ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

TRƯƠNG XUÂN THẮNG

GIAO TIẾP VỚI VI ĐIỀU KHIỂN ARM

LUẬN VĂN THẠC SĨ

Hà Nội - 2011

MỤC LỤC

KÝ HIỆU CÁC CHỮ VIẾT TẮT

ADC	Analog to Digital Converter	Bộ chuyển đổi tương tự sang số
AMBA	Advanced Microcontroller Bus Architecture	Kiến trúc bus truyền vi điều khiển cao cấp
AHB	Advanced High-performance Bus	Bus truyền dữ liệu hiệu suất cao
AIC	Advanced Interrupt Controller	Bộ điều khiển ngắt cao cấp
ASIC	Application-Specific Integrated Circuit	Mạch tích hợp chuyên dụng
ASB	Advanced System Bus	Hệ thống bus truyền đa năng
API	Application Programming Interface	Giao diện lập trình ứng dụng
APB	Advanced Peripheral Bus	Bus truyền ngoại vi đa năng
BRG	Baud Rate Generator	Bộ tạo tốc độ Baud
CLK	Clock	Xung nhịp
CMSIS	The Cortex Microcontroller Software Interface Standard	Chuẩn giao tiếp phần mềm vi điều khiển Cortex
CRC	Cyclic Redundancy Check	Kiểm tra độ dư vòng
DMA	Direct Memory Access	Sự truy cập bộ nhớ trực tiếp
DSP	Digital Signal Processors	Bộ xử lý tín hiệu số
DRAM	Dynamic Random Access Memory	Bộ nhớ truy cập ngẫu nhiên động
EEPROM	Electrically Erasable Programmable Read-Only Memory	Bộ nhớ chỉ đọc có thể xóa được bằng điện
EPROM	Erasable Programmable Read-Only Memory	Bộ nhớ chỉ để đọc có khả năng lập trình lại được
FAT	File Allocation Table	Bảng phân bố tập tin
FIFO	First In First Out	Vào trước ra trước
FIQ	Fast Interrupt Request	Yêu cầu ngắt nhanh
GIPO	General Purpose Input/Output	Đầu vào hoặc ra đa mục đích
GSM	Global System for Mobile Communications	Hệ thống truyền thông di động toàn cầu
IEM	Intelligent Energy Management	Bộ quản lý mức tiêu thụ năng lượng thông minh
IRQ	Interrupt Request	Yêu cầu ngắt
LCD	Liquid Crystal Display	Màn hình tinh thể lỏng
LSB	Least Significant Bit	Bit có giá trị thấp nhất

MAC	Multiply-Accumulate Unit	Bộ tích lũy nhân
MSB	Most Significant Bit	Bit có giá trị cao nhất
PDA	Personal Digital Assistant	Máy hỗ trợ cá nhân kỹ thuật số
PLD	Programmable Logic Device	Bộ logic có khả năng lập trình
PLL	Phase Lock Loop	Vòng khóa pha
PMC	Power Management Controller	Bộ quản lý nguồn
PWM	Pulse Width Modulation	Bộ điều chế độ rộng xung
PHY	Physical	Lớp vật lý
RAM	Random Access Memory	Bộ nhớ truy nhập ngẫu nhiên
ROM	Read Only Memory	Bộ nhớ chỉ đọc
RTC	Real Time Clock	Đồng hồ thời gian thực
Rx	Receive	Nhận dữ liệu
SD Card	Secure Digital Card	Thẻ nhớ dữ liệu số
SPI	Serial Peripheral Interface	Giao tiếp ngoại vi nối tiếp
SRAM	Static Random Access Memory	Bộ nhớ truy cập ngẫu nhiên tĩnh
SSRAM	Synchronous Static Random Access Memory	Bộ nhớ truy cập ngẫu nhiên đồng bộ tĩnh
Tx	Transmit	Truyền dữ liệu
TIC	Test Interface Controller	Bộ giao tiếp kiểm thử
UART	Universal Asynchronous Receiver/Transmitter	Bộ thu/phát không đồng bộ đa năng
USB	Universal Serial Bus	Bus nối tiếp đa năng
VGA	Video Graphics Array	Mảng đồ họa hình ảnh

MỞ ĐẦU

Xuất hiện từ những năm đầu thập niên 1960, hệ thống nhúng đang phát triển mạnh mẽ trong lĩnh vực kỹ thuật điện tử và công nghệ thông tin, với những ứng dụng rộng rãi trong công nghiệp và đời sống.

Hệ thống nhúng hiện nay đòi hỏi phải có cấu trúc mạnh, đáp ứng thời gian thực tốt, dung lượng bộ nhớ lớn, khả năng tính toán nhanh, khả năng tiêu thụ năng lượng thấp, tính ổn định cao và tích hợp sẵn nhiều ngoại vi.

Vi điều khiển ARM được đánh giá là một trong những dòng vi điều khiển mạnh, đáp ứng được những yêu cầu trong hệ thống nhúng ngày nay, được sử dụng rộng rãi ở trên thế giới và đang được nghiên cứu phát triển ở Việt Nam.

Trong khuôn khổ của đề tài, ta sẽ tìm hiểu mô hình kiến trúc, các giao tiếp với vi điều khiển ARM, đặc điểm chung của dòng lõi xử lý này và thử nghiệm một số ứng dụng giao tiếp với vi điều khiển AT91SAM7S64 có lõi xử lý là ARM7TDMI.

PHẦN I - LÝ THUYẾT CHUNG

CHƯƠNG 1

CẤU TRÚC VI ĐIỀU KHIỂN ARM

✓ Để có thể thực hiện giao tiếp với vi điều khiển ARM thì yêu cầu trước hết đặt ra là phải hiểu rõ về cấu trúc và những đặc điểm của vi điều khiển này [5] [6].

1.1 Đôi nét về lịch sử hình thành và phát triển vi điều khiển ARM

Việc thiết kế ARM được bắt đầu từ năm 1983 trong một dự án phát triển của công ty máy tính Acorn.

Nhóm thiết kế, dẫn đầu bởi Roger Wilson và Steve Furber, bắt đầu phát triển một bộ vi xử lý có nhiều điểm tương đồng với kỹ thuật MOS 6502 tiên tiến. Acorn đã từng sản xuất nhiều máy tính dựa trên 6502, vì vậy việc tạo ra một chip như vậy là một bước tiến đáng kể của công ty này.

Nhóm thiết kế hoàn thành việc phát triển mẫu gọi là ARM1 vào năm 1985, và vào năm sau, nhóm hoàn thành sản phẩm ARM2. ARM2 có đường truyền dữ liệu 32 bit, không gian địa chỉ 26 bit tức cho phép quản lý đến 64 Mbyte địa chỉ và 16 thanh ghi 32 bit. Một trong những thanh ghi này đóng vai trò là bộ đếm chương trình với 6 bit có giá trị cao nhất và 2 bit có giá trị thấp nhất lưu giữ các cờ trạng thái của bộ vi xử lý. Thế hệ sau, ARM3 được tạo ra với 4KB bộ nhớ đệm và có chức năng được cải thiện tốt hơn nữa.

Vào những năm cuối thập niên 80, hãng máy tính Apple Computer bắt đầu hợp tác với Acorn để phát triển các thế hệ lõi ARM mới. Công việc này trở nên quan trọng đến nỗi Acorn nâng nhóm thiết kế trở thành một công ty mới gọi là Advanced RISC Machines. Từ lý do đó hình thành chữ viết tắt ARM của Advanced RISC Machines thay vì Acorn RISC Machine. Về sau, Advanced RISC Machines trở thành công ty ARM Limited.

Kết quả sự hợp tác này là ARM6. Mẫu đầu tiên được công bố vào năm 1991 và Apple đã sử dụng bộ vi xử lý ARM 610 dựa trên ARM6 làm cơ sở cho PDA hiệu Apple Newton. Vào năm 1994, Acorn dùng ARM 610 làm CPU trong các máy vi tính RiscPC của họ.

Trải qua nhiều thế hệ nhưng lõi ARM gần như không thay đổi kích thước. ARM2 có 30.000 transistors trong khi ARM6 chỉ tăng lên đến 35.000. Ý tưởng của nhà sản xuất lõi ARM là sao cho người sử dụng có thể ghép lõi ARM với một số bộ phận tùy chọn nào đó để tạo ra một CPU hoàn chỉnh, một loại CPU mà có thể tạo ra trên những nhà máy sản xuất bán dẫn cũ và vẫn tiếp tục tạo ra được sản phẩm với nhiều tính năng mà giá thành vẫn thấp.

Thế hệ khá thành công của hãng là lõi xử lý ARM7TDMI, với hàng trăm triệu lõi được sử dụng trong các máy điện thoại di động, hệ thống video game cầm tay.

ARM đã thành một thương hiệu đứng đầu thế giới về các ứng dụng sản phẩm nhúng đòi hỏi tính năng cao, sử dụng năng lượng ít và giá thành thấp.

Chính nhờ sự nổi trội về thị phần đã thúc đẩy ARM liên tục được phát triển và cho ra nhiều phiên bản mới.

Những thành công quan trọng trong việc phát triển ARM:

- Giới thiệu ý tưởng về định dạng các tập lệnh được nén lại (Thumb) cho phép tiết kiệm năng lượng và giảm giá thành ở những hệ thống nhỏ.
- Giới thiệu các họ điều khiển ARM.
- Phát triển môi trường làm việc ảo của ARM trên máy tính.
- Các ứng dụng cho hệ thống nhúng dựa trên lõi xử lý ARM ngày càng trở nên rộng rãi.

Hầu hết các nguyên lý của hệ thống trên chip và cách thiết kế bộ xử lý hiện đại được sử dụng trong ARM, ARM còn đưa ra một số khái niệm mới như giải nén động các dòng lệnh. Việc sử dụng ba trạng thái nhận lệnh – giải mã – thực thi trong mỗi chu kỳ máy mang tính quy phạm để thiết kế các hệ thống xử lý thực. Do đó, lõi xử lý ARM được sử dụng rộng rãi trong các hệ thống phức tạp.

1.2 Cấu trúc cơ bản ARM

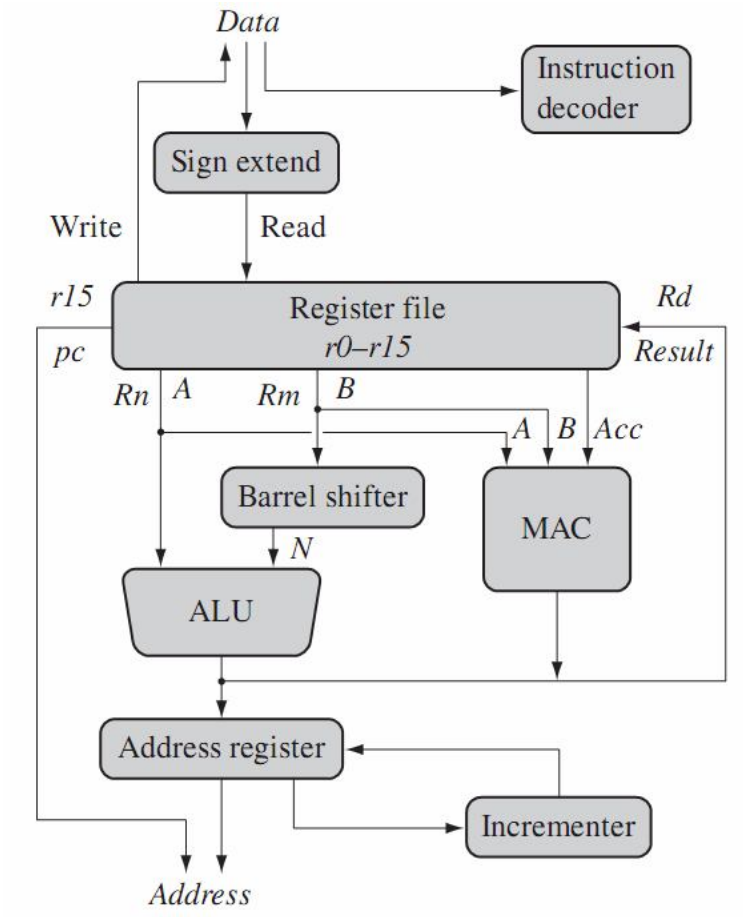
- Cấu trúc load-store (nạp-lưu trữ).
- Cho phép truy xuất dữ liệu không thẳng hàng.
- Tập lệnh trực giao.
- Tập lệnh ARM-32bit.
- Hầu hết các lệnh đều thực hiện trong vòng một chu kỳ đơn.

Trong ARM có một số tính chất mới như sau:

- Hầu hết tất cả các lệnh đều cho phép thực thi có điều kiện, điều này làm giảm việc phải viết các tiêu đề rẽ nhánh cũng như bù cho việc không có một bộ dự đoán rẽ nhánh.
- Trong các lệnh số học, để chỉ ra điều kiện thực hiện, người lập trình chỉ cần sửa mã điều kiện.
- Có một thanh ghi dịch 32 bit mà có thể sử dụng đầy đủ chức năng với hầu hết các lệnh số học và việc tính toán địa chỉ.
- Có các kiểu định địa chỉ theo chỉ số rất mạnh.
- Có hệ thống con thực hiện ngắt hai mức ưu tiên đơn giản nhưng rất nhanh, kèm theo cho phép chuyển từng nhóm thanh ghi.

1.3 Mô hình kiến trúc

Các thành phần nhúng cùng với một lõi xử lý ARM được mô tả trong hình 1.1. Đây cũng là một kiến trúc chung trong họ xử lý với lõi ARM.



Hình 1.1: **Mô hình kiến trúc lõi xử lý ARM.**

Lõi xử lý ARM là một khối chức năng được kết nối bởi các bus dữ liệu, các mũi tên thể hiện cho dòng chảy của dữ liệu, các đường thể hiện cho bus dữ liệu, và các ô biểu diễn trong hình là một khối hoạt động hoặc một vùng lưu trữ. Cấu hình này cho thấy các dòng dữ liệu và các thành phần tạo nên một bộ xử lý ARM.

Dữ liệu đi vào lõi xử lý thông qua các bus dữ liệu. Các dữ liệu có thể là một hướng để thực hiện hoặc một trường dữ liệu. Hình 1.1 cho thấy ưu điểm kiến trúc Harvard của ARM là sử dụng trên hai bus truyền khác nhau (bus dữ liệu và bus lệnh tách riêng), còn kiến trúc Von Neumann chia sẻ dữ liệu trên cùng bus.

Các bộ giải mã sẽ định hướng dịch chuyển trước khi chúng được thực thi. Mỗi một chỉ lệnh thực hiện thuộc về một tập lệnh riêng biệt.

Bộ xử lý ARM, giống như tất cả bộ xử lý RISC, sử dụng kiến trúc load-store. Điều này có nghĩa là có hai loại chỉ lệnh để chuyển dữ liệu vào và ra của bộ xử lý: lệnh load cho phép sao chép dữ liệu từ bộ nhớ vào thanh ghi trong lõi xử lý, và ngược lại lệnh store cho phép sao chép dữ liệu từ thanh ghi tới bộ nhớ. Không có lệnh xử lý dữ liệu trực tiếp trong bộ nhớ. Do đó, việc xử lý dữ liệu chỉ được thực hiện trong các thanh ghi.

Tất cả dữ liệu thao tác nằm trong các thanh ghi, các thanh ghi có thể là toán hạng nguồn, toán hạng đích, con trỏ bộ nhớ. Các dữ liệu 8 bit, 16 bit đều được mở rộng thành 32 bit trước khi đưa vào thanh ghi.

Tập lệnh ARM nằm trong hai nguồn thanh ghi Rn và Rm, và kết quả được trả về thanh ghi đích Rd. Nguồn toán hạng được đọc từ thanh ghi đang sử dụng trên bus nội bộ A và B tương ứng.

Khối số học và logic (ALU: Arithmetic Logic Unit) hay bộ tích lũy nhân (MAC: Multiply-Accumulate Unit) lấy các giá trị thanh ghi Rn và Rm từ bus A và B, và tính toán kết quả (bộ tích lũy nhân có thể thực hiện phép nhân giữa hai thanh ghi và cộng kết quả với một thanh ghi khác). Các lệnh xử lý dữ liệu ghi các kết quả trực tiếp trong Rd rồi trả về tệp thanh ghi.

Một tính năng quan trọng của ARM là thanh ghi Rm còn có thể được xử lý trước trong shifter (bộ dịch chuyển) trước khi nó đi vào ALU. Shifter và ALU có thể phối hợp với nhau để tính toán các biểu thức và địa chỉ.

Mô hình thanh ghi theo kiến trúc Registry – Registry, giao tiếp với bộ nhớ thông qua các lệnh load-store, các lệnh load và store sử dụng ALU để tính toán địa chỉ được lưu trong các thanh ghi địa chỉ, ngoài ra tập lệnh này còn sử dụng ALU để tạo ra địa chỉ được tổ chức trên địa chỉ thanh ghi và truyền đi trên các bus địa chỉ. Bộ gia tốc dùng trong các trường hợp truy xuất các vùng nhớ liên tục.

Sau khi đi qua các khối chức năng, kết quả trong Rd được ghi trở lại tệp thanh ghi. Tập lệnh load-store cập nhật tăng địa chỉ thanh ghi trước khi lõi xử lý đọc hoặc ghi giá trị thanh ghi từ vị trí nhớ tuần tự tiếp theo. Lõi xử lý tiếp tục thực hiện các lệnh cho đến khi xảy ra một ngắt ngoại lệ hoặc có thay đổi dòng chảy thực hiện bình thường.

Trên là tổng quan về bộ xử lý ARM. Các thành phần chính của bộ vi xử lý gồm lõi xử lý, các thanh ghi, kiến trúc đường ống sẽ được trình bày trong phần kế tiếp.

Chế độ hoạt động của ARM:

ARM có bẩy chế độ hoạt động, chế độ người dùng là chế độ cơ bản và ít đặc quyền nhất, khi đó CPU thực hiện mã hóa dữ liệu cho người dùng.

Các chế độ hoạt động của ARM được mô tả trong bảng 1.1.

Bảng 1.1: **Các chế độ hoạt động của RAM.**

Chế độ	Ký hiệu quy ước	Mức được ưu tiên	Chế độ các bit [4:0]
Abort	abt	có	1 0 1 1 1
Fast Interrupt Request	fiq	có	1 0 0 0 1
Interrupt Request	irq	có	1 0 0 1 0
Supervisor	svc	có	1 0 0 1 1
System	sys	có	1 1 1 1 1
Undefined	und	có	1 1 0 1 1
User	usr	không	1 0 0 0 0

Trong đó:

- Abort : Được nhập vào sau khi dữ liệu hoặc lệnh được bỏ qua quá trình tiền nạp.
- FIQ : Xử lý các ngắt có mức ưu tiên cao, hỗ trợ việc truyền dữ liệu và các kênh xử lý
- IRQ : Được sử dụng cho việc xử lý các ngắt mục đích chung.
- Supervisor : Chế độ bảo vệ dùng cho hệ điều hành .
- System : Chế độ ưu tiên, dùng cho hệ điều hành .
- Undefined : Dùng cho trường hợp mã lệnh không hợp lệ.
- User : Chế độ người dùng có mức ưu tiên thấp.

Các chế độ này có thể được thiết lập bằng phần mềm hoặc thông qua các ngắt bên ngoài hoặc thông qua quá trình xử lý ngoại lệ. Phần lớn các chương trình ứng dụng được thực thi trong chế độ ***User***. Mỗi chế độ điều khiển đều có các thanh ghi hỗ trợ để tăng tốc độ bắt các ngoại lệ.

1.4 Mô hình thiết kế ARM

1.4.1 Lõi xử lý

Dạng đơn giản của lõi xử lý gồm những phần cơ bản sau:

- Program Counter (PC): Bộ đếm chương trình: giữ địa chỉ của lệnh hiện tại.
- Thanh ghi tích lũy (ACC): giữ giá trị dữ liệu khi đang làm việc.
- Đơn vị xử lý số học (ALU): thực thi các lệnh nhị phân như cộng, trừ, gia tăng…
- Thanh ghi lệnh (IR): giữ tập lệnh hiện tại đang thực thi.

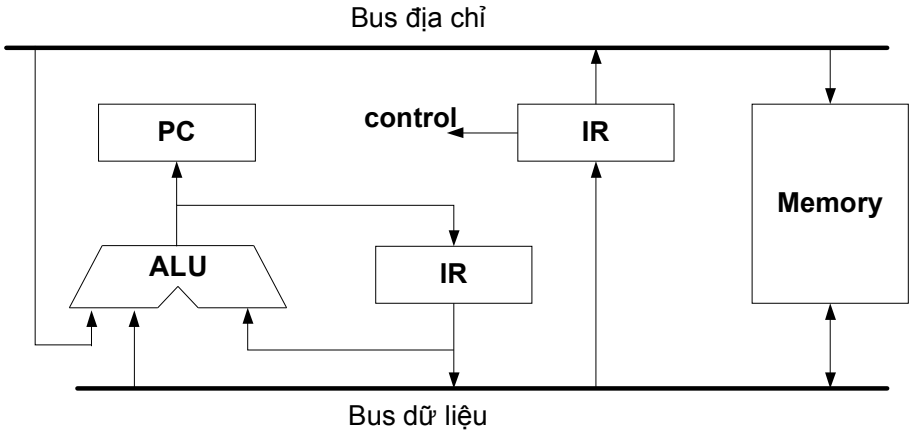
Lõi xử lý MU0 được phát triển đầu tiên và là lõi xử lý đơn giản, có tập lệnh dài 16 bit, với 12 bit địa chỉ và 4 bit mã hóa.

Cấu trúc tập lệnh lõi MU0 có dạng:

4 bits	12 bits
opcode	**S**

Hình 1.2: **Cấu trúc chuẩn cho tập lệnh của MU0.**

Mô hình thiết kế đường truyền dữ liệu đơn giản của lõi xử lý MU0 được mô tả trong hình 1.3. Việc thiết kế ở cấp chuyển đổi mức thanh ghi (RTL): Bộ đếm chương trình (PC) chỉ đến tập lệnh cần thực thi, nạp vào thanh ghi lệnh (IR), giá trị chứa trong IR chỉ đến vùng địa chỉ ô nhớ, nhận giá trị, kết hợp với giá trị đang chứa trong thanh ghi tích lũy (ACC) qua đơn vị xử lý số học (ALU) để tạo giá trị mới, chứa vào ACC. Mỗi một lệnh như vậy, tùy vào số lần truy cập ô nhớ mà tốn số chu kỳ xung nhịp tương đương. Sau mỗi lệnh thực thi, PC sẽ được tăng thêm.

