثال:
فرض کنید یک avr دارید که با 1 مگاهرتز داره کار میکنه و میخوایم ببینم اجرای دستوراتی که نوشته چقدر طول می کشه؟
که پایینش نوشته
دستور branch بسته به اینکه شرط درسته یا نادرسته می تونه توی یک سیکل ساعت یا دو سیکل ساعت انجام بشه
دستورات دایرکتیو چون توسط Cpu اجرا نمیشن سیکل ساعتی هم به خودشون اختصاص نمیدن

مثال

Find the size of the delay of the code snippet below if the crystal frequency is 10 MHz:

```
                                        Instruction Cycles
          .DEF COUNT = R20                      0
DELAY:    LDI      COUNT, 0xFF                  1
AGAIN:    NOP                                   1
          NOP                                   1
          DEC      COUNT                        1
          BRNE     AGAIN                        2/1
          RET                                   4
```

Therefore, we have a time delay of $[1 + ((1 + 1 + 1 + 2) \times 255) + 4] \times 0.1\ \mu s = 128.0\ \mu s$. Notice that BRNE takes two instruction cycles if it jumps back, and takes only one when falling through the loop. That means the above number should be 127.9 μs.

مثال:
می خوایم ببینیم این ساب روتین چقدر طول میکشه که اجرا بشه اگر فرکانس کاری میکرویی که قراره اینو اجرا بکنه 10 مگاهرتز باشه؟
مقدار count ما FF هست واسه همین این حلقه 255 بار تکرار میشه
نکته: در اخرین دور اجرای این حلقه شرط برنچ درست خواهد بود و در یک سیکل ساعت اجرا میشه و اگر بخوایم اونو لحاظ بکنیم و یک سیکل ساعت که برابر با 0.1 میکروثانیه است از کل جواب کم بکنیم تاخیر دقیقش به دست میاد

Delay Calculation for AVR

- Delay subroutine consists of two parts
  - Setting the counter
  - A loop
- Most of the time delay is performed by the body of the loop
- Very often
  - We calculate the time delay based in the instructions inside the loop
  - Ignore the clock cycles associated with the instructions outside of the loop

در بسیاری از موارد چون Delay subroutine از دو قسمت تشکیل شده ینی:
setting the counter
loop
معمولا تاخیر عمده ای که این subroutine به ما میده همون تاخیر ناشی از بدنه کد هستش ینی همون حلقه بنابراین توی عمل ممکنه از اون یک سیکل ساعت که مال LDI هست و 4 سیکل ساعت که مال RET هست توی مثال قبل داشتیم بگذریم و فقط تعداد تکرارهای بدنه حلقه رو حساب کنیم
اگر بخوایم بر این مبنا عمل بکنیم میشه مثال صفحه بعدی...

مثال

Find the size of the delay in the following program if the crystal frequency is 1 MHz:

```
.INCLUDE "M32DEF.INC"
.ORG 0
          LDI R16,HIGH(RAMEND) ;initialize SP
          OUT SPH,R16
          LDI R16,LOW(RAMEND)
          OUT SPL,R16
BACK:
          LDI R16,0x55         ;load R16 with 0x55
          OUT PORTB,R16        ;send 55H to port B
          RCALL DELAY          ;time delay
          LDI R16,0xAA         ;load R16 with 0xAA
          OUT PORTB,R16        ;send 0xAA to port B
          RCALL DELAY          ;time delay
          RJMP BACK            ;keep doing this indefinitely
;-----this is the delay subroutine
          .ORG 0x300           ;put time delay at address 0x300
DELAY:    LDI R20,0xFF         ;R20 = 255,the counter
AGAIN:
          NOP                  ;no operation wastes clock cycles
          NOP
          DEC R20
          BRNE AGAIN           ;repeat until R20 becomes 0
          RET                  ;return to caller
```

مثال:
توی این مثال می خوایم محاسبه بکنیم تاخیر توی delay رو
صفحه بعدی...

مثال

we have the following machine cycles for each instruction of the DELAY subroutine:

		Instruction Cycles
DELAY:	LDI R20,0xFF	1
AGAIN:	NOP	1
	NOP	1
	DEC R20	1
	BRNE AGAIN	2/1
	RET	4

Therefore, we have a time delay of $[1 + (255 \times 5) - 1 + 4] \times 1\ \mu s = 1279\ \mu s$.

دور اخر که شرطش درسته یک واحد ازش کم میشه

پایان

موفق و پیروز باشید