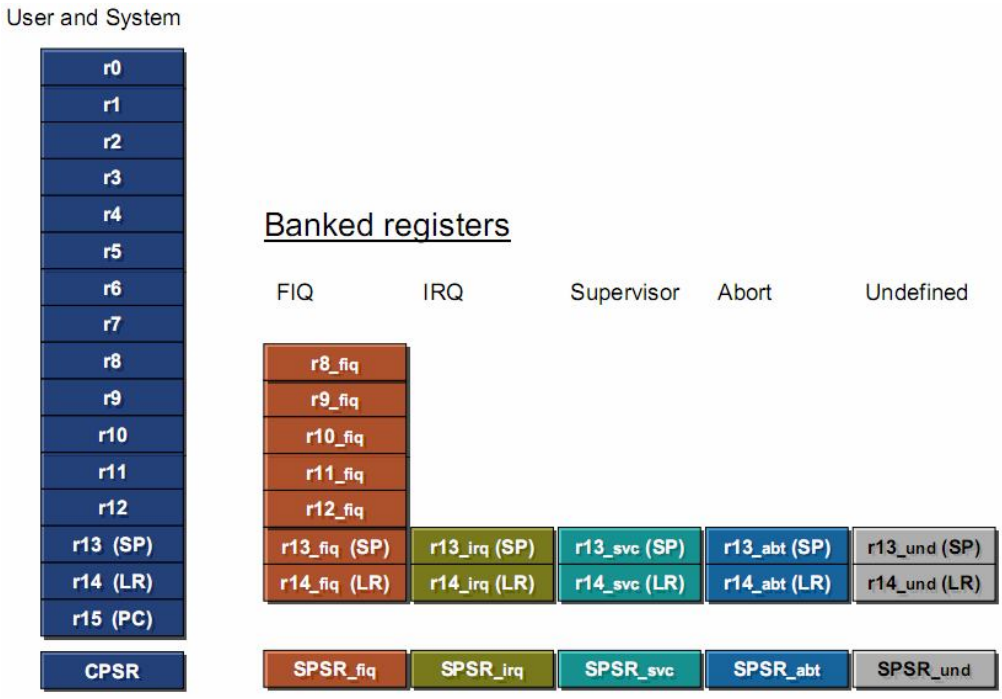


Hình 1.3: **Đường truyền dữ liệu của lõi xử lý MU0.**

1.4.2 Các thanh ghi của ARM

Để phục vụ mục đích của người dùng: r0 ÷ r14 là 15 thanh ghi đa dụng, r15 là thanh ghi Program Counter (PC), thanh ghi trạng thái chương trình hiện tại (CPSR - Current Program Status Register). Các thanh ghi khác được giữ lại cho hệ thống (như thanh ghi chứa các ngắt).

Các thanh ghi của ARM được mô tả trong hình 1.4.



Hình 1.4: **Mô hình các thanh ghi của ARM.**

- Thanh ghi CPSR được người dùng sử dụng chứa các bit điều kiện.
- N: Negative - cờ này được bật khi bit cao nhất của kết quả xử lý ALU bằng 1.
- Z: Zero - cờ này được bật khi kết quả cuối cùng trong ALU bằng 0.
- C: Carry - cờ này được bật khi kết quả cuối cùng trong ALU lớn hơn giá trị 32 bit và tràn.

- V: Overflow - cờ báo tràn sang bit dấu.

- Thanh ghi SPSR (Save Program Status Register) dùng để lưu giữ trạng thái của thanh ghi CPSR khi thay đổi chế độ.

1.5 Cấu trúc load-store

Cũng như hầu hết các bộ xử lý dùng tập lệnh RISC khác, ARM cũng sử dụng cấu trúc load-store. Điều đó có nghĩa là: tất cả các lệnh (cộng, trừ…) đều được thực hiện trên thanh ghi. Chỉ có lệnh sao chép giá trị từ bộ nhớ vào thanh ghi (load) hoặc chép lại giá trị từ thanh ghi vào bộ nhớ (store) mới có ảnh hưởng tới bộ nhớ.

Các bộ xử lý CISC cho phép giá trị trên thanh ghi có thể cộng với giá trị trong bộ nhớ, đôi khi còn cho phép giá trị trên bộ nhớ có thể cộng với giá trị trên thanh ghi. ARM không hỗ trợ cấu trúc lệnh dạng từ bộ nhớ đến bộ nhớ. Vì thế, tất cả các lệnh của ARM thuộc một trong ba loại sau:

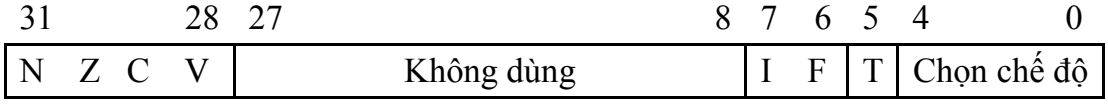
- Lệnh xử lý dữ liệu: chỉ thay đổi giá trị trên thanh ghi.
- Lệnh load-store: sao chép giá trị từ thanh ghi vào bộ nhớ và sao chép giá trị từ bộ nhớ vào thanh ghi.
- Lệnh điều khiển dòng lệnh: bình thường, ta thực thi các lệnh chứa trong một vùng nhớ liên tiếp, tập lệnh điều khiển dòng lệnh cho phép chuyển sang các địa chỉ khác nhau khi thực thi lệnh, tới những nhánh cố định (lệnh rẽ nhánh) hoặc là lưu và trở lại địa chỉ để phục hồi chuỗi lệnh ban đầu (lệnh rẽ nhánh và kết nối) hay là đè lên vùng mã của hệ thống.

1.6 Cấu trúc tập lệnh của ARM

1.6.1 Thực thi lệnh có điều kiện

ARM cung cấp khả năng thực hiện một cách có điều kiện hầu hết các lệnh dựa trên tổ hợp trạng thái của các cờ điều kiện trong thanh ghi CPSR.

Thanh ghi CPSR cho biết trạng thái của chương trình hiện tại và được mô tả trong hình 1.5.



Hình 1.5: **Vị trí các bit trên thanh ghi CPSR.**

1.6.2 Phương thức định địa chỉ

Đối với những lệnh xử lý dữ liệu chỉ có hai phương thức là trực tiếp thanh ghi và giá trị trực tiếp.

Đối với những lệnh load và store thì phương thức địa chỉ là gián tiếp các thanh ghi (không có phương thức trực tiếp bộ nhớ).

1.6.3 Ngăn xếp

ARM hỗ trợ việc lưu và phục hồi giá trị nhiều thanh ghi, gồm hai lệnh:

- LDM : Load multiple register.
- STM : Store multiple register.

Việc lưu hoặc phục hồi giá trị thanh ghi với bộ nhớ bắt đầu từ địa chỉ được lưu trong thanh ghi nền, giá trị của thanh ghi nền có thể giữ nguyên hoặc được cập nhật.

Thứ tự địa chỉ bộ nhớ sao lưu các thanh ghi tăng hoặc giảm tùy theo phương thức định địa chỉ.

1.6.4 Tập lệnh ARM

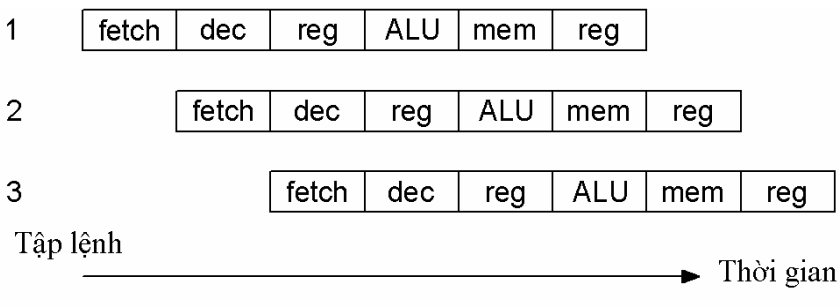
Tất cả lệnh của ARM đều là 32 bit:

- Có cấu trúc dạng load-store.
- Cấu trúc lệnh định dạng ba địa chỉ (nghĩa là địa chỉ của hai toán hạng nguồn và toán hạng đích đều là các địa chỉ riêng biệt).
- Mỗi một lệnh thực thi một điều kiện.
- Có cả lệnh load-store nhiều thanh ghi đồng thời.
- Có khả năng dịch bit kết hợp với thực thi lệnh ALU trong chỉ một chu kỳ máy.
- Chế độ Thumb code: là một chế độ đặc biệt của ARM dùng để tăng mật độ mã bằng cách nén lệnh 32 bit thành 16 bit. Một phần cứng đặc biệt sẽ giải nén lệnh Thumb 16 bit thành lệnh 32 bit.

ARM hỗ trợ sáu kiểu dữ liệu:

- 8 bit có dấu và không dấu.
- 16 bit có dấu và không dấu.
- 32 bit có dấu và không dấu.
- Các toán tử của ARM có 32 bit, khi làm việc với các dữ liệu ngắn hơn, các bit cao của toán tử sẽ được thay thế bằng bit ‘0’.

Cách tổ chức và thực thi tập lệnh của ARM:



Hình 1.6: **Chu kỳ thực thi lệnh theo kiến trúc đường ống.**

Cách tổ chức của lõi ARM không thay đổi nhiều từ năm 1983 ÷ 1995, đều sử dụng tập lệnh có kiến trúc đường ống ba tầng. Từ 1995 trở về đây, ARM đã giới thiệu một số lõi mới có sử dụng kiến trúc đường ống chín tầng.

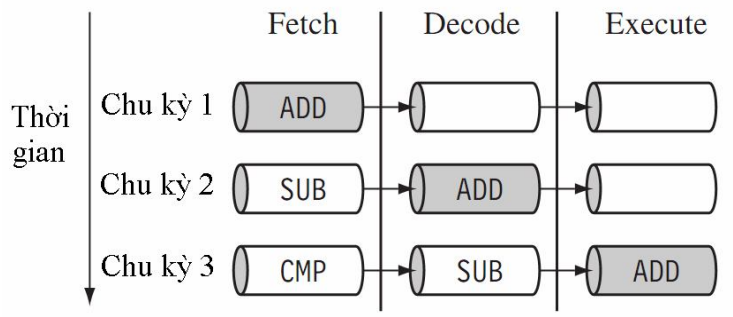
Chu kỳ thực thi lệnh theo kiến trúc đường ống được mô tả trong hình 1.6.

Các bước thực thi lệnh gồm:

- Nhận lệnh từ bộ nhớ (fetch);
- Giải mã lệnh, xác định các tác động cần có và kích thước lệnh (decode);
- Truy cập các toán hạng có thể được yêu cầu từ thanh ghi (reg);
- Kết hợp với toán hạng đấy để tạo thành kết quả hay địa chỉ bộ nhớ (ALU);
- Truy cập vào bộ nhớ cho toán hạng dữ liệu nếu cần thiết (mem);
- Viết kết quả ngược lại thanh ghi (res).

Kiến trúc đường ống

Kiến trúc đường ống là kiến trúc cơ bản trong vi điều khiển ARM, hình 1.7 mô tả kiến trúc đường ống ba tầng để minh họa các bước thực thi lệnh: fetch – decode – excute (nhận lệnh – giải mã – thực thi).



Hình 1.7: **Kiến trúc đường ống ba tầng.**

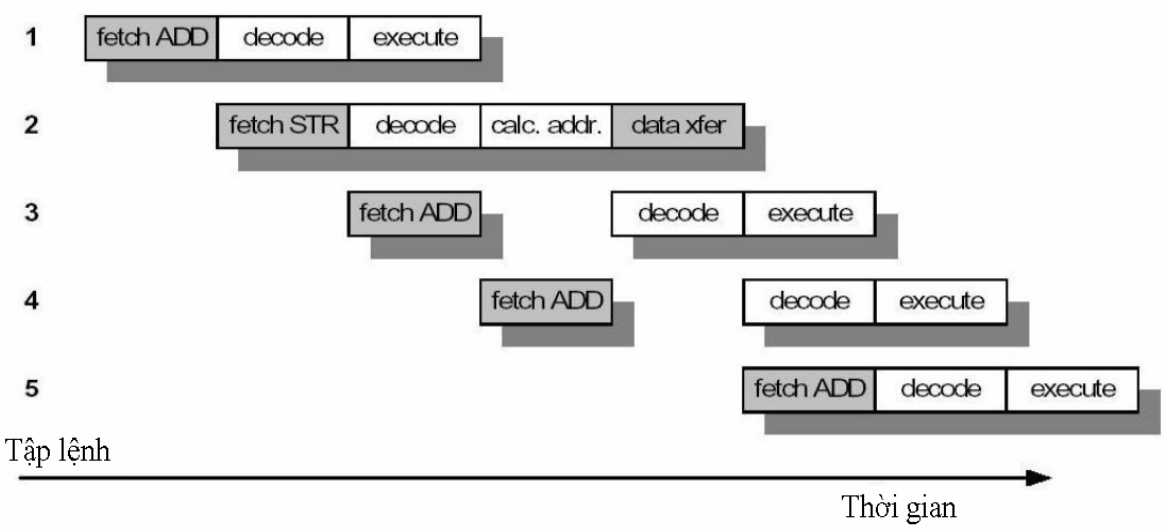
Trong kiến trúc đường ống, khi một lệnh đang được thực thi thì lệnh thứ hai đang được giải mã và lệnh thứ ba bắt đầu được nạp từ bộ nhớ. Với kỹ thuật này thì tốc độ xử lý tăng lên rất nhiều trong một chu kỳ máy.

Trong hình 1.7 cho ta thấy được một chuỗi ba lệnh được nạp, giải mã, và thực thi bởi bộ xử lý. Mỗi lệnh có một chu trình duy nhất để hoàn thành sau khi đường ống được lấp đầy.

Tập lệnh được đặt vào các đường ống liên tục. Trong chu kỳ đầu tiên lõi xử lý nạp lệnh ADD (cộng) từ bộ nhớ. Trong chu kỳ thứ hai lõi tìm nạp các lệnh SUB (trừ) và giải mã lệnh ADD. Trong chu kỳ thứ ba, cả hai lệnh SUB và ADD được di chuyển dọc theo đường ống. Lệnh ADD được thực thi, lệnh SUB được giải mã, và lệnh CMP (so sánh) được nạp. Quá trình này được gọi là lấp đầy đường ống. Kiến trúc đường ống cho phép lõi xử lý thực hiện lệnh trong mỗi chu kỳ.

Khi tăng chiều dài đường ống, số lượng công việc thực hiện ở từng công đoạn giảm, điều này cho phép bộ xử lý phải đạt được đến một tần số hoạt động cao hơn để tăng hiệu suất thực thi. Thời gian trễ của hệ thống cũng sẽ tăng lên bởi vì có nhiều chu kỳ hơn để lấp đầy đường ống trước khi lõi xử lý có thể thực thi một lệnh. Chiều dài đường ống tăng lên cũng có nghĩa là dữ liệu cũng có thể sẽ phải phụ thuộc giữa các công đoạn nhất định.

ARM giới thiệu và đưa ra kiến trúc đường ống có năm tác vụ, với vùng nhớ dữ liệu và chương trình riêng biệt. Từ kiến trúc lệnh có ba tác vụ được chia nhỏ lại thành năm tác vụ cũng làm cho mỗi chu kỳ xung nhịp sẽ thực hiện một công việc đơn giản hơn ở mỗi công đoạn, cho phép có thể tăng chu kỳ xung nhịp của hệ thống. Sự tách rời bộ nhớ chương trình và bộ nhớ dữ liệu cũng cho phép giảm đáng kể tài nguyên chiếm của mỗi lệnh trong một chu kỳ máy.



Hình 1.8: **Kiến trúc đường ống ba tầng trong tập lệnh có nhiều chu kỳ máy.**

Thời gian để bộ xử lý thực thi một chương trình được tính bởi công thức:

$$T_{pro} = \frac{N_{inst}\ x\ \ CPI}{f_{clk}}$$

Trong đó:

- CPI là số xung nhịp trung bình cần cho mỗi lệnh;
- N_{inst} là số lệnh thực thi một chương trình (cố định);
- f_{clk} là tần số xung nhịp.

Với công thức trên thì có hai cách để giảm thời gian thực thi một chương trình:

- Tăng tần số xung nhịp: điều này đòi hỏi trạng thái của mỗi nhiệm vụ trong dòng chảy lệnh đơn giản, và do đó số tác vụ sẽ tăng thêm.

- Giảm CPI: điều này đòi hỏi mỗi lệnh cần nhiều dòng chảy lệnh hơn với tác vụ không đổi, hoặc các tác vụ cần đơn giản hơn, hoặc kết hợp cả hai lại với nhau.

1.7 Kết luận

Chương 1 trình bầy những khái quát cơ bản của vi điều khiển ARM, qua đó ta nắm được quá trình phát triển và hình thành vi điều khiển ARM, mô hình kiến trúc của vi điều khiển ARM và cấu trúc tập lệnh với rất nhiều ưu điểm như: tập lệnh 32 bit, cấu trúc load-store, cách tổ chức và thực thi tập lệnh của ARM dưới dạng kiến trúc đường ống và tập lệnh trực giao, hầu hết tất cả các lệnh đều cho phép thực thi có điều kiện và thực thi trong một chu kỳ đơn. Với các đặc tính kỹ thuật đặc trưng này thì vi điều khiển ARM là một trong những vi điều khiển có tốc độ xử lý, hiệu suất thực thi cao và khả năng tiêu thụ năng lượng ít nhất vào thời điểm hiện nay.

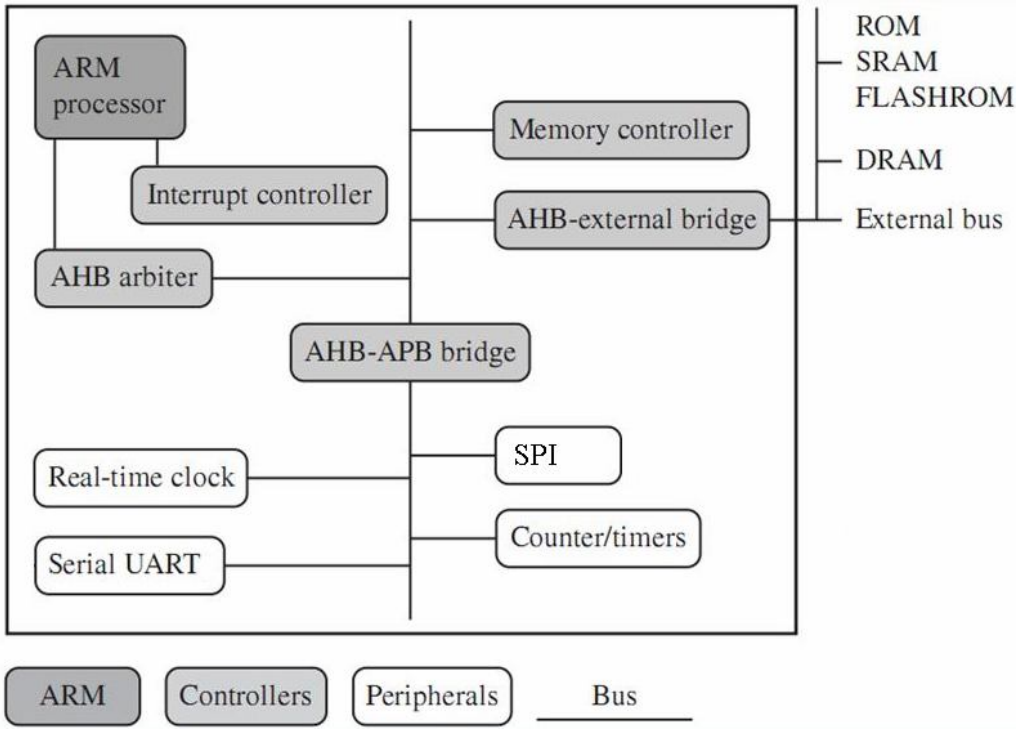
CHƯƠNG 2
GIAO TIẾP VỚI VI ĐIỀU KHIỂN ARM

2.1 Mô hình giao tiếp trong vi điều khiển ARM

Vi điều khiển ARM là một hệ thống có chứa lõi vi xử lý ARM với các giao tiếp hỗ trợ bên trong [6].

Vi điều khiển ARM được thực thi trên hệ thống kiến trúc các bus truyền dữ liệu đa chức năng của vi điều khiển. Bao gồm bộ xử lý ARM kết nối qua hệ thống bus truyền dữ liệu hiệu suất cao để đồng bộ nhanh với SRAM, các bus giao tiếp ngoài, và cầu nối tới các bus truyền ngoại vi công suất thấp, được mô tả trong hình 2.1.

Thiết bị ngoại vi bên ngoài được xây dựng từ các thiết bị riêng và tùy theo ứng dụng người dùng.



Hình 2.1: **Mô hình giao tiếp trong vi điều khiển ARM.**

Các khối chức năng trong vi điều khiển ARM bao gồm:

- Bộ xử lý ARM;
- Bộ điều khiển ngắt;
- Bộ phân xử bus truyền hiệu suất cao (AHB - Advanced High-performance Bus);
- Bộ điều khiển bộ nhớ;
- SRAM;
- EPROM hoặc Flash;

- DRAM;
- Cầu nối AHB – APB (Advanced Peripheral Bus: Bus truyền ngoại vi tối ưu)
- Cầu nối ngoài AHB;
- Bộ đếm/định thời;
- Khối SPI (Serial Peripheral Interface): Khối giao tiếp các thiết bị ngoại vi nối tiếp;
- Khối Serial UART (Serial Universal Asynchronous Receiver/Transmitter): Khối giao tiếp nối tiếp truyền/thu không đồng bộ đa năng.

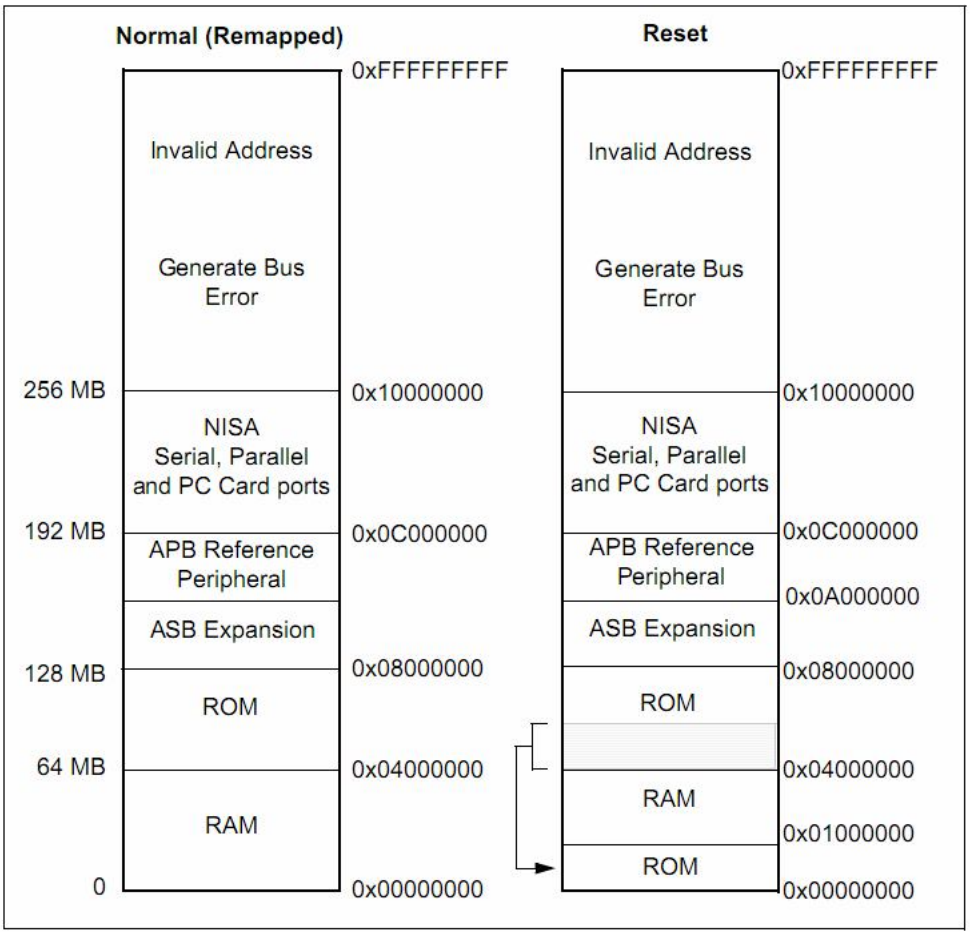
2.2 Các giao tiếp cơ bản trong vi điều khiển ARM

2.2.1 Giao tiếp với bộ nhớ

Giao tiếp với bộ nhớ trong vi điều khiển ARM [7] có tính năng truy xuất dữ liệu rất nhanh.

Trong vi điều khiển ARM, bộ nhớ nội bộ có thể có các dạng bộ nhớ như: SSRAM, SRAM, DRAM, EPROM/Flash.

Bản đồ bộ nhớ chính được mô tả trong hình 2.2:



Hình 2.2: **Sự phân tách hai trạng thái trên bản đồ bộ nhớ.**

Bản đồ bộ nhớ có hai trạng thái:

- Trạng thái Reset.

- Trạng thái thông thường: sau khi đã được ánh xạ các thanh ghi định địa chỉ vào.

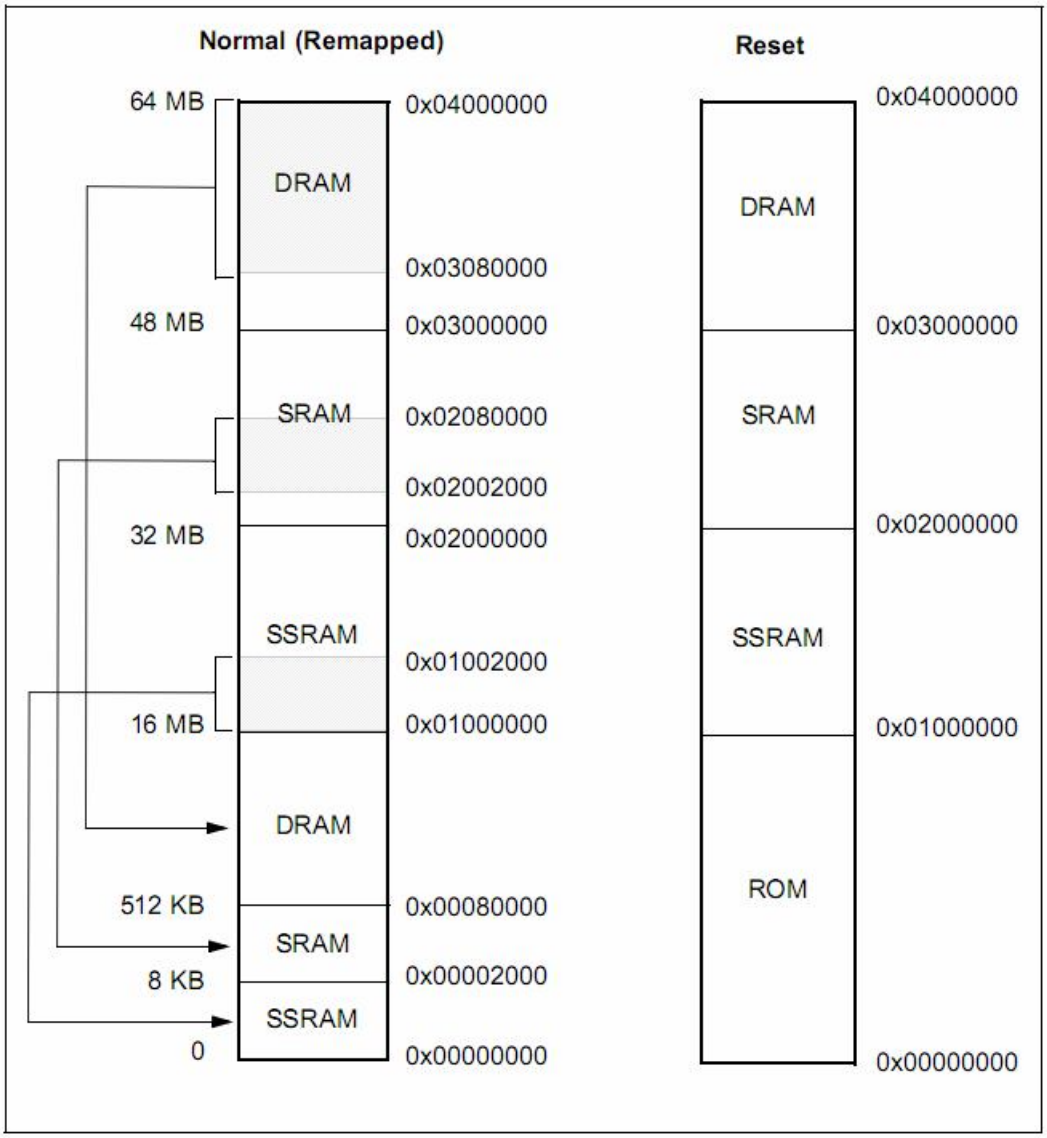
Trong cấu hình thông thường (đã được định địa chỉ), ký hiệu vùng RAM là từ địa chỉ 0x0 đến 0x04000000.

Trong cấu hình Reset, ROM được ánh xạ vào không gian được ký hiệu với khả năng truy cập RAM ở địa chỉ cao hơn.

Khi truy cập vào không gian bộ nhớ 0x10000000, bộ xử lý sẽ hủy bỏ các trường hợp ngoại lệ (sai địa chỉ).

Vùng RAM

Vùng RAM được chia thành bốn khối chính được mô tả trong hình 2.3. Từ phần 16MB dành riêng cho DRAM, SRAM, SSRAM.



Hình 2.3: **Vùng RAM.**

Khối 16MB (khởi điểm ở 0x0) là vùng ROM, có các điểm ký hiệu tới cấu hình cơ sở của ROM, hoặc có các vùng xếp chồng lên RAM (trong cấu hình thông thường).

Bảng địa chỉ trên vùng RAM trong cấu hình thông thường và Reset được mô tả trong bảng 2.1

Bảng 2.1: **Các địa chỉ trên vùng RAM.**

Cấu hình thông thường		**Cấu hình Reset**	
Địa chỉ	**Địa chỉ tương ứng**	**Địa chỉ**	**Địa chỉ tương ứng**
0x00000000	0x01000000	0x00000000	0x04000000
0x00002000	0x02002000		
0x00080000	0x03080000		

Trong liên kết định địa chỉ, theo mặc định khi ở cấu hình Reset, EPROM và Flash ở dưới đáy của địa chỉ bản đồ nhớ. Nếu thanh ghi định địa chỉ được ghi vào, tín hiệu định địa chỉ sẽ là mức cao và bộ giải mã chuyển sang bản đồ bộ nhớ thông thường, khi đó RAM sẽ ở dưới cùng.

Nếu khi hoạt động mà hệ thống không có EPROM hay Flash, hoặc nếu muốn khởi động từ RAM, phải gỡ liên kết định địa chỉ. Tín hiệu định địa chỉ sẽ luôn ở mức cao, và việc khởi động với SSRAM sẽ ở dưới đáy của địa chỉ bản đồ nhớ.

Các bộ nhớ trong vùng RAM có chức năng:

- SSRAM

Đồng bộ SRAM (SSRAM) được dùng để cung cấp bộ nhớ chu kỳ đơn. Thiết bị SSRAM tổ chức dưới dạng 32KB x 32 bit. Vùng này được sử dụng cho các trình giới hạn thời gian, như là các bộ điều khiển ngắt. SSRAM thường ở địa chỉ 0, cấu hình của bộ nhớ này là không bắt buộc.

- SRAM

SRAM được dùng để thể hiện các giản đồ bộ nhớ khác nhau. SRAM cho phép điều khiển bốn mức vật lý 128KB x 8 bit để chia thành hai dãy logic của mỗi một mức 256KB. Mỗi một dãy logic này có thể được định cấu hình 8, 16, hoặc 32 bit bộ nhớ mở rộng. SRAM mô phỏng bộ nhớ hệ thống này bằng cách chèn chính xác số trạng thái chờ.

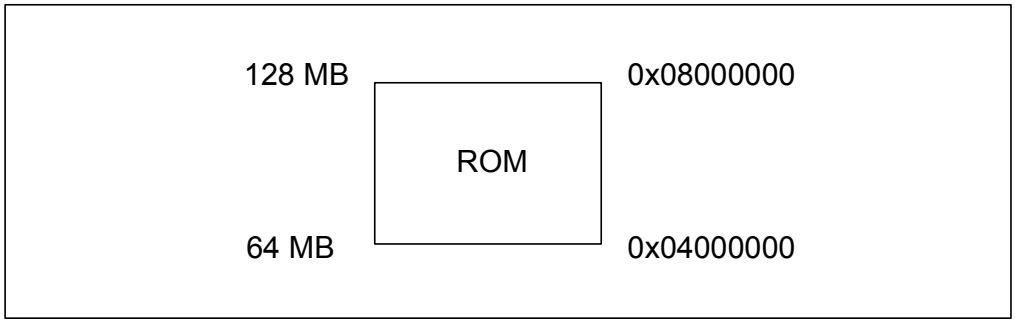
- DRAM

DRAM cung cấp:

- Hỗ trợ chế độ tuần tự truy cập;
- Hỗ trợ các bước chuyển byte, halfword và word;
- Bộ điều khiển làm tươi DRAM;
- Tự động định lại cấu hình kích thước khối.

Vùng ROM

Có một vùng được dành riêng cho ROM. Vùng ROM giống như cả hai cấu hình thông thường và cấu hình Reset của bản đồ bộ nhớ, vùng ROM được mô tả như trong hình 2.4.



Hình 2.4: **Vùng ROM.**

Khi Reset vi điều khiển ARM, ROM được xác định ở vị trí 0x0 và khi chuyển sang trạng thái thông thường thì RAM sẽ ở vị trí dưới cùng của bản đồ địa chỉ.

EPROM/Flash

EPROM/Flash có hai dạng, một cho 8 bit bộ nhớ mở rộng và một cho 16 bit bộ nhớ mở rộng. Các bộ nhớ này có thể được truy cập cùng theo chuẩn EPROM hoặc theo chuẩn 5V Flash.

2.2.2 Giao tiếp với bộ điều khiển ngắt

Bộ điều khiển ngắt trong ARM [8] cung cấp giao tiếp phần mềm độc lập cho hệ thống ngắt. Các bit ngắt được định nghĩa cho yêu cầu chức năng cơ bản trong thiết kế hệ thống.

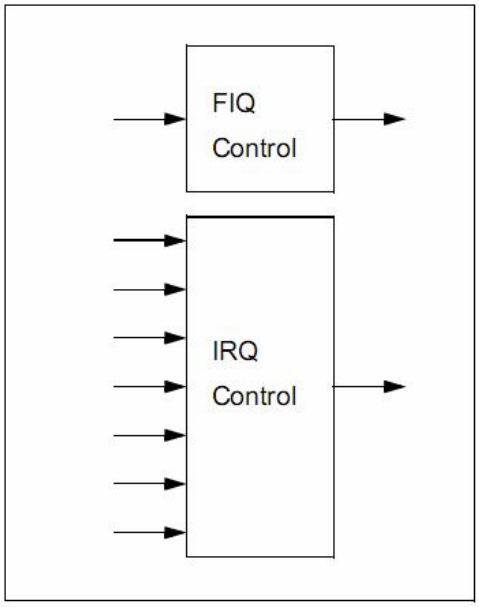
Trong hệ thống ARM có hai mức ngắt:

- FIQ (Fast Interrupt Request) dành cho ngắt nhanh.

- IRQ (Interrupt Request) dành cho các ngắt chung.

FIQ được sử dụng ở tại bất kỳ thời điểm nào. Nó cung cấp ngắt với thời gian trễ thấp, giống như là một nguồn độc lập đảm bảo chương trình phục vụ ngắt có thể thực thi trực tiếp mà không cần sự quyết định của nguồn ngắt chính. Điều đó làm giảm bớt thời gian trễ ngắt như là các thanh ghi đặc biệt, các thanh ghi này dành cho các ngắt FIQ và dùng để tối đa hiệu suất.

Các bộ điều khiển ngắt được chia ra để sử dụng cho FIQ và IRQ, chỉ khác ở vị trí một bit riêng được định rõ cho bộ điều khiển FIQ, các bit còn lại trong bộ điều khiển ngắt này dành cho nguồn ngắt độc lập trong 32 bit của bộ điều khiển IRQ và được mô tả trong hình 2.5.



Hình 2.5: **Các bộ điều khiển ngắt FIQ và IRQ.**

Bộ điều khiển ngắt sử dụng vị trí bit cho mỗi nguồn ngắt khác nhau và mỗi vị trí được định rõ bởi phần mềm chương trình ngắt, các kênh truyền thông và các bộ định thời. Bit 0 không được xác định trong bộ điều khiển IRQ, mặc dù vậy nó vẫn có thể được sử dụng chung tương tự nguồn ngắt như trong bộ điều khiển FIQ.

Tất cả nguồn ngắt được đưa vào sẽ hoạt động ở mức cao, bất kỳ yêu cầu đảo hay chốt tác động đến phải được cung cấp tại nguồn ngắt chung.

Quyền ưu tiên trong sơ đồ phần cứng không được cung cấp, cũng không có bất kỳ dạng quyền ưu tiên véc tơ ngắt, tất cả các chức năng này có thể được cung cấp trong phần mềm.

Thanh ghi ngắt chương trình cũng cung cấp tới ngắt chung dưới sự điều khiển của phần mềm. Đây là dạng sử dụng để giảm cấp từ ngắt FIQ thành ngắt IRQ.

Bộ điều khiển ngắt cung cấp các trạng thái nguồn ngắt, trạng thái yêu cầu ngắt và thanh ghi cho phép ngắt. Thanh ghi cho phép ngắt được dùng để quyết định một nguồn ngắt hoạt động nếu sinh ra một yêu cầu ngắt để hệ thống xử lý.

Trạng thái nguồn ngắt chỉ xác định nếu nguồn ngắt tương thích ưu tiên hoạt động. Nguồn ngắt sẽ hoạt động ở mức cao, do đó một mức logic cao trong thanh ghi trạng thái nguồn chỉ báo nguồn ngắt hoạt động.

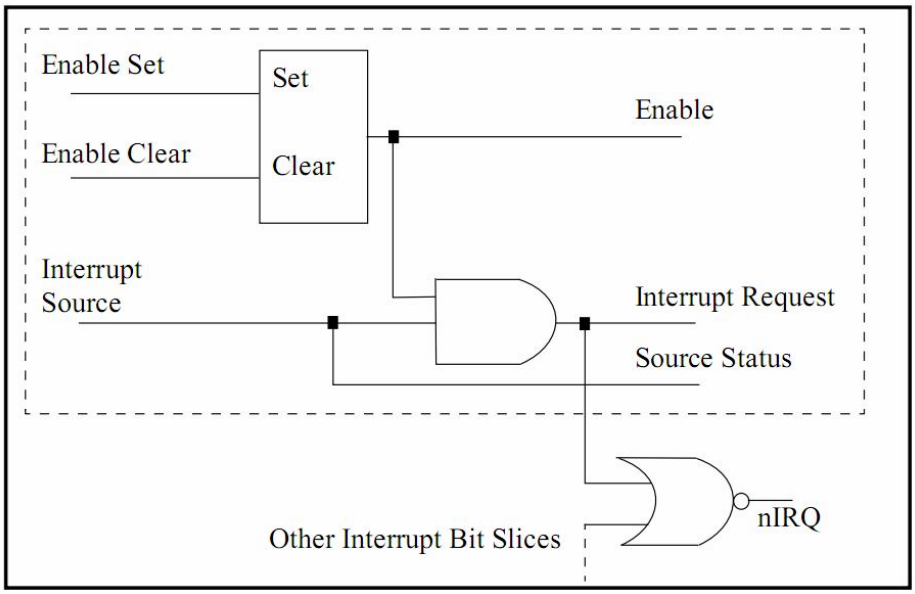
Trạng thái yêu cầu ngắt được xác định nếu nguồn ngắt sinh ra một yêu cầu ngắt tới bộ xử lý.

Thanh ghi cho phép ngắt có cơ chế kép cho việc thiết lập và xóa các bit cho phép. Các bit thiết lập hay xóa này là độc lập và không liên quan gì đến các bit khác trong thanh ghi cho phép.

Khi ghi vào vị trí thiết lập cho phép ngắt, mỗi một bit dữ liệu sẽ được thiết lập ở mức cao tương ứng với bit trong thanh ghi cho phép và tất cả các bit khác trong thanh

ghi cho phép không bị ảnh hưởng. Khi ghi vào vị trí xóa trên thanh ghi cho phép nghĩa là dùng bit xóa đảo lại, không dùng các bit khác.

Một kênh của bộ điều khiển ngắt được mô tả trong hình 2.6.



Hình 2.6: **Sơ đồ một kênh của bộ điều khiển ngắt.**

Bộ điều khiển ngắt FIQ được dành riêng bit 0. Bộ điều khiển IRQ có số lượng lớn hơn, kích thước của bộ điều khiển này phụ thuộc vào hệ thống xử lý.

Các thanh ghi điều khiển ngắt:

Các thanh ghi sau được quy định cho cả hai bộ điều khiển ngắt FIQ và IRQ

- Thanh ghi cho phép
  - Trạng thái chỉ đọc.
  - Thanh ghi cho phép được dùng để chắc chắn nguồn ngắt đầu vào và xác định nguồn ngắt đầu vào được tác động sẽ tạo ra một yêu cầu ngắt đến bộ xử lý. Thanh ghi này là chỉ đọc và các giá trị của nó chỉ có thể thay đổi bởi các vị trí thiết lập hoặc xóa. Nếu các bit trong bộ điều khiển ngắt chưa được kích hoạt (thiết lập hoặc xóa) thì có nghĩa là các bit trong thanh ghi cho phép đó sẽ đọc nhưng với trạng thái không xác định.
  - Bit cho phép là 1 chỉ báo rằng ngắt đã được kích hoạt và sẽ cấp một yêu cầu ngắt tới bộ xử lý. Bit kích hoạt là 0 chỉ báo rằng ngắt được xóa. Trạng thái khởi động lại sẽ xóa bỏ tất cả các ngắt.

- Thanh ghi cho phép thiết lập:
  - Trạng thái chỉ ghi.
  - Vị trí này dùng để thiết lập các bit trong thanh ghi cho phép ngắt. Khi ghi vào vị trí này bit dữ liệu mức cao sẽ sinh ra bit tương ứng trong thanh ghi cho phép được thiết lập. Các bit dữ liệu thấp không ảnh hưởng đến bit tương ứng trong thanh ghi cho phép.

- Thanh ghi cho phép xóa:
  - Trạng thái chỉ đọc.
  - Vị trí này dùng để xóa các bit trong thanh ghi cho phép ngắt. Khi ghi vào thanh ghi này bit dữ liệu mức cao sẽ sinh ra bit tương ứng trong thanh ghi cho phép được xóa. Các bit dữ liệu thấp không ảnh hưởng đến bit tương ứng trong thanh ghi cho phép ngắt.

- Thanh ghi trạng thái nguồn:
  - Trạng thái chỉ đọc.
  - Vị trí này cung cấp tình trạng của các nguồn ngắt tới bộ điều khiển ngắt. Bit cao chỉ báo rằng một yêu cầu ngắt thích hợp là đảm bảo được quyền ưu tiên hoạt động.

- Thanh ghi yêu cầu ngắt:
  - Trạng thái chỉ đọc.
  - Vị trí này cung cấp tình trạng của nguồn ngắt. Bit cao chỉ báo rằng ngắt hoạt động và sẽ tạo ra một ngắt đưa tới bộ xử lý.

- Thanh ghi ngắt chương trình IRQ:
  - Trạng thái chỉ ghi.
  - Khi ghi vào thanh ghi này sẽ thiết lập hoặc xóa một chương trình ngắt. Ghi vào thanh ghi này bit 1 ở phần cao sẽ tạo ra một chương trình ngắt, trong khi ghi vào thanh ghi này bit 1 ở phần thấp sẽ xóa chương trình ngắt. Giá trị thanh ghi này sẽ quyết định bởi việc đọc bit 1 của thanh ghi trạng thái nguồn. Không sử dụng bit 0 trong thanh ghi này.

Một số thanh ghi dành riêng cho kiểm tra. Các thanh ghi này không được truy cập trong quá trình hoạt động thông thường.

Các bit được định nghĩa trong bộ điều khiển ngắt

Bộ điều khiển ngắt FIQ gồm một bit (bit 0).

Bit 1 đến bit 5 trong bộ điều khiển ngắt được định nghĩa như trong bảng 2.2. Bit 6 trở lên đến bit 31 tùy theo yêu cầu sử dụng.

Bảng 2.2: **Các bit định nghĩa trong bộ điều khiển ngắt.**

Bit	Nguồn ngắt
0	FIQ source
1	Programmed Interrrupt
2	Comms Rx
3	Comms Tx
4	Timer 1
5	Timer 2

Bản đồ nhớ bộ điều khiển ngắt được mô tả trong bảng 2.3.

Bảng 2.3: **Bản đồ nhớ bộ điều khiển ngắt.**

Address	Read Location	Write Location
IntBase + 0x00	IRQStatus	Reserved
IntBase + 0x04	IRQRawStatus	Reserved
IntBase + 0x08	IRQEnable	IRQEnableSet
ntBase + 0x0C	Reserved	IRQEnableClear
IntBase + 0x10	Reserved	IRQSoft
IntBase + 0x100	FIQStatus	Reserved
IntBase + 0x104	FIQRawStatus	Reserved
IntBase + 0x108	FIQEnable	FIQEnableSet
IntBase + 0x10C	Reserved	FIQEnableClear

Địa chỉ cơ sở của bộ điều khiển ngắt không cố định và có thể khác nhau đối với mỗi hệ thống xử lý cụ thể. Tuy nhiên, khoảng cách của các thanh ghi từ địa chỉ cơ sở được cố định.

2.2.3 Giao tiếp với bộ định thời

Giới thiệu chung

Có tối thiểu hai bộ định thời trong một hệ thống ARM [8], mặc dù được định nghĩa như vậy nhưng có thể dễ dàng mở rộng thêm các bộ định thời. Cùng với nguyên tắc mở rộng đơn giản là tác động tới cấu trúc thanh ghi sẽ cung cấp thêm các bộ định thời cho sử dụng từ việc lập trình.

Mỗi một bộ định thời là một bộ đếm ngược rộng 16 bit, có thể lựa chọn phân chia tần số đầu vào. Mạch đếm cho phép xung hệ thống được sử dụng trực tiếp, hoặc xung được chia bởi 16, 256 hoặc 1024 tùy theo sử dụng. Việc phân chia này được cung cấp bởi các bậc 0, 4, 8 hoặc 10 của bộ chia tỉ lệ xung.

Bộ định thời có hai chế độ hoạt động:

- Kiểu đếm tự do.
- Kiểu tuần hoàn.

Trong chế độ định thời tuần hoàn bộ đếm sẽ tạo ra một ngắt tại một khoảng thời gian. Trong chế độ định thời tự do, bộ định thời sẽ tràn bộ đếm sau khi đến giá trị 0 và tiếp tục đếm ngược từ giá trị cực đại.

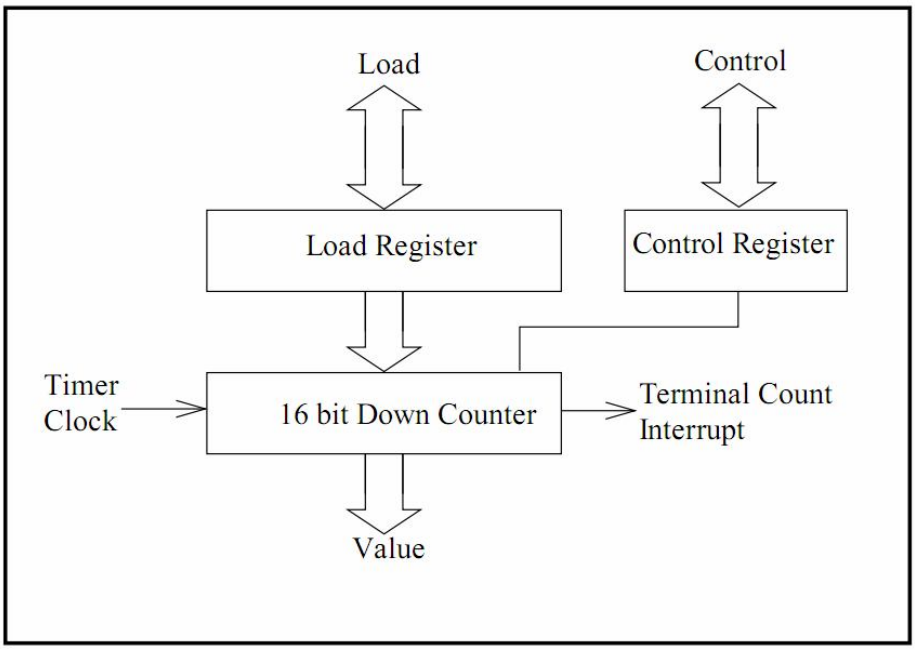
Hoạt động của bộ định thời

Bộ định thời được nạp bởi việc ghi vào thanh ghi nạp, sau đó nếu như được cho phép, bộ định thời sẽ đếm ngược về 0. Trên một hành trình đếm về 0 như vậy, một ngắt sẽ được tạo ra. Ngắt sẽ được xóa bằng cách ghi vào thanh ghi xóa.

Sau hành trình đếm về 0, nếu bộ định thời đang hoạt động ở chế độ định thời tự do thì bộ định thời sẽ tiếp tục đếm giảm từ giá trị cực đại. Nếu đang ở trong chế độ tuần hoàn thì bộ định thời sẽ tải lại từ thanh ghi nạp và tiếp tục đếm giảm. Trong chế độ này bộ định thời sẽ tạo ra một chu kỳ ngắt (ngắt định kỳ). Các chế độ được lựa chọn bởi một bit trong thanh ghi điều khiển.

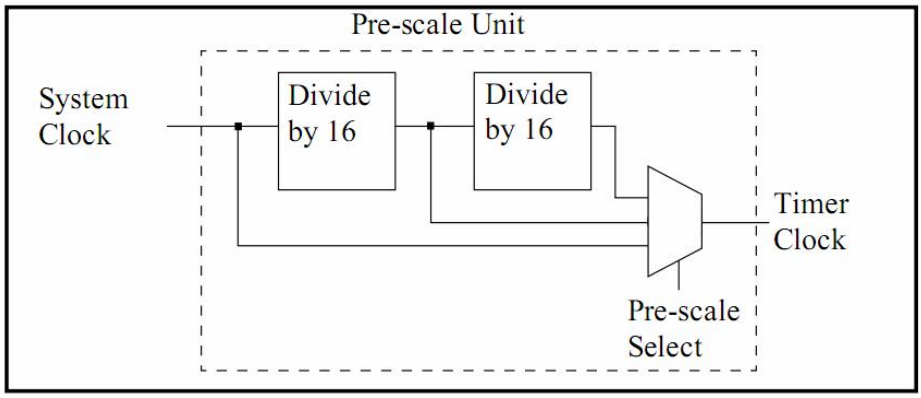
Ở bất kỳ điểm hiện hành nào thì giá trị bộ định thời cũng có thể đọc từ bộ đếm ngược, được mô tả trong hình 2.7.

Bộ định thời được kích hoạt bằng một bit trong thanh ghi điều khiển. Việc khởi động lại sẽ làm bộ định thời được xóa, ngắt sẽ được xóa và thanh ghi nạp sẽ không xác định. Các chế độ và giá trị trong bộ chia tỉ lệ cũng sẽ không xác định.



Hình 2.7: **Giản đồ khối bộ định thời.**

Xung nhịp bộ định thời được tạo ra bởi bộ chia tỉ lệ xung. Xung nhịp bộ định thời có thể là xung hệ thống, xung hệ thống được chia từ bộ chia 16, được tạo ra bởi 4 bit của bộ chia tỉ lệ, hoặc xung hệ thống được chia từ bộ chia 256 được tạo ra bởi tổng hợp 8 bit của bộ chia tỉ lệ, được mô tả trong hình 2.8.



Hình 2.8: **Bộ chia tỉ lệ xung.**

Các thanh ghi trong bộ định thời

- Thanh ghi nạp:
  - Thanh ghi nạp có trạng thái thanh đọc hoặc ghi.
  - Thanh ghi nạp chứa giá trị khởi tạo của bộ định thời và dùng giá trị này để nạp lại trong chế độ định thời tuần hoàn. Khi ghi vào thanh ghi này, 16 bit trên cùng ghi vào là 0 và khi đọc 16 bit trên cùng đó sẽ không xác định.

- Thanh ghi giá trị:
  - Thanh ghi giá trị có trạng thái chỉ để đọc.
  - Vị trí thanh ghi giá trị cho biết giá trị hiện hành của bộ định thời.

- Thanh ghi xóa:
  - Thanh ghi xóa có trạng thái chỉ ghi.
  - Khi ghi vào vị trí thanh ghi xóa sẽ xóa đi một ngắt đã được tạo ra bởi bộ đếm của bộ định thời.

- Thanh ghi điều khiển:
  - Thanh ghi điều khiển có trạng thái đọc hoặc ghi.
  - Thanh ghi điều khiển cung cấp việc cho phép hay loại bỏ các chế độ và cấu hình chia tỉ lệ cho bộ định thời.

- Vị trí các bit trong thanh ghi điều khiển cho bộ định thời được mô tả trong hình 2.9.

31 – 8	7	6	5	4	3 – 2	1	0
0	Cho phép	Chọn chế độ	0	0	Bộ chia tỉ lệ	0	0

Hình 2.9: **Vị trí các bit trong thanh ghi điều khiển.**

- Chức năng các bit trong thanh ghi điều khiển được mô tả trong bảng 2.4.

Bảng 2.4: **Mô tả các bit trong thanh ghi điều khiển cho bộ định thời.**

Các bit 0 – 1, 4 – 5, 8 – 31: không xác định	Ghi giá trị 0 và được đọc như là không xác định
Bit 2, 3 – Các bit dành cho bộ chia tỉ lệ xung	Được mô tả trong bảng 2.5
Bit 6 – Bit lựa chọn chế độ	0 : Chế độ định thời tự do
	1 : Chế độ định thời tuần hoàn
Bit 7 – Bit cho phép	0 : Không cho phép bộ định thời hoạt động
	1 : Cho phép bộ định thời hoạt động

- Chế độ các bit của bộ chia tỉ lệ xung trong thanh ghi điều khiển được mô tả trong bảng 2.5.

Bảng 2.5: **Chế độ các bit của bộ chia tỉ lệ xung trong thanh ghi điều khiển.**

Bit 3	Bit 2	Xung được chia	Các bậc của bộ chia tỉ lệ xung
0	0	1	0
0	1	16	4
1	0	256	8
1	1	1024	10

- Bản đồ nhớ bộ định thời:

Địa chỉ cơ sở của bộ định thời không cố định và có thể khác đối với mỗi hệ thống xử lý cụ thể. Tuy nhiên, khoảng cách của các thanh ghi từ địa chỉ cơ sở được cố định như trong bảng 2.6.

Bảng 2.6: **Bản đồ địa chỉ bộ định thời.**

Address	Read Location	Write Location
TimerBase	Timer1Load	Timer1Load
TimerBase + 0x04	Timer1Value	Reserved
TimerBase + 0x08	Timer1Control	Timer1Control
TimerBase + 0x0C	Reserved	Timer1Clear
TimerBase + 0x10	Reserved	Reserved
TimerBase + 0x20	Timer2Load	Timer2Load
TimerBase + 0x24	Timer2Value	Reserved
TimerBase + 0x28	Timer2Control	Timer2Control
TimerBase + 0x2C	Reserved	Timer2Clear
TimerBase + 0x30	Reserved	Reserved

2.2.4 Giao tiếp với bộ điều khiển tạm dừng và Reset

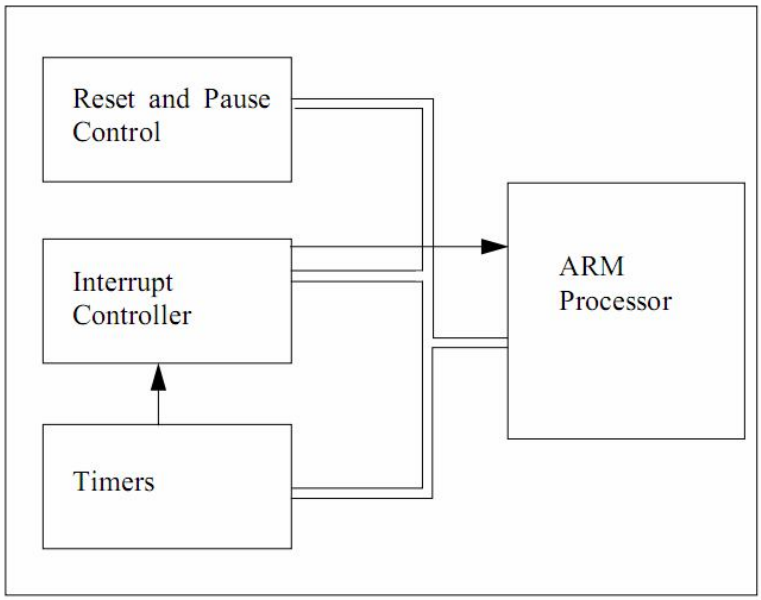
Giới thiệu chung

Bộ điều khiển tạm dừng và Reset là sự kết hợp của bốn chức năng riêng biệt: chức năng tạm dừng, chức năng xác nhận, các trạng thái Reset và bản đồ bộ nhớ Reset [8]. Hình 2.10 mô tả giao tiếp lõi xử lý ARM với bộ điều khiển tạm dừng và Reset, bộ điều khiển ngắt và bộ định thời. Ý tưởng thiết kế các thiết bị ngoại vi này là làm tăng sự đồng nhất trong hệ thống ARM và làm tăng mức linh động của phần mềm giữa các hệ thống.

- Chức năng điều khiển tạm dừng:

Điều khiển tạm dừng là để hệ thống xử lý trong trạng thái công suất thấp như là trạng thái đợi trong lúc ngắt (hệ thống không yêu cầu bộ xử lý hoạt động).

Vị trí xác định tạm dừng là chỉ ghi. Khi ghi vào vị trí tạm dừng, hệ thống sẽ đi vào trạng thái chờ. Thông thường nó sẽ ngăn chặn bộ xử lý tìm nạp lệnh thêm cho đến khi nhận được một ngắt.



Hình 2.10: **Giao tiếp lõi ARM với bộ điều khiển tạm dừng và Reset.**

- Chức năng xác nhận:

Thanh ghi xác nhận cho biết cấu hình hệ thống. Thanh ghi xác nhận là chỉ để đọc. Chỉ duy nhất một bit được cài đặt cho thanh ghi xác nhận, bit 0 dùng để cho biết thêm thông tin.

Bit 0 là bit xác nhận với trạng thái thiết lập:

0 – không có thông tin.

1 – xác nhận thêm thông tin.

Nếu bit dưới cùng của thanh ghi xác nhận được thiết lập, các bit được yêu cầu thêm sau đó sẽ cung cấp chi tiết hơn về thông tin hệ thống.

- Chức năng Reset:

Thanh ghi Reset là chỉ đọc. Chỉ có một bit của thanh ghi này được định nghĩa, đó là bit Reset nguồn. Bit Reset nguồn là bit 0 trong thanh ghi Reset và các giá trị của nó được biểu biễn như sau:

0 – không Reset nguồn.

1 – Reset nguồn.

Ngoài ra các bit trong thanh ghi Reset có thể cung cấp chi tiết hơn về thông tin trạng thái Reset.

Thanh ghi Reset có cơ chế kép dành cho các bit cài đặt và xóa, các bit này hoàn toàn độc lập và khi thay đổi sẽ không nhận các bit khác trong thanh ghi.

Vị trí xóa trạng thái Reset là chỉ ghi. Vị trí này dùng để xóa các cờ trạng thái Reset. Khi ghi vào mỗi một bit dữ liệu vào thanh ghi này ở mức cao sẽ thiết lập bit tương ứng trong thanh ghi Reset là xóa. Khi ghi vào mỗi một bit dữ liệu vào thanh ghi này ở mức thấp sẽ không gây ảnh hưởng đến bit tương ứng trong thanh ghi Reset.

Vị trí Reset không có biến trong đặc tính tham chiếu các thiết bị ngoại tối thiểu, vì bit Reset trạng thái nguồn không thể thiết lập được bằng phần mềm. Thanh ghi này được cài đặt sẵn trong đặc tính kỹ thuật để đảm bảo khả năng của chức năng Reset.

- Xóa bản đồ nhớ của bộ điều khiển Reset:

Vị trí xóa bản đồ Reset là chỉ ghi.

Khi bản đồ bộ nhớ Reset đã xóa và bản đồ nhớ thông thường được dùng, hệ thống sẽ không quay trở lại dùng bản đồ bộ nhớ Reset, ngoại trừ phải trải qua điều kiện thiết lập lại.

Ở trạng thái Reset, hệ thống xử lý sẽ ánh xạ ROM đến vị trí 0, và khi quá trình hoạt động thông thường trở lại, RAM sẽ ở vị trí 0.

- Bản đồ nhớ bộ điều khiển tạm dừng và Reset được mô tả trong bảng 2.7.

Bảng 2.7: **Bản đồ nhớ bộ điều khiển tạm dừng và Reset.**

Address	Read Location	Write Location
RemapBase	Reserved	Pause
RemapBase + 0x10	Identification	Reserved
RemapBase + 0x20	Reserved	ClearResetMap
RemapBase + 0x30	ResetStatus	ResetStatusSet
RemapBase + 0x34	Reserved	ResetStatusClear

Địa chỉ cơ sở của bộ điều khiển tạm dừng và Reset là không cố định và có thể khác nhau đối với mỗi hệ thống xử lý cụ thể. Tuy nhiên, khoảng cách của các thanh ghi từ địa chỉ cơ sở được cố định.

2.2.5 Giao tiếp với khối GIPO

Khối GIPO (General Purpose Input/Output) là khối điều khiển 8 bit đầu vào hoặc ra đa mục đích [9], được kết hợp chặt chẽ với bộ logic có khả năng lập trình (PLD). Bộ PLD giao tiếp qua các bus mở rộng với GPIO.

- Các bit trong GPIO:

Các bit từ 0 – 3 là các bit đầu ra chỉ đọc.

Các bit từ 4 – 7 là các bit được lập trình độc lập như là một đầu vào hoặc là một đầu ra. Các bit trong thanh ghi dữ liệu được thiết lập và xoá được dùng bởi các thanh

ghi GPIO_DATASET và GIPO_DATACLR. Các bit đọc và ghi được dùng bởi thanh ghi GPIO_DATAIN và GPIO_DATAOUT.

- Các thanh ghi GPIO:

GPIO cung cấp các tín hiệu đầu ra và đầu vào đa mục đích. Mỗi một đường GPIO đều có điện trở kéo 10KΩ bên trong lên 3,3V. Các thanh ghi GPIO được chỉ rõ trong bảng 2.8.

Bảng 2.8: **Bảng tổng quát các thanh ghi GPIO.**

Tên thanh ghi	Trạng thái	Độ dài (bit)	Chức năng
GPIO_DATASET	Ghi	8	Thiết lập đầu ra dữ liệu
GPIO_DATAIN	Đọc	8	Đọc các chân đầu vào dữ liệu
GPIO_DATACLR	Ghi	8	Xóa đầu ra thanh ghi dữ liệu
GPIO_DATAOUT	Đọc	8	Đọc các chân đầu ra dữ liệu
GPIO_DIRN	Đọc/Ghi	8	Điều khiển hướng dữ liệu vào/ra

- Thanh ghi thiết lập đầu ra dữ liệu:

Vị trí GPIO_DATASET được dùng để thiết lập đầu ra các bit như sau:

- 1 = thiết lập bit đầu ra GPIO.
- 0 = ngắt liên kết bit đầu ra GPIO.

- Thanh ghi đọc đầu vào dữ liệu:

Dùng để đọc trạng thái hiện hành của các chân GPIO từ vị trí GPIO_DATASET.

- Thanh ghi xóa đầu ra dữ liệu:

Vị trí GPIO_DATACLR được dùng để xóa các bit đầu ra GPIO độc lập như sau:

- 1 = xóa bit đầu ra GPIO.
- 0 = không tác động đến bit đầu ra GPIO.

- Các chân đọc đầu ra dữ liệu:

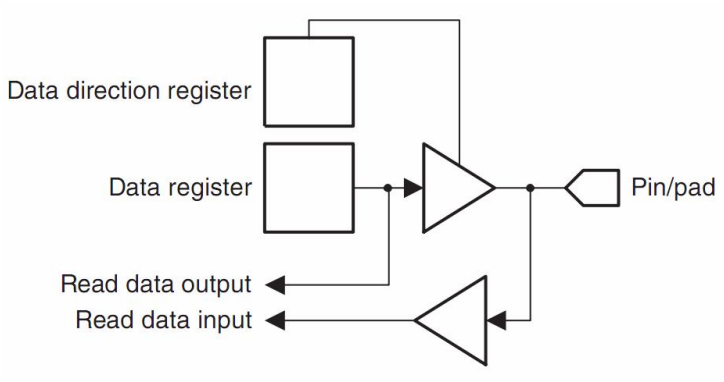
Dùng để đọc trạng thái hiện hành của các bit thanh ghi đầu ra GPIO từ vị trí GPIO_DATACLR.

- Điều khiển hướng dữ liệu:

Thanh ghi GPIO_DIRN được dùng để thiết lập hướng cho mỗi chân GPIO như sau:

- 1 = chân là đầu ra.
- 0 = chân là đầu vào (mặc định).

Điều khiển hướng dữ liệu cho một bit GPIO được mô tả trong hình 2.11.



Hình 2.11: **Điều khiển hướng dữ liệu GPIO (1 bit).**

2.2.6 Giao tiếp với khối truyền/thu không đồng bộ đa năng (UART)

Giao tiếp UART là giao tiếp nối tiếp, có các tính năng sau:

- Cấp tín hiệu cho các đầu vào bộ điều khiển;
- Điều khiển đầu ra tín hiệu;
- Thiết lập tốc độ baud;
- Truyền và nhận 16 byte FIFO;
- Đưa ra các ngắt.

Chức năng UART

UART trên vi điều khiển ARM hỗ trợ cả hai kiểu giao tiếp là giao tiếp song công và giao tiếp bán song công. Giao tiếp song công tức là có thể gửi và nhận dữ liệu vào cùng một thời điểm. Còn giao tiếp bán song công là chỉ có một thiết bị có thể truyền dữ liệu vào một thời điểm, với tín hiệu điều khiển hoặc mã sẽ quyết định bên nào có thể truyền dữ liệu. Giao tiếp bán song công được thực hiện khi mà cả hai chiều chia sẻ một đường dẫn hoặc nếu có hai đường nhưng cả hai thiết bị chỉ giao tiếp qua một đường ở cùng một thời điểm.

Dữ liệu truyền được ghi vào thành 16 byte FIFO (bộ đệm vào trước ra trước) và bắt đầu quá trình truyền các khung dữ liệu với các tham số được xác định trong thanh ghi điều khiển UART. Truyền sẽ kết thúc khi dữ liệu trong FIFO là trống.

Trong quá trình thu, UART bắt đầu lấy mẫu sau khi nhận một bit khởi động (đầu vào mức thấp). Khi một word (16 bit) được nhận đủ, nó sẽ được chứa trong FIFO nhận.

Có thể không dùng các chế độ FIFO. Trong trường hợp này, UART cung cấp các thanh ghi giữ một byte cho việc truyền và nhận các kênh. Một bit tràn trong UART_RSR làm một ngắt được thiết lập trong trường hợp một byte nhận trước một byte được đọc.

Có thể không sử dụng tính năng của FIFO nhưng nếu xảy ra lỗi tràn, dữ liệu dư vẫn được chứa trong FIFO và phải đọc ra để xóa FIFO.

Thiết lập tốc độ baud của UART được lập trình bởi các thanh ghi chia tốc độ bit UART_LCRM và UART_LCRL.

Các ngắt UART

Mỗi UART tạo ra bốn ngắt:

- Bộ ngắt trạng thái: dùng để xác nhận khi có bất kỳ sự thay đổi trạng thái nào. Bộ ngắt này được xóa bằng cách ghi vào thanh ghi UART_ICR.
- Ngắt loại bỏ UART: dùng để xác nhận khi UART được loại bỏ và bit khởi động (mức thấp) được phát hiện trên đường thu. Trạng thái này sẽ được xóa nếu UART được kích hoạt hoặc đường thu ở mức cao.
- Ngắt Rx (ngắt thu dữ liệu): dùng để xác nhận khi một trong các trường hợp sau xảy ra:
  - Bộ FIFO thu được kích hoạt và bộ FIFO đã chứa nửa hoặc quá nửa (8 byte hoặc nhiều hơn 8 byte).
  - Bộ FIFO thu không còn trống và không có dữ liệu cho hơn chu kỳ 32 bit.
  - Bộ FIFO thu bị vô hiệu và dữ liệu đã được thu.
  - Ngắt Rx được xóa bằng việc đọc nội dung của FIFO.
- Ngắt Tx (ngắt truyền dữ liệu): dùng để xác nhận khi một trong các trường hợp sau xảy ra:
  - Bộ FIFO truyền được kích hoạt và bộ FIFO chứa một nửa hoặc ít hơn một nửa.
  - Bộ FIFO truyền bị vô hiệu hóa và vùng đệm lưu trữ là trống.
  - Ngắt Tx sẽ được xóa khi chèn vào bộ FIFO hơn quá nửa hoặc ghi vào thanh ghi lưu trữ.

Dạng khung truyền

Khung truyền trong giao tiếp UART gồm bốn thành phần, như trong hình 2.12.

Hình 2.12: **Khung truyền trong giao tiếp UART.**

- Start bit (1 bit ở mức logic 0): bắt đầu một gói tin, đồng bộ xung nhịp clock;
- Data (có thể là 5,6,7 hoặc 8 bit): dữ liệu cần truyền;
- Parity bit (1 bit: chẵn (even), lẻ (odd), mark, space): bit cho phép kiểm tra lỗi;
- Stop bit (1 hoặc 2 bit): kết thúc một gói tin.

Tốc độ truyền

- Tính bằng đơn vị bit/giây: bps (bit per second) hay còn gọi là tốc độ baud.
- Là số bit truyền trong một giây.
- Tốc độ tối đa =Tần số xung nhịp clock/hằng số.

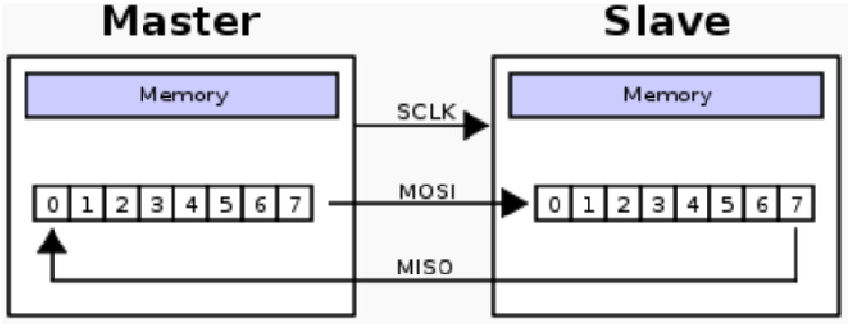
- VD: tần số thạch anh ngoài cho vi điều khiển là 18.432MHz, hằng số =16 -> tốc độ truyền là: 115.200 bps.
- Bên trong UART hỗ trợ các thanh ghi cho phép xác định các tốc độ làm việc khác, vd: 1.200, 2.400, 4.800, 9.600, 19.200, 38.400… bps, có thể thiết lập tốc độ baud bằng phần mềm.

2.2.7 Giao tiếp ngoại vi nối tiếp (SPI)

Chức năng SPI

Trong vi điều khiển ARM, khối SPI (Serial Peripheral Interface) được tích hợp sẵn bên trong và nằm trên bus truyền dữ liệu ngoại vi tối ưu để tăng hiệu suất truyền dữ liệu và tối ưu khả năng tiêu thụ công suất.

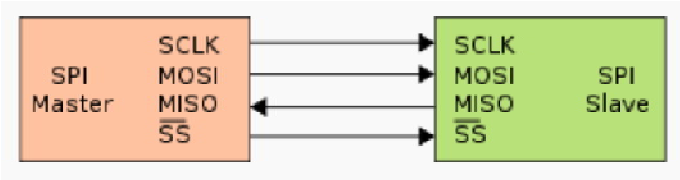
- SPI sử dụng phương thức truyền: Nối tiếp – Đồng bộ – Song công.
  - Nối tiếp: truyền một bit dữ liệu trên mỗi nhịp truyền;
  - Đồng bộ: có xung nhịp đồng bộ quá trình truyền;
  - Song công: cho phép gửi, nhận đồng thời.
- SPI là giao thức Master – Slave
  - Thiết bị đóng vai trò Master điều khiển xung đồng bộ (SCK);
  - Tất cả các thiết bị slave bị điều khiển bởi xung đồng bộ phát ra bởi Master.
  - SPI là giao thức trao đổi dữ liệu (Data Exchange): một bit được gửi ra sẽ có một bit khác được nhận về, được mô tả như trong hình 2.13.



Hình 2.13: **Giao thức Master – Slave trong giao tiếp SPI.**

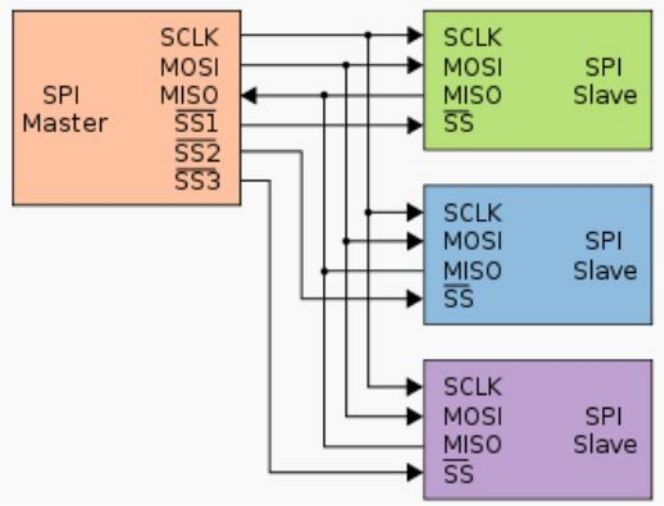
Các cấu hình ghép nối cơ bản trong giao tiếp SPI

- Cấu hình ghép nối một thiết bị được mô tả trong hình 2.14



Hình 2.14: **Ghép nối một thiết bị.**

- Cấu hình ghép nối nhiều thiết bị được mô tả trong hình 2.15 (1 Master – n Slave):



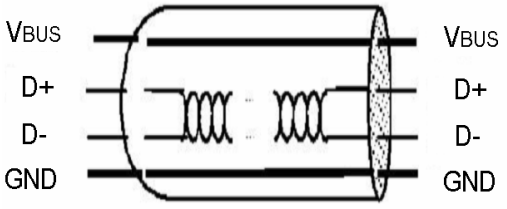
Hình 2.15: **Ghép nối nhiều thiết bị.**

- Mô tả các chân trong giao tiếp SPI:
  - MISO (Master Input Slave Output);
  - MOSI (Master Output Slave Input);
  - SCK: xung đồng bộ;
  - SS (Slave select): chân chọn thiết bị (để một thiết bị slave có thể làm việc, chân SS phải giữ ở mức thấp).
- Các thiết bị sử dụng giao tiếp SPI rất đa dạng, bao gồm: thẻ nhớ SD/MMC, bộ nhớ, cảm biến ảnh, ADC, LCD, …

2.2.8 Giao tiếp USB

Trong vi điều khiển ARM, khối giao tiếp chuẩn USB được tích hợp sẵn bên trong và nằm trên bus truyền dữ liệu ngoại vi tối ưu để tăng hiệu suất truyền dữ liệu và tối ưu khả năng tiêu thụ công suất. Trong khối giao tiếp USB có một bộ xử lý truyền và nhận dữ liệu và một bộ đệm FIFO. Dữ liệu truyền trên USB theo giao thức đã được định nghĩa sẵn.

Giao tiếp USB là một chuẩn truyền dữ liệu nối tiếp đa năng [10] với các thiết bị ngoại vi. Sơ đồ truyền tín hiệu theo chuẩn giao tiếp USB được mô tả trong hình 2.16.



Hình 2.16: **Sơ đồ truyền tín hiệu theo chuẩn giao tiếp USB.**

Quá trình trao đổi dữ liệu

Các thiết bị USB có thể trao đổi dữ liệu với máy chủ theo bốn kiểu hoàn toàn khác nhau, cụ thể:

- Truyền điều khiển (control transfer);
- Truyền ngắt (interrupt transfer);
- Truyền theo khối (bulk transfer);
- Truyền đẳng thời (isochronous transfer).

Truyền điều khiển: để điều khiển phần cứng, các yêu cầu điều khiển được truyền. Kiểu truyền này làm việc với mức ưu tiên cao và với khả năng kiểm soát lỗi tự động. Tốc độ truyền lớn vì có đến 64 byte trong một yêu cầu có thể được truyền.

Truyền ngắt: các thiết bị, cung cấp một lượng dữ liệu nhỏ, tuần hoàn. Hệ thống sẽ hỏi theo chu kỳ, chẳng hạn 10 ms một lần xem có các dữ liệu mới gửi đến.

Truyền theo khối: khi có lượng dữ liệu lớn cần truyền và cần kiểm soát lỗi truyền nhưng lại không có yêu cầu về thời gian truyền thì dữ liệu thường được truyền theo khối.

Truyền đẳng thời: khi có khối lượng dữ liệu lớn với tốc độ dữ liệu đã được quy định. Theo cách truyền này một giá trị tốc độ xác định được duy trì. Việc hiệu chỉnh lỗi không được thực hiện vì những lỗi truyền nhỏ không gây ảnh hưởng đáng kể.

Ưu điểm giao tiếp USB

- Dễ sử dụng:
  - Một giao tiếp dùng chung cho nhiều thiết bị ngoại vi khác nhau;
  - Tự động cấu hình;
  - Dễ dàng đấu nối;
  - Hỗ trợ khả năng cắm nóng (Hot pluggable);
  - Thường không cần sử dụng nguồn ngoài.
- Tốc độ cao và tin cậy, hỗ trợ nhiều tốc độ khác nhau:
  - Tốc độ cao nhất: 480 Mbps;
  - Tốc độ cao : 12 Mbps;
  - Tốc độ thấp : 1,5 Mbps.
- Tiết kiệm điện.
- Lợi ích cho người dùng phát triển (thiết kế phần cứng, lập trình nhúng, lập trình ứng dụng).
- Linh hoạt:
  - Chuẩn giao tiếp USB hỗ trợ bốn kiểu truyền và ba tốc độ khác nhau -> có thể phù hợp cho nhiều loại thiết bị ngoại vi;

- Có thể hỗ trợ truyền các gói dữ liệu có ràng buộc hoặc không ràng buộc về thời gian làm tăng tính thời gian thực
- Hỗ trợ giao thức để giao tiếp với các thiết bị chuẩn như máy in, bàn phím, ổ đĩa, đầu đọc thẻ, …

- Được hỗ trợ bởi hệ điều hành:
  - Các hệ điều hành phổ biến đều hỗ trợ chuẩn USB: Windows, Linux, Macintosh;
  - Phát hiện khi thiết bị được cắm vào hay rút ra khỏi hệ thống;
  - Giao tiếp với thiết bị được cắm vào để tìm ra cách trao đổi dữ liệu;
  - Hỗ trợ các giao diện hàm chuẩn (API – Application Programming Interface) cho phép lập trình giao tiếp với thiết bị;
  - Được hỗ trợ bởi nhiều nhà sản xuất;
  - Các chip chuyên dụng hỗ trợ giao tiếp theo chuẩn USB khá phổ biến;
  - Giới hạn về khoảng cách: giới hạn chiều dài đường truyền không quá 5m;
  - Có thể tăng khoảng cách bằng các mạch chuyển đổi (USB <-> RS485, wifi,…).

2.2.9 Kiến trúc bus truyền dữ liệu cao cấp của vi điều khiển ARM

Giới thiệu chung

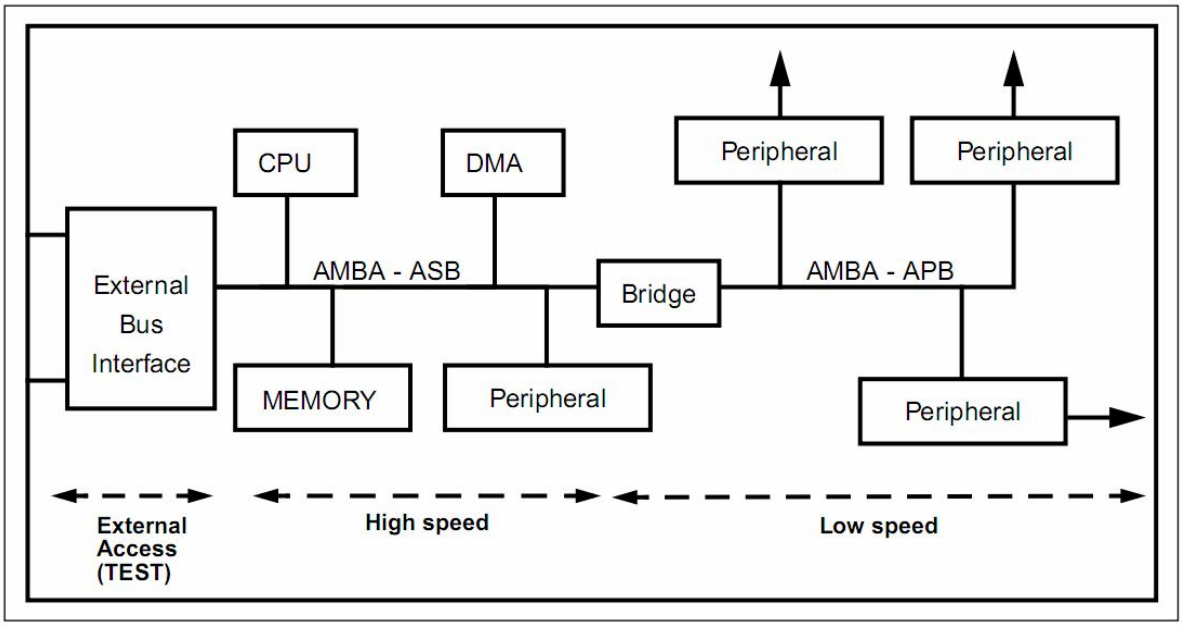
ARM giao tiếp với các khối ngoại vi bởi hệ thống bus truyền dữ liệu cao cấp AMBA (Advanced Microcontroller Bus Architecture) [11], [12].

Đặc điểm của AMBA là chuẩn truyền thông trên chip dành cho thiết kế các vi điều khiển 16 và 32 bit với hiệu suất cao, các bộ xử lý tín hiệu và các thiết bị ngoại vi phức tạp.

AMBA là một đặc tính dành cho các bus trên chip, cho phép các khối lệnh (như điều khiển bộ nhớ, CPU, DSP và các thiết bị ngoại vi) được kết nối với nhau để thành một vi điều khiển hay thành chip ngoại vi phức tạp.

- AMBA được thiết kế vào:
  - Các bộ vi điều khiển PDA, với số lượng lớn các thiết bị ngoại vi được tích hợp và khả năng tiêu thụ điện rất thấp.
  - Các bộ vi điều khiển đa phương tiện như các bộ điều khiển hình ảnh có bộ nhớ dung lượng lớn.
  - Các thiết bị ASIC phức tạp cho các sản phẩm chuyên dụng.
  - Tích hợp điều khiển và các chức năng xử lý tín hiệu các thiết bị truyền thông di động số.
- Chính sách của ARM là có thể hỗ trợ sử dụng AMBA để phát triển các bo mạch và các công cụ khác.

- Mục tiêu của AMBA:
  - Dễ dàng phát triển các mạch nhúng vi điều khiển với một hay nhiều bộ xử lý và nhiều thiết bị ngoại vi.
  - Giảm thiểu được tổng chi phí trong sản xuất thử nghiệm chip.
  - AMBA hỗ trợ thiết kế các khối, cấu trúc và các bộ xử lý độc lập, hỗ trợ phát triển thư viện các thiết bị ngoại vi và sẵn sàng cung cấp bộ nhớ truy cập nhanh, các lõi CPU cao cấp.
- Để đạt được như vậy, kiến trúc AMBA có những tính năng chính sau:
  - Tính đơn thể cao.
  - Hỗ trợ đa dịch vụ.
  - Nguồn tiêu thụ thấp.
  - Phương pháp kiểm thử cao cấp.
- AMBA là một hệ mở, bất kỳ ai cũng có thể sao chép các đặc tính kỹ thuật từ ARM và dùng giao thức của AMBA để thiết kế chip. Không có sự liên đới đến bản quyền hay tiền bản quyền. Các hội viên ARM có thể cung cấp và hỗ trợ về hệ thống AMBA.
- Đặc tính kỹ thuật AMBA:
  - Bus băng thông rộng, tốc độ cao.
  - Bus ngoại vi có công suất thấp, cấu trúc đơn giản.
  - Cho phép truy cập kiểm thử các khối nhanh.
  - Các hoạt động quản lý hiệu quả (Reset hoặc bật nguồn, quá trình khởi tạo và chế độ nghỉ).
- Kiến trúc AMBA điển hình được mô tả trong hình 2.17.



Hình 2.17: **Vi điều khiển dựa trên kiến trúc AMBA điển hình.**

- Bus hệ thống hiệu suất cao (ASB - Advanced System Bus):
    - Bus này là xương sống của hệ thống chính, đảm bảo tốc độ dữ liệu giữa hệ thống với các giao tiếp bus bên ngoài. Bộ xử lý trung tâm (CPU), các bus truyền chính khác (như bộ điều khiển truy cập bộ nhớ trực tiếp – DMA), bộ nhớ trong cũng được kết nối với bus truyền tốc độ cao này.
    - ASB được kết nối với APB bởi cầu nối.
- Bus truyền dữ liệu ngoại vi tối ưu (APB - Advanced Peripheral Bus):

Bus truyền ngoại vi này có công suất thấp, tốc độ thấp và đơn giản. Đây là các thiết kế thường thấy trong ARM, các bus ở đây hẹp hơn và đơn giản hơn dành cho kết nối các thiết bị ngoại vi chung như là các bộ định thời, các cổng vào ra song song, bộ thu/phát không đồng bộ đa năng,… bằng cách đặt các thiết bị ngoại vi truy cập trên APB và phân vùng chúng đi từ ASB, làm giảm bớt lượng tải trên ASB và đạt được hiệu suất tối đa trên ASB.

- Truy cập ngoài:

Truy cập ngoài được sử dụng cho việc truy cập kiểm thử. Trường hợp thông thường là giao tiếp với bộ nhớ ngoài nhưng có thể cài đặt bất kỳ chân ra nào cho việc kiểm thử. Kiểm thử viên có thể kiểm soát các bus và kiểm tra các thành phần một cách độc lập hay lần lượt. Phương pháp truy xuất song song cho phép kiểm thử nhanh đặc biệt là đối với các vùng nhớ đệm CPU.

- Vi điều khiển dựa trên AMBA chứa bộ quản lý thông tin nguồn và trạng thái Reset để đảm bảo:
    - Trình điều khiển duy nhất khi cấp nguồn.
    - Các chế độ tiết kiệm điện hay chế độ ngủ.
    - Cơ chế khởi động lại cho các trạng thái bình thường, nóng hoặc chờ.
- Bus hệ thống hiệu suất cao (ASB) được thiết kế sử dụng với hiệu năng cao và băng thông lớn với những đặc tính:
    - Bus địa chỉ và bus truyền dữ liệu riêng biệt;
    - Hỗ trợ kiến trúc đường ống;
    - Hỗ trợ các bus dữ liệu chính;
    - Hỗ trợ các thiết bị ghép nối phụ, bao gồm cả cầu nối đến bus truyền dữ liệu ngoại vi tối ưu (APB);
    - Bộ phân xử và bộ giải mã trung tâm.
    - Tốc độ truyền tùy thuộc vào đặc điểm thiết kế và mục đích sử dụng. Cấu hình tốc độ truyền này không bị giới hạn bởi chỉ tiêu kỹ thuật.
- Bus truyền dữ liệu ngoại vi tối ưu (APB) được thiết kế để làm bus con cho bus truyền chính ASB và được kết nối bằng cầu nối (cầu nối này giới hạn tải ASB).

- Mục tiêu APB là bus truyền dữ liệu đơn giản, sử dụng điện áp thấp với những đặc tính:
    - Truy cập dữ liệu được điều khiển chỉ bởi sự lựa chọn và cho qua, không cần xung nhịp.
    - Công suất tiêu hao gần như bằng không khi bus truyền này không sử dụng.
    - Giao tiếp đường truyền đơn giản.
    - Tốc độ truyền dữ liệu phụ thuộc vào tốc độ của các thiết bị ngoại vi.
    - Cấu hình tốc độ truyền không bị giới hạn bởi chỉ tiêu kỹ thuật, có thể thay đổi theo thiết kế người dùng.

Các bus dữ liệu của APB có thể được tối ưu hóa để tương thích với các thiết bị ngoại vi kết nối. Rất nhiều các thiết bị ngoại vi có yêu cầu đường truyền dữ liệu hẹp, và một cơ chế kết nối thiết bị ngoại vi 32 bit trước cầu nối, tiếp theo sau cầu nối là cơ chế kết nối với các thiết bị ngoại vi 8 bit, để giảm vùng không cần sử dụng trong bus truyền dữ liệu, tối ưu hóa bus truyền dữ liệu.

Mặc dù xung nhịp không được thiết lập trong AMBA, phân vùng cung cấp bởi cầu nối và APB đã tối giản được việc tiêu hao công suất. Rất nhiều các thiết bị ngoại vi như các bộ định thời, bộ tạo tốc độ Baud (BRG), bộ điều chế độ rộng xung (PWM) yêu cầu chia xung nhịp hệ thống, và các vị trí đó có thể lập trình, được phân chia bên cạnh cầu nối rất tiện lợi và tạo ra chế độ nguồn hiệu quả.

Không có bus truyền dữ liệu chính trong APB (ngoại trừ cầu nối). Tất cả các thiết bị ngoại vi hoạt động như là các hệ thụ động.

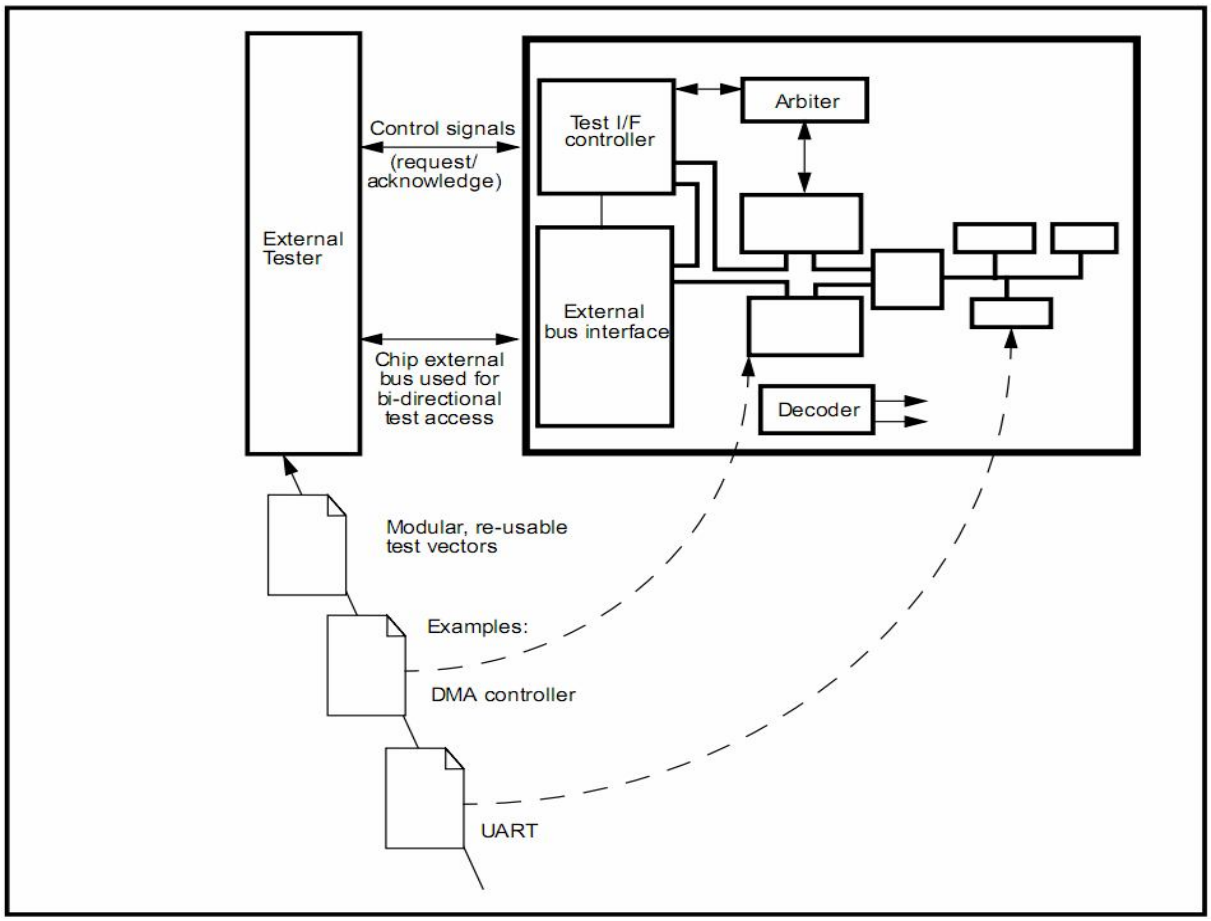
Đặc tính giữa ASB và APB

- ASB được dùng cho các bộ điều khiển CPU, DSP, DMA và các bus truyền dữ liệu chính khác, hay các thiết bị ngoại vi có hiệu suất cao.
- APB được dùng cho đường truyền dẫn phụ, định địa chỉ thanh ghi các thiết bị ngoại vi, đặc biệt là khi số lượng các thiết bị ngoại vi lớn mà nguồn tiêu thụ yêu cầu thấp.
- ASB và APB sử dụng cùng hệ phương pháp kiểm thử trong AMBA.

Bộ điều khiển giao tiếp kiểm thử

Bộ điều khiển giao tiếp kiểm thử là một bus truyền dữ liệu ASB chính, được dùng để giao tiếp với bus truyền dữ liệu bên ngoài (hoặc với các chân tương thích khác) để truy cập kiểm thử của thiết bị bên ngoài, được mô tả trong hình 2.18.

Cơ chế này cho phép truy cập cổng kiểm thử đếm một mức thấp ở cổng logic, có thể kiểm thử nhanh bằng cách truy cập song song. Phương pháp kiểm thử cho phép dùng lại các kết quả kiểm thử trung gian, lưu lại giá trị tại thời điểm thử. Chẳng hạn như khi một khối thiết bị ngoại vi được dùng trở lại, khối trung gian kiểm thử (khối đã được kiểm thử vẫn tồn tại) có thể được sử dụng lại, và tiếp tục kiểm thử ở mức cao hơn của chương trình.



Hình 2.18: **Bộ điều khiển giao tiếp kiểm thử sử dụng theo dạng khối.**

Bộ điều khiển giao tiếp kiểm thử có khả năng kiểm tra hoạt động của các thiết bị ngoại vi, bộ phân xử có nhiệm vụ phân chia tốc độ bus cho phù hợp tốc độ truyền của từng thiết bị ngoại vi.

Ưu điểm

AMBA đã được sử dụng trên các sản phẩm ARM và đã chứng tỏ được các tính năng hiệu dụng. Hiện nay kiến trúc bus truyền dữ liệu AMBA được sử dụng nhiều trên các bộ vi điều khiển tích hợp cao cấp khác bởi đáp ứng được các vấn đề:

- Kiến trúc chip có phân vùng tốc độ, tiết kiệm điện năng.
- Quản lý dự án tiện ích bởi việc sử dụng khối kiểm thử, do đó giảm bớt thời gian nghiên cứu phát triển sản phẩm.
- Sản phẩm có nhiều tính năng cao cấp, bộ vi xử lý độc lập, hiện đại và có tính cạnh tranh.
- ARM luôn sẵn sàng hỗ trợ các công cụ và môi trường phát triển hệ thống bus truyền dữ liệu AMBA.

2.3 Kết luận

Chương 2 trình bầy tổng quan giao tiếp cơ bản trong vi điều khiển ARM, trên cơ sở đó, tùy theo ứng dụng vi điều khiển ARM được bổ xung thêm các tính năng cao cấp

hơn (được trình bầy trong chương 3). Các giao tiếp trên vi điều khiển ARM luôn có khả năng tương tác tốt với các thiết bị ngoại vi trên hệ thống bus truyền dữ liệu có tốc độ xử lý cao, nhưng tiêu thụ năng lượng thấp.

Các giao tiếp với vi điều khiển ARM đều được hãng hỗ trợ, định nghĩa thành các module và được tối ưu từ khi thiết kế theo kiểu cấu trúc trong lập trình C, rất thuận lợi cho người lập trình tiếp cận và phát triển ứng dụng. Chính vì vậy, vi điều khiển ARM có phạm vi ứng dụng rộng rãi, luôn có sẵn các công cụ hỗ trợ giao tiếp cho cả phần cứng và phần mềm.

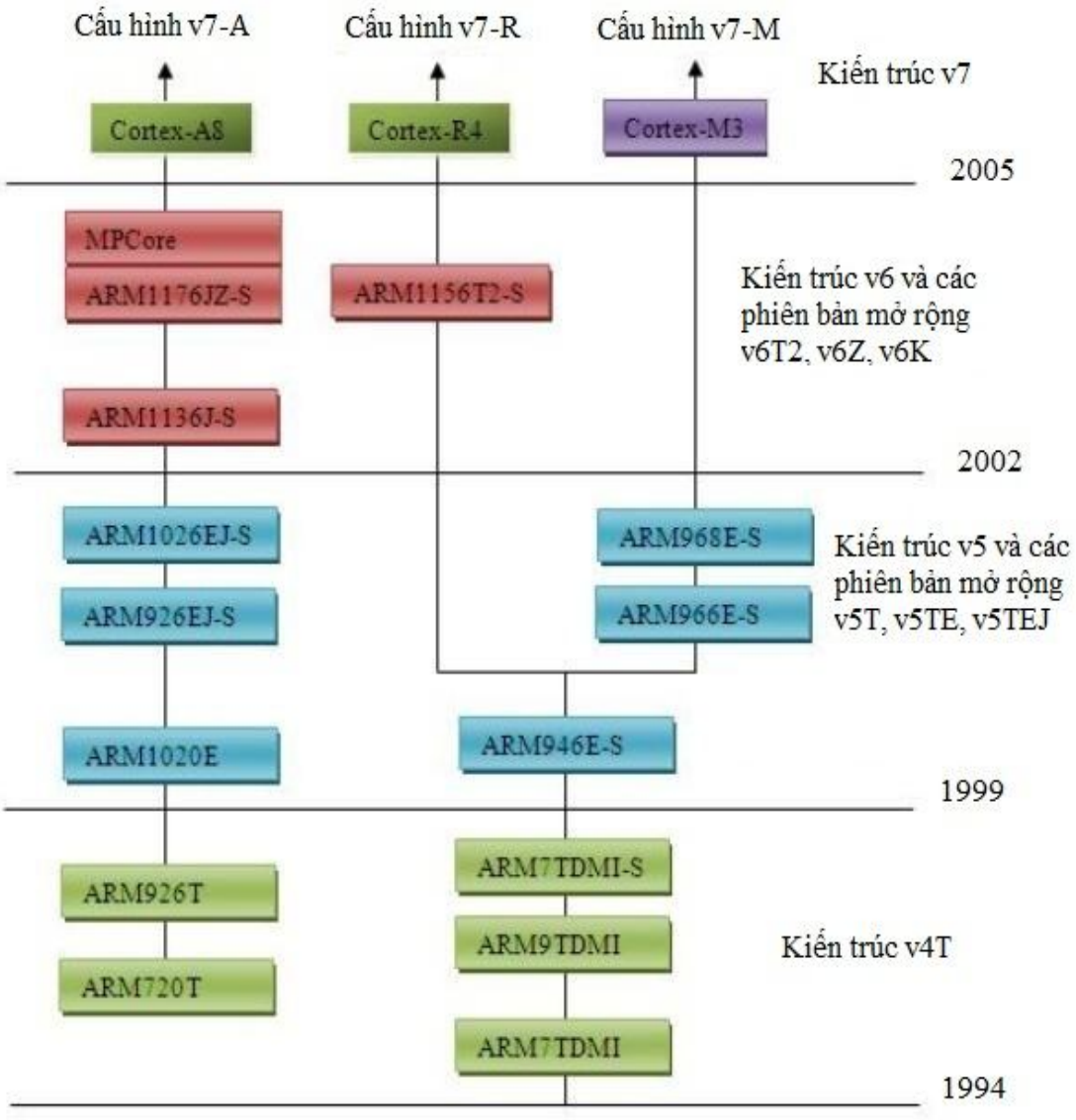
CHƯƠNG 3
ĐẶC ĐIỂM CÁC DÒNG LÕI XỬ LÝ ARM

3.1 Phân loại và tính năng các dòng lõi xử lý ARM

Phân loại các dòng lõi xử lý ARM

Các dòng lõi xử lý ARM cho đến nay có bốn kiến trúc [13], bao gồm: kiến trúc v4T, kiến trúc v5, kiến trúc v6 và kiến trúc v7.

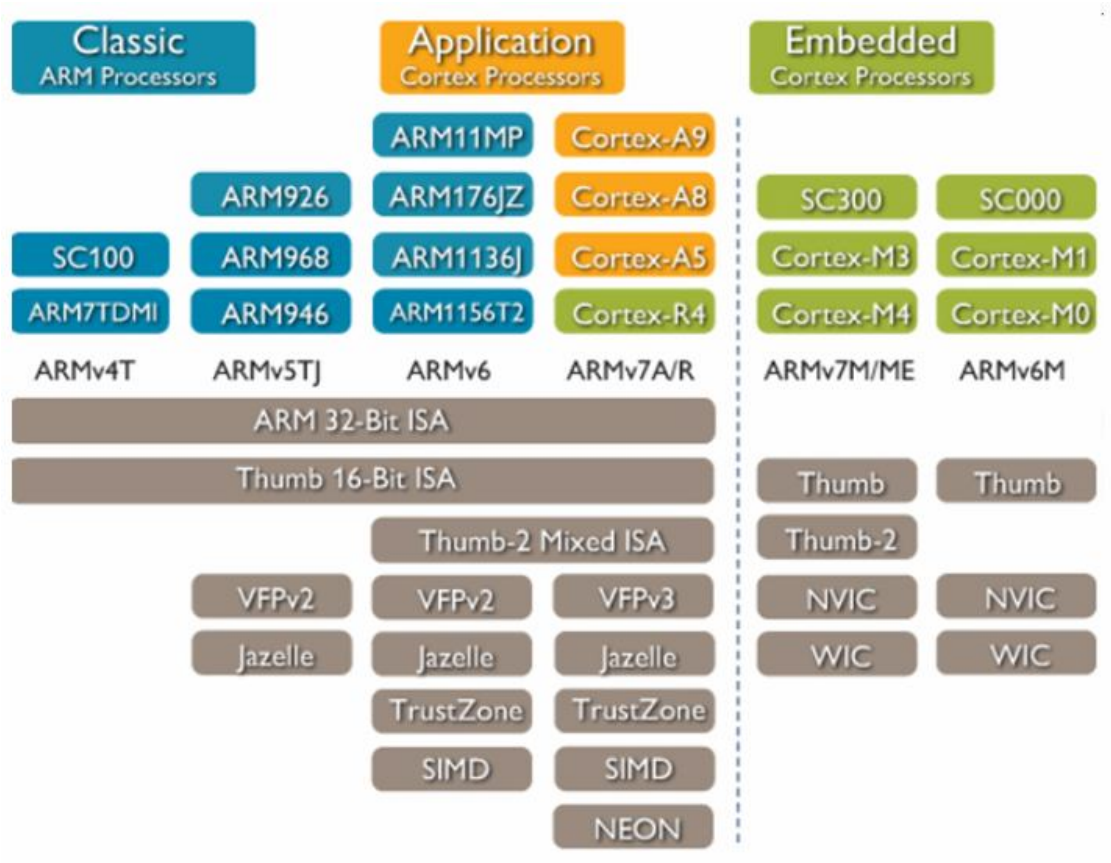
Kiến trúc ARMv4T là kiến trúc cơ bản, các kiến trúc ARM sau bao gồm v5, v6, v7 đều kế thừa từ kiến trúc ARMv4T. Hình 3.1 mô tả sự hình thành và phát triển các kiến trúc lõi xử lý ARM từ kiến trúc ARMv4 đến ARMv7.



Hình 3.1: **Các kiến trúc lõi xử lý ARM.**

Tính năng các dòng lõi xử lý ARM

Tính năng các dòng lõi xử lý ARM được mô tả trong hình 3.2.



Hình 3.2: **Tính năng các dòng lõi xử lý ARM.**

Trong đó:

ARM 32-Bit ISA (Instruction Set Architecture): cấu trúc tập lệnh ARM 32 bit.

Thumb 16-Bit ISA: cấu trúc tập lệnh Thumb 16 bit được thiết lập bằng cách phân tích tập lệnh ARM 32 bit và chuyển hóa tốt nhất phù hợp với tập lệnh 16 bit, làm giảm kích thước mã.

Thumb: đặc tính Thumb để cải thiện mật độ biên dịch mã, bộ xử lý thực hiện tập lệnh 16 bit. Ở chế độ này có một số toán hạng đi kèm sẽ ẩn đi và giới hạn một số khả năng so với chế độ tập lệnh ARM đầy đủ. Trong Thumb, các mã sẽ nhỏ hơn và ít chức năng cho cải thiện mật độ mã tổng thể. Trong trường hợp bộ nhớ hoặc bus truyền dữ liệu bị hạn chế dưới 32 bit, mã Thumb cho phép tăng hiệu suất thành mã ARM 32 bit để tăng khả năng xử lý trên băng thông lớn hơn.

Thumb-2: được đưa ra để bổ sung cho các giới hạn tập lệnh 16 bit Thumb với việc cung cấp thêm tập lệnh 32 bit mở rộng. Mục tiêu của Thumb-2 là đạt được mật độ mã như Thumb với hiệu suất tương đương như tập lệnh ARM 32 bit.

Thumb-2 Mixed ISA: kết hợp tập lệnh 16 bit và 32 bit mà có thể không cần chế độ chuyển mạch. Kết hợp tập lệnh 16 bit và 32 bit ngay tại thời điểm đang thực hiện lệnh và chỉ trong một lệnh đơn (không gây giảm hiệu suất thực thi).

VFPv2 (Vector Floating Point): là bộ thực hiện phép tính dấu chấm động của kiến trúc ARM. VFPv2 có 16 thanh ghi, hoạt động với một chu kỳ đơn, khả năng tính toán và xử lý rất nhanh, độ trễ thấp, có độ chính xác cao.

VFPv3 là phiên bản nâng cấp của VFPv2, VFPv3 có độ chính xác cao hơn với 32 thanh ghi và một số tập lệnh được mở rộng.

NVIC (Nested Vectored Interrupt Controller): Bộ điều khiển vector ngắt lồng nhau có khả năng xử lý ngắt rất linh hoạt và nhanh chóng và cho phép rút ngắn thời gian trì hoãn đáp ứng ngắt (hệ thống đáp ứng ngắt nhanh hơn) với nhiều mức ưu tiên khác nhau.

Jazella: là công nghệ hỗ trợ trình thông dịch mã Java, cho phép lõi ARM thực thi trực tiếp mã Java trong cấu trúc phần cứng như là trạng thái thực thi thứ ba cùng với các chế độ ARM và Thumb hiện hành, làm tăng tốc khả năng thực thi.

WIC (Wake-up Interrupt Controller): Bộ điều khiển đánh thức khi có ngắt, giúp cho các hoạt động tiêu tốn ít năng lượng hơn.

TrustZone: Khối tăng tính bảo mật, đảm bảo các đoạn mã độc hại không làm ảnh hưởng đến hệ thống.

SIMD (Single Instruction Multiple Data): Khối tập lệnh đơn đa dữ liệu, khối này cho phép tập lệnh hoạt động tại cùng thời điểm trên các mục dữ liệu khác nhau, làm tăng khả năng xử lý dữ liệu và đặc biệt hiệu quả đối với các dữ liệu dạng âm thanh và hình ảnh.

NEON: Công nghệ NEON mục đích là tăng hiệu suất xử lý cho các định dạng đa phương tiện, công nghệ này được phát triển mở rộng từ công nghệ SIMD, có khả năng làm tăng các thuật toán xử lý tín hiệu như là mã hóa và giải mã các định dạng âm thanh, hình ảnh; đồ họa hai chiều, ba chiều; trò chơi; tổng hợp xử lý giọng nói, hình ảnh trong thoại với hiệu suất xử lý cao.

3.2 Đặc điểm các dòng lõi xử lý ARM

3.2.1 Đặc điểm của kiến trúc dòng lõi xử lý ARM v4T

Kiến trúc v4T được ARM giới thiệu năm 1994, bao gồm các lõi xử lý bao gồm: ARM7TDMI, ARM720T, ARM920T.

Kiến trúc v4T hỗ trợ tập lệnh Thumb (viết tắt là T trong các ký hiệu của bộ xử lý). Hỗ trợ cùng lúc tập lệnh Thumb 16 bit và ARM 32 bit. Với tập lệnh Thumb 16 bit cho phép trình biên dịch tạo ra chương trình nhỏ hơn mà vẫn tương thích với hệ thống

32 bit. Điển hình ở kiến trúc này là lõi ARM7TDMI được thiết kế nhằm đáp ứng các ứng dụng yêu cầu hiệu suất cao, tiêu thụ năng lượng thấp và nhỏ gọn.

Ý nghĩa các ký hiệu trên "ARM7TDMI":

- T là hỗ trợ tập lệnh Thumb 16 bit;

- D là Debug - hiệu chỉnh lỗi;

- M có nghĩa là "Long Multiply Support" - hỗ trợ phép toán 64 bit;

- I là Interface, hỗ trợ giao tiếp ngoại vi.

ARM7TDMI hỗ trợ giải mã lỗi bằng khối Embedded Trace Macrocell (ETM) đây là giải pháp giải mã lỗi hoàn chỉnh dành cho lõi ARM, ngoài ra ARM7TDMI có khả năng kết hợp với các lõi khác nhằm tăng cường khả năng xử lý.

ARM7TDMI có kiến trúc đường ống ba tầng, là kiến trúc Von Neumann, bộ xử lý số học 32 bit. Hệ thống tập lệnh 16 và 32 bit có khả năng mở rộng thông qua giao diện đồng xử lý với lõi ngoài.

Ở phiên bản mở rộng ARM720T, bộ nhớ đệm và hệ thống quản lý bộ nhớ (MMU – Memory Management Unit) được tích hợp. Tiếp đó phiên bản ARM9TDMI sử dụng kiến trúc đường ống năm tầng và kiến trúc Harvard.

3.2.2 Đặc điểm kiến trúc dòng lõi xử lý ARM v5

Kiến trúc v5 và các phiên bản mở rộng v5T, v5TE, v5TEJ được ARM giới thiệu năm 1999, bao gồm các lõi xử lý:

ARM1020E/1022E v5T;

ARM946E-S/ARM966E-S/ARM968E-S v5TE;

ARM7EJ-S/ARM92EJ-S/ ARM1026EJ-S v5TEJ.

Đặc tính kỹ thuật chung của dòng ARMv5 được mô tả trong bảng 3.1:

Bảng 3.1: **Đặc điểm kỹ thuật chung của dòng ARMv5.**

Bộ xử lý	Lõi xử lý dạng RISC 32 bit
I/O	Ánh xạ bộ nhớ (Memory map I/O)
Dung lượng không gian bộ nhớ	4GBytes
Tập lệnh	16 bit, 32 bit
Chế độ hoạt động	Có bảy chế độ: User, Supervisor, Abort, Undefined, System, IRQ, FIQ
Cấu trúc tập lệnh	Hỗ trợ tập lệnh ARM 32 bit và Thumb 16 bit mở rộng
Ngắt	IRQ (Interrupt Request) và FIQ (Fast Interrupt)

Phiên bản v5T: bộ lệnh Thumb được cải tiến, hỗ trợ CLZ (Count Leading Zero), CLZ là tập lệnh hỗ trợ cho phép xác định một biến đếm giảm về 0 chỉ trong một chu kỳ lệnh, giúp giảm thời gian trong các lệnh cộng, trừ, nhân, chia các số nhị

phân; giúp tăng tốc độ trong xử lý tín hiệu số so với công nghệ thực hiện phép tính dấu chấm động.

Phiên bản v5TE: hỗ trợ khối xử lý tín hiệu số DSP (Digital Signal Processing). Với khối DSP này, năng lực xử lý tính toán số được tăng lên 70%.

Phiên bản v5TE-J: khối Jazelle được thêm vào nhằm hỗ trợ trình thông dịch mã Java và bộ thực thi mã Java. Thời gian thực thi mã Java được tăng lên tám lần và giảm được hơn 80% năng lượng tiêu thụ so với lõi xử lý không hỗ trợ khối Jazelle. Tính năng này cho phép lập trình viên thực thi mã Java một cách độc lập với hệ điều hành.

Kiến trúc v5 được sử dụng nhiều ở dòng ARM10, đặc biệt là phiên bản v5TE-J. Mặc dù không có nhiều thay đổi về kiến trúc, tuy nhiên phiên bản kiến trúc v5 được sử dụng rất nhiều bởi vi xử lý tích hợp hệ thống nên tạo được sự linh hoạt với nhiều tính năng cao cấp.

3.2.3 Đặc điểm kiến trúc dòng lõi xử lý ARM v6

Kiến trúc v6 và các phiên bản mở rộng v6T2, v6Z và v6K được ARM giới thiệu năm 2002, bao gồm các lõi xử lý: ARM1136J(F)-S, ARM1156T2(F)-S(v6T2), ARM1176JZ(F)-S(v6Z), MPCore(v6K).

Có nhiều bổ sung ở kiến trúc v6 theo hướng tạo ra những hệ thống nhúng cao cấp và phức tạp hơn nhưng vẫn giữ được ưu điểm về khả năng tiêu thụ điện năng thấp. Với mỗi phiên bản sẽ có những tính năng đặc biệt được thêm vào. Kế thừa các đặc điểm nổi trội của kiến trúc v4 và v5, ở kiến trúc v6 các khối 'TEJ' được tích hợp vào lõi ARM. Để đảm bảo khả năng tương thích ngược phần bộ nhớ và xử lý ngoại lệ được kế thừa từ kiến trúc v5.

Về kiến trúc v6, có năm điểm chính được cải tiến:

- Quản lý bộ nhớ: bộ nhớ cache và khối quản lý bộ nhớ (MMU- Memory Management Unit) được cải tiến làm tăng hiệu suất thực thi của hệ thống lên 30% so với kiến trúc cũ.
- Đa lõi xử lý (Multiprocessor): đáp ứng các hệ thống mà ở đó yêu cầu khả năng tốc độ xử lý nhanh như: phương tiện giải trí cá nhân, xử lý số… Các lõi xử lý chia sẻ và đồng bộ dữ liệu với nhau thông qua vùng nhớ chung.
- Hỗ trợ xử lý đa phương tiện: tích hợp bộ tập lệnh SIMD (Single Instruction Multiple Data) làm tăng khả năng xử lý dữ liệu dạng âm thanh và hình ảnh. SIMD cũng cho phép các nhà phát triển cài đặt các ứng dụng phức tạp hơn như: giải mã dữ liệu âm thanh và hình ảnh, các bài toán nhận dạng, hiển thị hình ảnh 3D hoặc hỗ trợ thiết bị sử dụng công nghệ không dây.
- Kiểu dữ liệu: là cách hệ thống sử dụng và lưu trữ dữ liệu trong bộ nhớ. Các hệ thống SoC (System on Chip), các chip vi xử lý đơn, hệ điều hành và các giao

diện ngoại vi như USB hoặc PCI thường hoạt động dựa trên kiểu dữ liệu "little endian". Một số các giao thức như TCP/IP hay MPEG hoạt động dựa trên kiểu dữ liệu "big endian". Để có thể tối ưu hóa khả năng tích hợp của hệ thống, ARMv6 hỗ trợ cùng lúc cả hai định dạng "little" và "big" endian, gọi tắt là "mixed-endian". Bên cạnh đó, ARMv6 còn cung cấp tập lệnh để xử lý dữ liệu dạng "unalignment" - có kích thước dữ liệu thay đổi. Tương tự như ARMv5, ARMv6 cũng là kiến trúc 32 bit, nên hỗ trợ đường truyền dữ liệu 64 bit hoặc cao hơn.

- Xử lý ngoại lệ và ngắt: để thích ứng cho các hệ thống xử lý thời gian thực.

Nhằm tăng cường tính an toàn khi thực thi mã chương trình, khối TrustZone được tích hợp ở phiên bản v6Z. Vấn đề thực thi mã an toàn xuất phát từ thực tế ngày càng nhiều thiết bị di động dựa trên nền tảng của ARM, nhiều chương trình được tải từ trên mạng do đó tính an toàn của các đoạn mã nhiều khi chưa được kiểm chứng. TrustZone đảm bảo các đoạn mã độc hại không làm ảnh hưởng đến hệ thống.

Dòng ARM11 là đại diện phổ biến nhất của kiến trúc ARMv6. Với kiến trúc đường ống tám tầng (ở ARM1156T áp dụng kiến trúc đường ống chín tầng), hệ thống dự đoán rẽ nhánh (Branch Prediction) và kết quả trả về (Return Stack) giúp ARM11 nâng cao hiệu suất thực thi lệnh.

Tập lệnh Thumb-2 cũng được giới thiệu hỗ trợ các lệnh Thumb 16 bit và 32 bit. Ở phiên bản ARM1176JZ(F)-S bổ sung khối IEM (Intelligent Energy Management) để quản lý mức tiêu thụ năng lượng tốt hơn.

3.2.4 Kiến trúc dòng lõi xử lý ARM v7

Kiến trúc v7 và các phiên bản mở rộng v7-A, v7-R và v7-M được ARM giới thiệu vào năm 2005, đặc trưng bao gồm các lõi xử lý: Cortex-A8 (v7-A), Cortex-R4 (v7-R), Cortex-M3 (v7-M).

Kiến trúc v7 được chia thành ba dòng chính dựa trên đặc thù của ứng dụng thực tiễn:

- Dòng A (viết tắt của Application), lõi ARM dòng này hỗ trợ cho các ứng dụng đòi hỏi tính phức tạp, mức độ tương tác người dùng cao như: thiết bị cầm tay di động, máy tính, công nghệ không dây…
- Dòng R (viết tắt của Realtime), lõi ARM dòng này hỗ trợ cho các ứng dụng cần tính toán xử lý thời gian thực.
- Dòng M (viết tắt của Microcontroller), lõi ARM dòng này dành cho các ứng dụng công nghiệp và điện tử tiêu dùng.

ARM Cortex là một phiên bản khác với các phiên bản ARM thường hay được ký hiệu bởi ARMXX. ARM Cortex không có tốc độ hoạt động hay hệ thống ngoại vi nhất

định mà tùy thuộc vào nhà sản xuất phần cứng sẽ thiết kế hệ thống ngoại vi khác nhau. Tuy nhiên tất cả đều dùng chung lõi ARM Cortex và việc lập trình và truy cập phần cứng tuân theo chuẩn CMSIS (The Cortex Microcontroller Software Interface Standard: Chuẩn giao tiếp phần mềm vi điều khiển Cortex).

3.3 Kết luận

Chương 3 trình bầy các tính năng và kiến trúc các loại lõi xử lý trong các dòng vi điều khiển ARM. Từ kiến trúc lõi xử lý ARMv4 đến kiến trúc lõi xử lý ARMv7, mỗi một kiến trúc đều có những tính năng xử lý đặc trưng, phiên bản sau có sự bổ sung thêm các tính năng đặc biệt, nhưng có tính kế thừa của các phiên bản trước. Ở phiên bản ARMv7, hãng ARM đã phát triển chuẩn giao tiếp phần mềm vi điều khiển Cortex (CMSIS), kết hợp chặt chẽ với các nhà cung cấp phần mềm để chuẩn hóa các giao tiếp với các thiết bị ngoại vi, các hệ điều hành thời gian thực và các thiết bị trung gian. Đây là một trong những thuận lợi khi tìm hiểu để phát triển và ứng dụng các dòng vi điều khiển có lõi xử lý ARM cao cấp.

PHẦN II - THỰC NGHIỆM

CHƯƠNG 4

ỨNG DỤNG MỘT SỐ GIAO TIẾP VỚI VI ĐIỀU KHIỂN AT91SAM7S64

4.1 Giới thiệu

Trên thị trường Việt Nam hiện nay có hai hãng cung cấp vi điều khiển ARM khá phổ biến đó là Philips và Atmel. Trong luận văn này, ta chọn vi điều khiển AT91SAM7S64 của Atmel [14] để xây dựng ứng dụng thực nghiệm.

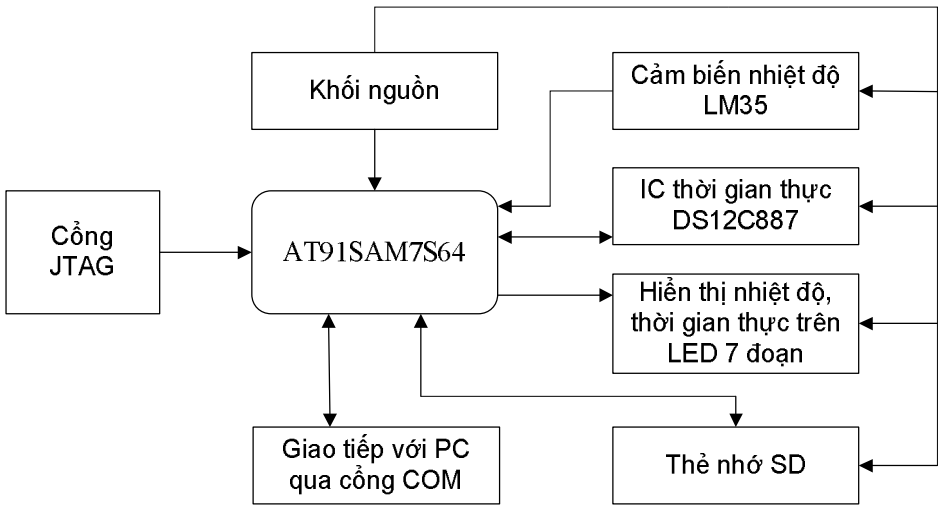
Mạch thực nghiệm có chức năng:

- Thu thập nhiệt độ, hiển thị nhiệt độ và thời gian thực trên LED 7 đoạn;
- Lưu dữ liệu vào thẻ nhớ với thời gian thực (nhiệt độ; thời gian; ngày/tháng/ năm);
- Đọc kết quả dữ liệu đã lưu vào thẻ nhớ trên máy tính bằng đầu đọc thẻ hoặc đọc trực tiếp trên mạch qua cổng COM;

Chi tiết mạch thực nghiệm bao gồm:

- Sử dụng vi điều khiển AT91SAM7S64 của Atmel;
- Cổng nạp chuẩn JTAG cho các dòng vi điều khiển AT91SAM của Atmel;
- Sử dụng cảm biến nhiệt độ tương tự LM35;
- Giao tiếp với IC thời gian thực DS12C887;
- Hiển thị nhiệt độ và thời gian thực trên LED 7 đoạn;
- Giao tiếp với SD Card;
- Giao tiếp máy tính qua cổng COM;
- Nguồn cung cấp cho mạch: nguồn ngoài 5 – 12VDC hoặc nguồn qua cổng USB trên máy tính.

Sơ đồ khối tổng quát mạch thực nghiệm được mô tả trong hình 4.1.

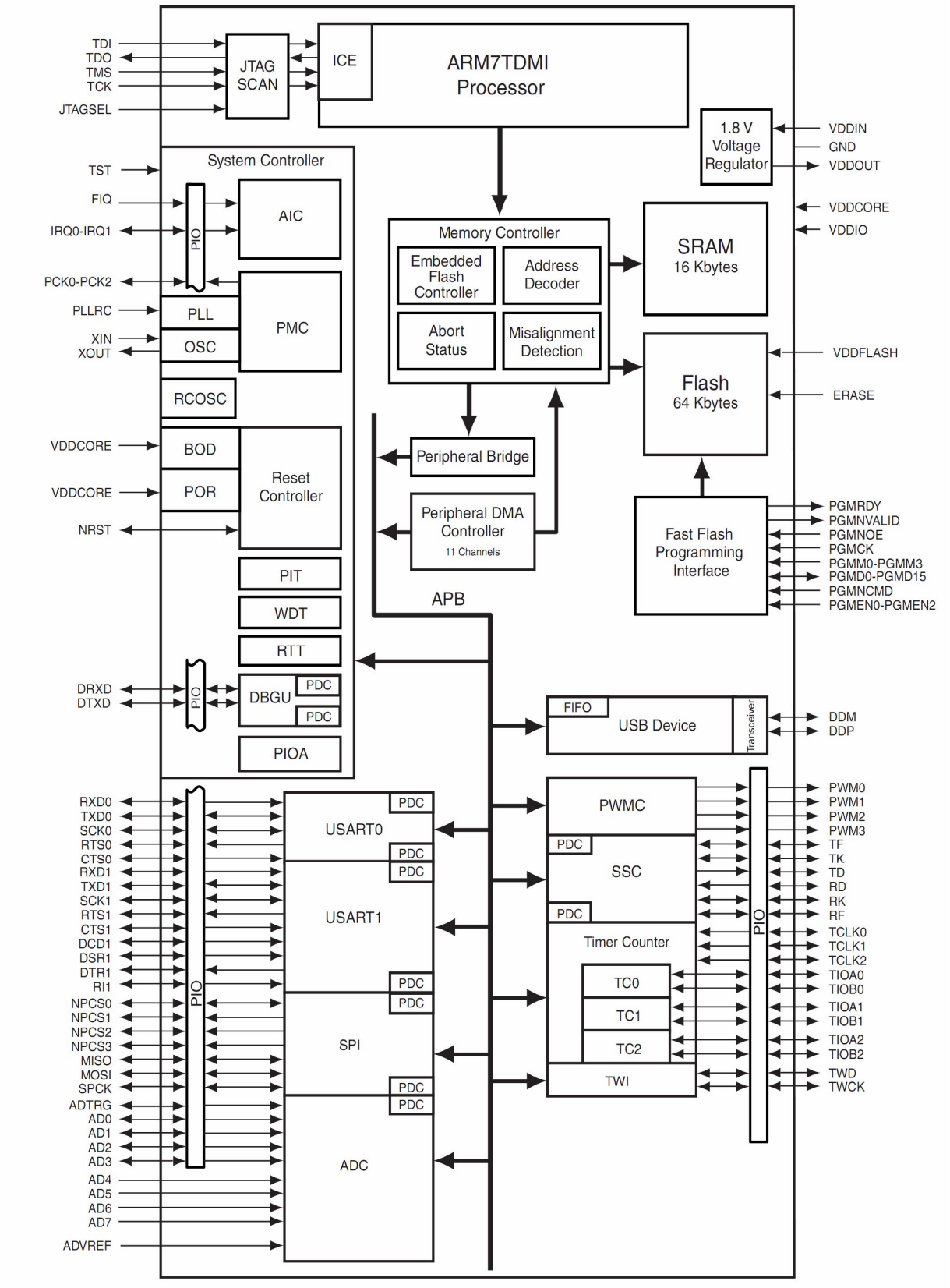


Hình 4.1: **Sơ đồ khối tổng quát mạch thực nghiệm.**

4.2 Đặc tính cơ bản của vi điều khiển AT91SAM7S64

Để xây dựng mạch thực nghiệm dựa trên vi điều khiển AT91SAM7S64, ta phải nắm được các đặc tính cơ bản và khả năng phát triển ứng dụng của vi điều khiển này [15].

Giản đồ khối của AT91SAM7S64 được mô tả trong hình 4.2.



Hình 4.2: **Giản đồ khối của vi điều khiển AT91SAM7S64.**

Vi điều khiển AT91SAM7S64 là một trong những vi điều khiển trong họ AT91SAM7S do hãng Atmel chế tạo, có kiến trúc thuộc dòng lõi xử lý ARM v4T và có lõi xử lý đặc trưng là ARM7TDMI. Họ vi điều khiển AT91SAM7S bao gồm:

- AT91SAM7S512 : 512 Kbytes bộ nhớ Flash.
- AT91SAM7S128 : 128 Kbytes bộ nhớ Flash.
- AT91SAM7S64 : 64 Kbytes bộ nhớ Flash.
- AT91SAM7S321/32 : 32 Kbytes bộ nhớ Flash.
- AT91SAM7S161/16 : 16 Kbytes bộ nhớ Flash.

Các tính năng cơ bản

- Lõi xử lý:

  Lõi ARM7TDMI theo kiến trúc RISC, Cấu trúc tập lệnh gồm: tập lệnh 32 bit thực hiện câu lệnh hiệu suất cao và tập lệnh 16 bit (Thumb) ưu tiên mã chương trình nhỏ gọn.

- Bộ nhớ:
  - 16 Kbytes SRAM;
  - 64 Kbytes Flash.

- Các giao tiếp ngoại vi:
  - Một cổng truyền USB 2.0 với tốc độ 12 Mbits/giây ;
  - Một bộ điều khiển đồng bộ nối tiếp (SSC);
  - Hai bộ giao tiếp UARTs;
  - Một bộ giao tiếp SPI;
  - Ba bộ Timer/Counter 16 bit;
  - Bốn bộ điều biến độ rộng xung: cho phép quản lý bốn kênh PWM 16 bit độc lập;
  - Một bộ TWI (Two-Wire Interface);
  - Tám kênh ADC 10 bit, trong đó bốn kênh dùng chung cho vào ra đa mục đích;
  - Một bộ điều khiển vào hoặc ra song song PIO quản lý 32 đường vào hoặc ra song song;
  - Một bộ Timer nội (PIT);
  - Một bộ Watchdog Timer;
  - Một bộ định thời thời gian thực 32 bit (RTT).

- Hệ thống:
  - Xung nhịp CPU tối đa có thể lên tới 55MHz ở 1,65V;
  - Ma trận các kênh truyền;

- Truy cập bộ nhớ trực tiếp (DMA – Direct Memory Access);
- Bộ điều khiển ngắt cao cấp (AIC – Advanced Interrupt Controller):
  - Cho phép lập trình độc lập nhiều nguồn ngắt với tám mức ưu tiên cho từng nguồn ngắt;
  - Có hai nguồn ngắt ngoài, một nguồn ngắt bên trong, các nguồn ngắt có chế độ bảo vệ chống các tạp nhiễu.
- Bộ quản lý nguồn (PMC – Power Management Controller) cho phép tiết kiệm trong việc sử dụng năng lượng, có khả năng tối ưu nguồn như trong chế độ chờ và trong chế độ nghỉ;

- Bộ tạo xung bao gồm:
  - Bộ tạo dao động tần số từ 3 đến 20 Mhz;
  - Bộ dao động chậm RC;
  - Bộ PLL cho phép tạo xung có tần số chính xác.

- Chế độ vào ra:

64 chân vào ra lập trình được.

- Nguồn cung cấp:

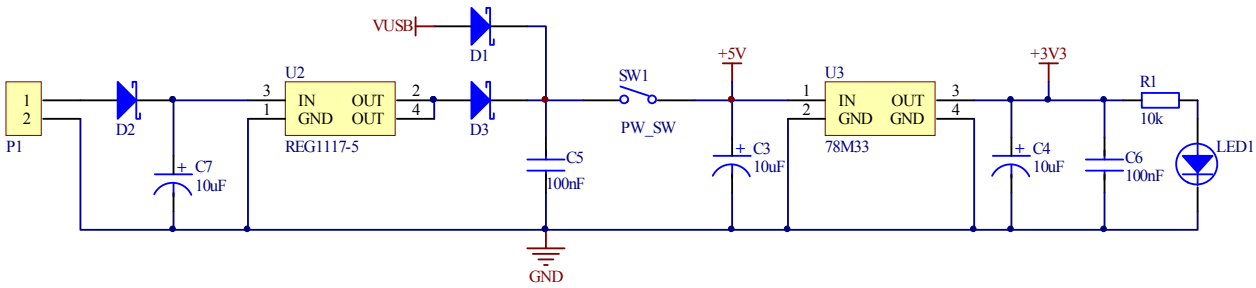
3,3VDC; bên trong có tích hợp một bộ điều chỉnh điện áp ra là 1,8VDC giúp tối ưu việc cấp nguồn trong vi điều khiển.

4.3 Khối nguồn cung cấp

Khối nguồn cung cấp cho toàn mạch

Khối nguồn có khả năng nhận nguồn ngoài từ 5 – 12VDC hoặc qua cổng USB trên máy tính.

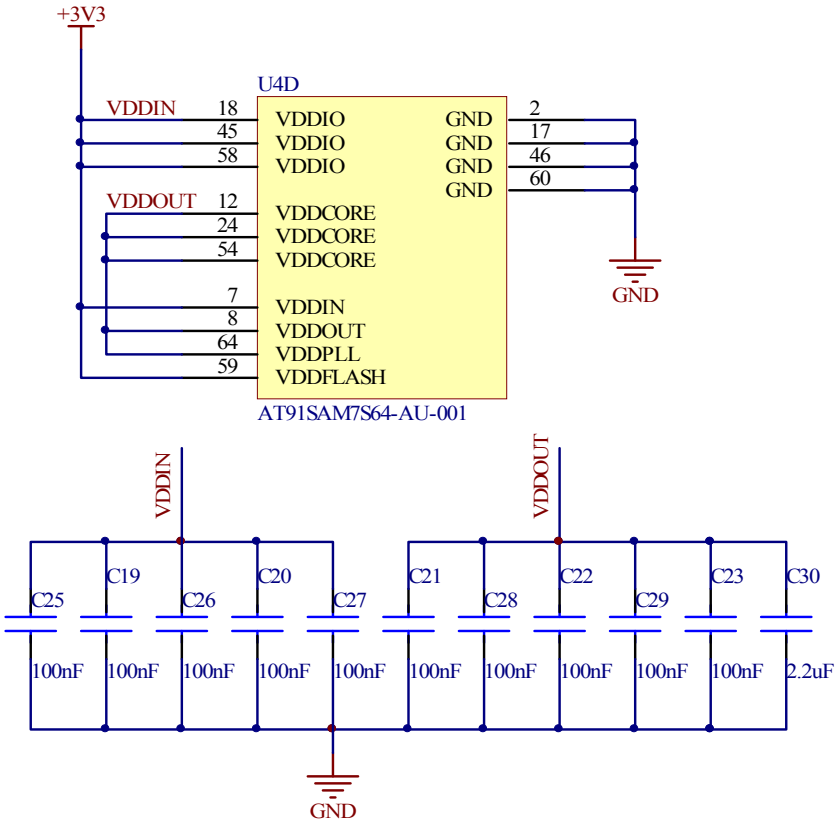
Khối nguồn có chức năng cung cấp điện áp và ổn áp ở mức 5VDC và 3,3VDC cho toàn mạch, được mô tả trong hình 4.3. Mạch tiêu thụ chủ yếu là LED 7 đoạn, do vậy tổng dòng điện tiêu thụ trên mạch lúc lớn nhất khoảng 150mA. Do đó mạch nguồn chỉ cần sử dụng các vi mạch NCP1117 ổn áp 5V với dòng 1A, 78M33 ổn áp 3,3V với dòng là 500mA và được thiết kế như sau:



Hình 4.3: **Sơ đồ nguyên lý mạch nguồn.**

Khối nguồn cung cấp trong vi điều khiển

Vi điều khiển AT91SAMS64 có sáu chân nguồn, được tích hợp thành bộ điều chỉnh điện áp cung cấp cho lõi ARM; bộ nhớ Flash, bộ nhân tần, bộ điều khiển vào ra bên trong vi điều khiển và được kết nối như trong hình 4.4.



Hình 4.4: **Sơ đồ mạch nguồn vào ra cho vi điều khiển.**

Chân VDDIN: cho phép điều chỉnh điện áp trong dải điện áp từ 3.0V đến 3.6V, nếu không sử dụng chân này để điều chỉnh thì điện áp định mức là 3.3V.

Chân VDDOUT: có đầu ra ổn áp mức 1.8V.

Chân VDDIO: các đường nguồn vào ra được cung cấp đồng thời, dải điện áp từ 3.0V đến 3.6V, điện áp định mức là 3.3V.

Chân VDDFLASH: đáp ứng về yêu cầu nguồn cho bộ nhớ Flash hoạt động chính xác. Dải điện áp từ 3.0V đến 3.6V, điện áp định mức là 3.3V.

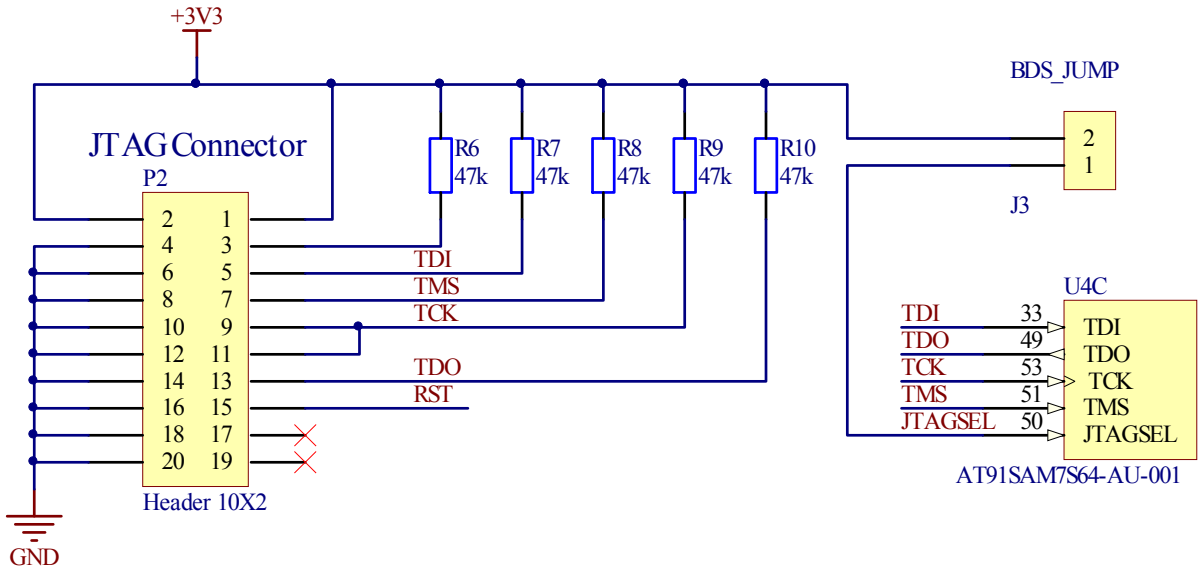
Các chân VDDCORE: cung cấp nguồn logic của thiết bị, dải điện áp từ 1.65V đến 1.95V, quy chuẩn là 1.8V. Có thể kết nối tới chân VDDOUT với các tụ khử ghép dùng để ngăn tín hiệu từ tầng này sang tầng kia. VDDCORE cung cấp nguồn cho thiết bị bao gồm cả bộ nhớ Flash.

Chân VDDPLL: nguồn cho bộ dao động và bộ PLL. Có thể kết nối trực tiếp tới chân VDDOUT.

Các nguồn của bộ điều chỉnh điện áp này không có các nguồn đất riêng biệt, chỉ dùng một nguồn đất chung. Trên mạch in các tụ lọc xuống đất và các chân nối đất của bộ điều chỉnh điện áp này phải được nối vào nguồn đất chung với khoảng cách ngắn nhất.

4.4 Cổng kết nối chuẩn JTAG

Cổng JTAG dùng để kết nối với mạch nạp cho vi điều khiển AT91SAM7S64, được mô tả trong hình 4.5.



Hình 4.5: **Sơ đồ mạch cổng kết nối chuẩn JTAG.**

TMS, TDI và TCK là các chân đầu vào schimitt trigger (có tác dụng chống nhiễu). TMS và TCK có khả năng giao tiếp được đến 5V, TDI thì không. Các chân TMS, TDI và TCK không tích hợp trở kéo.

TDO là tín hiệu bằng mức VDDIO và không có trở kéo.

Chân JTAGSEL được sử dụng để lựa chọn khi đã xác nhận ở mức cao thì sẽ chuyển sang chế độ Debug.

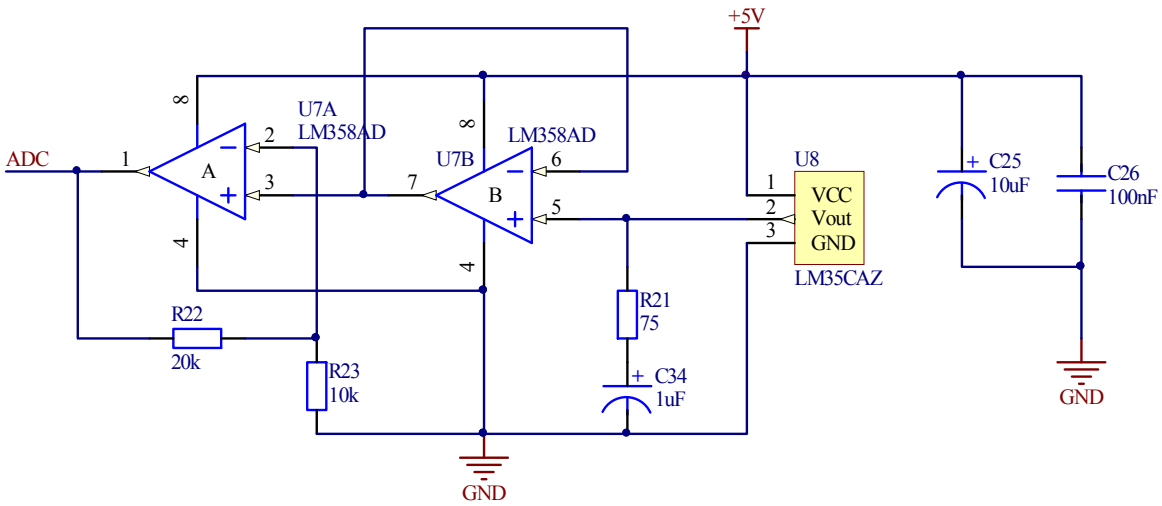
Các chân TMS, TDI, TCK và TDO đều phải mắc trở kéo lên dương nguồn để ổn định trạng thái xuất nhập dữ liệu.

4.5 Mạch cảm biến nhiệt độ

Sơ đồ mạch cảm biến nhiệt độ được mô tả trong hình 4.6. Mạch gồm hai khối:

- Khối cảm biến nhiệt độ;

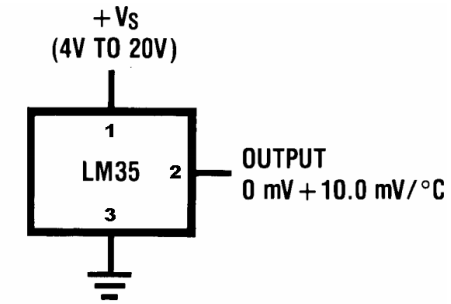
- Khối khuếch đại điện áp đầu ra của LM35 lên 3,3V.



Hình 4.6: **Sơ đồ mạch cảm biến nhiệt độ.**

Khối cảm biến nhiệt độ

Khối này sử dụng cảm biến LM35 [16], đây là một trong những IC thuộc họ cảm biến nhiệt độ tương tự, được sản xuất theo công nghệ bán dẫn dựa trên các chất bán dẫn dễ bị tác động bởi sự thay đổi của nhiệt độ, đầu ra của cảm biến là điện áp (V) tỉ lệ với nhiệt độ mà nó được đặt trong môi trường cần đo. Sơ đồ chân và dải điện áp ra của LM35 được mô tả trong hình 4.7.



Chân 1: Nguồn cung cấp từ 4 đến 20V;

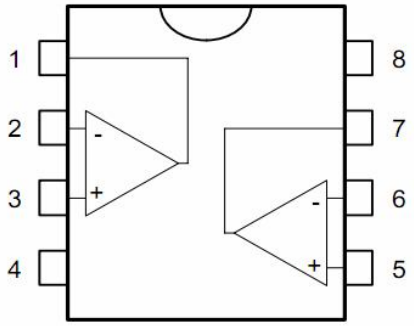
Chân 2: Điện áp ra từ 0 đến 1000mV theo thang 0mV + 10mV/^{o}C;

Chân 3: Nối đất.

Hình 4.7: **Sơ đồ chân và các giá trị điện áp vào ra của LM35.**

Khối mạch khuếch đại

Khối này sử dụng IC LM358AD [17], khuếch đại điện áp từ LM35 để phù hợp với thế chuẩn của ADC trên vi điều khiển. Sơ đồ LM358 được mô tả trong hình 4.8.



1 - Output 1
2 - Inverting input
3 - Non-inverting input
4 - V_{CC}^-
5 - Non-inverting input 2
6 - Inverting input 2
7 - Output 2
8 - V_{CC}^+

Hình 4.8: **Sơ đồ IC LM358AD và chức năng các chân tương ứng.**

Chức năng các chân IC LM358AD được mô tả trong bảng 4.1.

Bảng 4.1: **Chức năng các chân IC LM358AD.**

Chân	Chức năng
1	Đầu ra 1
2	Đầu vào đảo
3	Đầu vào không đảo
4	Đất
5	Đầu vào không đảo 2
6	Đầu vào đảo 2
7	Đầu ra 2
8	Nguồn dương

Giá trị ghép nối LM35 với ADC của vi điều khiển

Ta chọn bộ ADC 10 bit của AT91SAM7S64 → có 1024 mức lượng tử

Trường hợp 1: ghép nối thẳng LM35 với ADC

- Vref của ADC = 3.3V
- Dải đo LM35: 0 – 100 ^oC, 10mV/^oC
- V_{out} (điện áp ra LM35) từ 0V – 1V

→ Đầu vào ADC_{max} của LM35 = $\frac{1V}{3.3V}$ x 1024 = 320 mức

→ Mức tương ứng với nhiệt độ đo LM35: 1 mức = $\frac{1V}{320} \approx 3.1mV \approx 0.3\ ^oC$

Vậy nhiệt độ thay đổi ± 0.3 ^oC, V_{in} (ADC) thay đổi 3.1mV thì vi điều khiển mới phát hiện được có sự thay đổi nhiệt độ môi trường.

- Trường hợp 2: ghép nối LM35 qua khối mạch khuếch đại (IC LM358AD) đến bộ ADC.

Để giảm sai số, ta cho khuếch đại điện áp ra của LM35 lên ba lần để phù hợp với thế chuẩn của bộ ADC. Hệ số khuếch đại: K = 1 + $\frac{R2}{R1}$

Trong đó R2 = 20KΩ, R1 = 10KΩ → K = 3

Lúc này V_{in} (ADC) từ 0V – 3V

→ ADC_{max} = $\frac{3V}{3.3V}$ x 1024 = 931 mức

→ Mức tương ứng với nhiệt độ: 1 mức = $\frac{3V}{931} \approx 3.1mV$

→ Do qua bộ khuếch đại nên 1 mức LM35 = $\frac{3.1mV}{3} \approx 1mV \approx 0.1\ ^oC$

Vậy nhiệt độ thay đổi ± 0.1 ^{o}C, V_{in} (ADC) thay đổi 1mV thì vi điều khiển phát hiện được có sự thay đổi nhiệt độ môi trường, làm giảm sai số được ba lần so với trường hợp sử dụng trực tiếp đầu ra từ LM35.

Sử dụng thêm bộ khuếch đại ngoài tác dụng làm tăng độ phân dải còn có tác dụng làm bộ đệm trở kháng giữa LM35 và bộ ADC.

Việc đọc ADC được tính bởi công thức: $ADC = \frac{Vin}{Vref} \times 2^n$

Trong đó: Vin là điện áp được đưa vào từ mạch cảm biến nhiệt độ.

Vref = 3.3V (điện áp tham chiếu đầu vào của bộ ADC).

$2^n = 1024$ (n = 10: là số bit của bộ ADC).

Bước thay đổi của ADC là: $\frac{3.3V}{1024} = 3.2mV$

Tại 0 ^{o}C thì giá trị đầu ra của LM35 là 0mV tương ứng với ADC = 0;

Với ADC = 1 thì điện áp tương ứng là 3.2mV.

Mà LM35 thay đổi 10mV/^{o}C nên giá trị ADC thay đổi trong một đơn vị thì nhiệt độ thay đổi là: $\frac{3.2mV}{10mV} = 0.32$

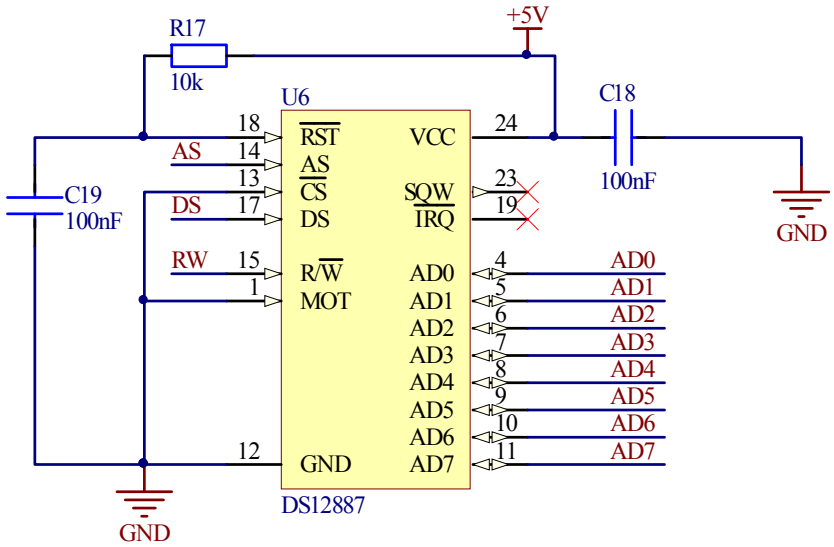
Vậy nhiệt độ đầu ra là: T = ADC * 0.32

4.6 Giao tiếp với IC thời gian thực DS12C887

DS12C887 là IC thời gian thực [18], có rất nhiều ưu điểm:

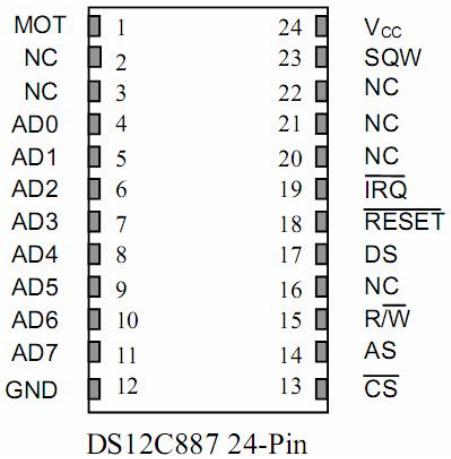
- Có độ chính xác cao;
- Dữ liệu thời gian được lưu trong bộ nhớ, có pin được tích hợp bên trong IC nên không bị mất dữ liệu khi mất nguồn cung cấp. Do vậy, dữ liệu được ghi vào thẻ nhớ luôn đảm bảo chính xác về thời gian.

Sơ đồ mạch kết nối IC DS12C887 được mô tả trong hình 4.9.



Hình 4.9: **Sơ đồ mạch kết nối IC DS12C887.**

Sơ đồ các chân IC DS12C887 được mô tả trong hình 4.10.



AD0-AD7 - Multiplexed Address/Data Bus
NC - No Connect
MOT - Bus Type Selection
$\overline{CS}$ - RTC Chip Select Input
AS - Address Strobe
R/$\overline{W}$ - Read/Write Input
DS - Data Strobe
$\overline{RESET}$ - Reset Input
$\overline{IRQ}$ - Interrupt Request Output
SQW - Square Wave Output
V_{CC} - +5 Volt Main Supply
GND - Ground

Hình 4.10: **Sơ đồ các chân IC DS12C887.**

Chức năng các chân IC DS12C887 được mô tả trong bảng 4.2.

Bảng 4.2: **Bảng ký hiệu và chức năng các chân DS12C887.**

Chân	Ký hiệu	Chức năng
1	MOT	Lựa chọn loại bus
2,3,16,20,21,22	NC	Bỏ trống
4-11	AD0-AD7	Bus đa hợp địa chỉ/dữ liệu
12	GND	Đất
13	$\overline{CS}$	Đầu vào lựa chọn RTC (Real Time Clock)
14	AS	Chốt địa chỉ
15	R/$\overline{W}$	Đầu vào đọc/ghi
17	DS	Chốt dữ liệu
18	$\overline{RESET}$	Đầu vào khởi động lại
19	$\overline{IRQ}$	Đầu ra yêu cầu ngắt
23	SQW	Đầu ra sóng vuông
24	VCC	Nguồn cung cấp 5V

Mô tả các chân tín hiệu của IC DS12C887

GND, V_{CC} – điện áp đầu vào 5V, khi điện áp được cung cấp đúng chuẩn 5V, thiết bị được truy cập đầy đủ và dữ liệu có thể đọc và ghi. Khi V_{CC} thấp hơn 4.25V, quá trình đọc và ghi bị cấm. Khi V_{CC} xuống thấp hơn 3V, RAM và bộ giữ giờ được chuyển sang nguồn năng lượng bên trong. Chức năng giữ thời gian duy trì độ chính xác vào khoảng ± một phút/tháng ở nhiệt độ 25^0C bất kể điện áp đầu vào ở chân V_{CC}.

MOT (Mode select) – chân này để lựa chọn giữa hai loại bus truyền dữ liệu. Khi được nối lên V_{CC}, bus định thời Motorola được chọn. Khi được nối xuống GND hoặc

không nối, bus định thời Intel được chọn. Chân này có điện trở kéo xuống bên trong, giá trị khoảng 20K.

SQW (Square Wave output) – Chân SQW có thể xuất tín hiệu ra từ một trong mười ba loại được cung cấp từ mười lăm trạng thái được chia bên trong của RTC. Tần số của chân SQW có thể thay đổi bằng cách lập trình. Tín hiệu SQW không xuất hiện khi V_{CC} thấp hơn 4.25V.

AD0 – AD7 (Multiplexed Bidirectional Address/Data Bus) – bus đa hợp tiết kiệm chân, vì thông tin địa chỉ và thông tin dữ liệu được dùng chung đường tín hiệu. Cùng tại những chân này, địa chỉ được xuất trong suốt phần thứ nhất của chu kỳ bus và được dùng cho dữ liệu trong phần thứ hai của chu kỳ, đa hợp địa chỉ/dữ liệu không làm chậm thời gian truy cập của DS12C887 khi bus chuyển từ địa chỉ sang dữ liệu xảy ra trong suốt thời gian truy cập RAM nội. Địa chỉ phải có giá trị trước khi xuất hiện sườn xuống của AS/ALE, tại thời điểm mà DS12C887 chốt địa chỉ từ AD0 đến AD6. Dữ liệu ghi phải được hiển thị và giữ ổn định trong suốt phần sau của xung DS hoặc WR. Trong chu kỳ đọc của DS12C887 đầu ra 8 bit của dữ liệu trong các phần sau của xung DS hoặc RD. Chu kỳ đọc được thực hiện xong và bus truyền trở về trạng thái tổng trở cao cũng như khi DS bắt đầu chuyển xuống thấp trong trường hợp định thời Motorola hoặc khi RD chuyển lên cao trong trường hợp định thời Intel.

AS (Address Strobe Input) – Xung dương cung cấp xung chốt địa chỉ trong việc phức hợp bus. Sườn xuống của AS/ALE làm cho địa chỉ bị chốt lại bên trong của DS12C887. Sườn lên tiếp theo khi xuất hiện trên bus AS sẽ xóa địa chỉ bất chấp chân CS có được chọn hay không. Lệnh truy cập có thể gửi tới bằng cả hai cách.

DS (Data Strobe or Read Input) – Chân DS/RD có hai kiểu sử dụng tùy thuộc vào mức của chân MOT. Khi chân MOT được kết nối lên Vcc, bus định thời Motorola được lựa chọn. Trong kiểu này DS là xung dương trong suốt phần sau của chu kỳ bus truyền và được gọi là Data Strobe. Trong suốt chu kỳ đọc, DS báo hiệu thời gian mà DS12C87 được điều khiển bus đôi. Trong chu kỳ đọc, sườn sau của DS làm DS12C887 chốt để dữ liệu được ghi. Khi chân MOT được nối xuống GND, bus định thời Intel được lựa chọn. Trong kiểu này, chân DS được gọi là Read (RD). RD xác định chu kỳ thời gian khi DS12C887 điều khiển bus đọc dữ liệu. Tín hiệu RD có cùng định nghĩa với tín hiệu cho phép xuất dữ liệu trong một bộ nhớ riêng.

R/$\overline{W}$ (Read/Write Input) – Chân R/$\overline{W}$ cũng có hai cách hoạt động. Khi chân MOT được kết nối lên V_{CC} cho chế độ định thời Motorola, R/$\overline{W}$ đang ở chế độ chỉ đọc hoặc là chu kỳ hiện tại là chu kỳ đọc hoặc ghi. Chu kỳ đọc đòi hỏi chân R/$\overline{W}$ phải ở mức cao khi chân DS ở mức cao. Chu kỳ ghi đòi hỏi chân R/$\overline{W}$ phải ở mức thấp trong suốt quá trình chốt tín hiệu của DS. Khi chân MOT được nối GND cho chế độ định thời Intel, tín hiệu R/$\overline{W}$ là tín hiệu hoạt động mức thấp được gọi là WR. Trong chế độ này, chân R/$\overline{W}$ được định nghĩa như tín hiệu cho phép ghi dữ liệu trong RAM chung.

CS (Chip Select Input) – Tín hiệu chọn lựa phải được xác định ở mức thấp ở chu kỳ bus truyền để DS12C887 được sử dụng. CS phải được giữ trong trạng thái hoạt động trong suốt DS và AS của chế độ định thời Motorola và trong suốt RD và WR của chế độ định thời Intel. Chu kỳ bus khi chọn vị trí mà không chọn CS sẽ chốt địa chỉ nhưng sẽ không có bất kỳ sự truy cập nào. Khi V_{CC} thấp hơn 4.25V, chức năng bên trong của DS12C887 ngăn chặn sự truy cập bằng cách không cho phép chọn lựa đầu vào CS. Quá trình này nhằm bảo vệ cả dữ liệu của đồng hồ thời gian thực bên trong cũng như dữ liệu RAM trong suốt quá trình mất nguồn.

$\overline{\text{IRQ}}$ (Interrupt Request Output) – Chân $\overline{\text{IRQ}}$ là đầu ra hoạt động mức thấp của DS12C887 mà có thể sử dụng như đầu vào ngắt tới bộ xử lý. Đầu ra $\overline{\text{IRQ}}$ ở mức thấp khi bit là nguyên nhân làm ngắt và phù hợp với bit cho phép ngắt được thiết lập. Để xóa chân $\overline{\text{IRQ}}$ chương trình của bộ vi xử lý thông thường được đọc ở thanh ghi C. Chân RESET cũng bị xóa trong lúc ngắt. Khi không có trạng thái ngắt nào được sử dụng, trạng thái $\overline{\text{IRQ}}$ ở trong trạng thái tổng trở cao. Nhiều thiết bị ngắt có thể nối tới một kênh truyền $\overline{\text{IRQ}}$. Kênh truyền $\overline{\text{IRQ}}$ là một đầu ra mở và yêu cầu một điện trở kéo lên bên ngoài.

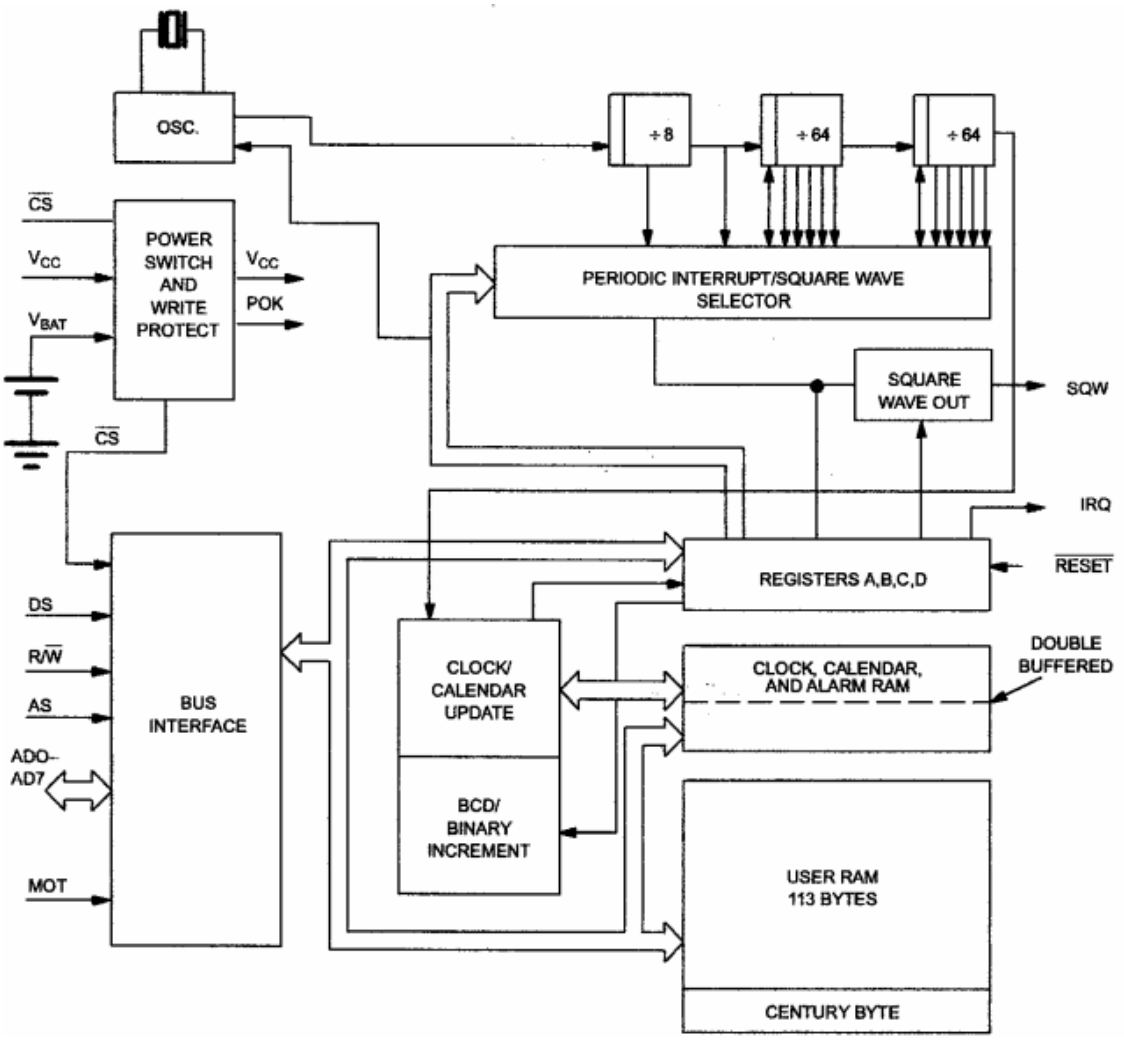
RESET (Reset Input) – Chân $\overline{\text{RESET}}$ không có hiệu lực đối với đồng hồ, lịch, hoặc là RAM. Ở chế độ cấp nguồn, chân $\overline{\text{RESET}}$ có thể bị kéo xuống trong thời gian cho phép để ổn định nguồn cung cấp. Thời gian mà chân $\overline{\text{RESET}}$ bị kéo xuống mức thấp phụ thuộc vào ứng dụng. Tuy nhiên nếu chân $\overline{\text{RESET}}$ được sử dụng ở chế độ cấp nguồn, thời gian $\overline{\text{RESET}}$ ở mức thấp có thể vượt quá 200ms ở mức thấp và V_{CC} ở trên 4.24V, những điều sau xảy ra:

- Bit cho phép ngắt định kỳ (PEIO: Periodic Interrupt Enable) được đặt ở mức 0.
- Bit cho phép ngắt chuông (AIE: Alamrm Interrupt Enable) được đặt ở mức 0.
- Bit cờ cho phép ngắt kết thúc cập nhật (UF: Update Ended Interrut Flag) được xóa về 0.
- Bit cờ trạng thái yêu cầu ngắt (IRQF: Interrupt Request Status Flag) được đặt ở mức 0.
- Bit cờ cho phép ngắt định kỳ (PF: Periodic Interrupt Flag) được đặt ở mức 0.
- Thiết bị không sử dụng được cho tới khi chân $\overline{\text{RESET}}$ trở lại mức logic 1.
- Bit cờ cho phép ngắt chuông (AF: Alarm Interrupt Flag) được đặt ở mức 0.
- Chân $\overline{\text{IRQ}}$ ở trong trạng thái tổng trở cao.
- Bit cho phép xuất sóng vuông (SQWE: Square Wave Output Enable) được đặt ở mức 0.
- Bit cho phép ngắt kết thúc cập nhật (UIE: Update Ended Interrupt Enable) được đặt ở mức 0.

Trong các ứng dụng thông thường chân $\overline{\text{RESET}}$ có thể được nối lên V_{CC}. Kết nối như vậy sẽ cho phép DS12C887 hoạt động và khi mất nguồn sẽ không làm ảnh hưởng đến bất kỳ thanh ghi điều khiển nào.

Cấu trúc bên trong IC DS12C887

Cấu trúc bên trong IC DS12C887 được mô tả trong hình 4.11.



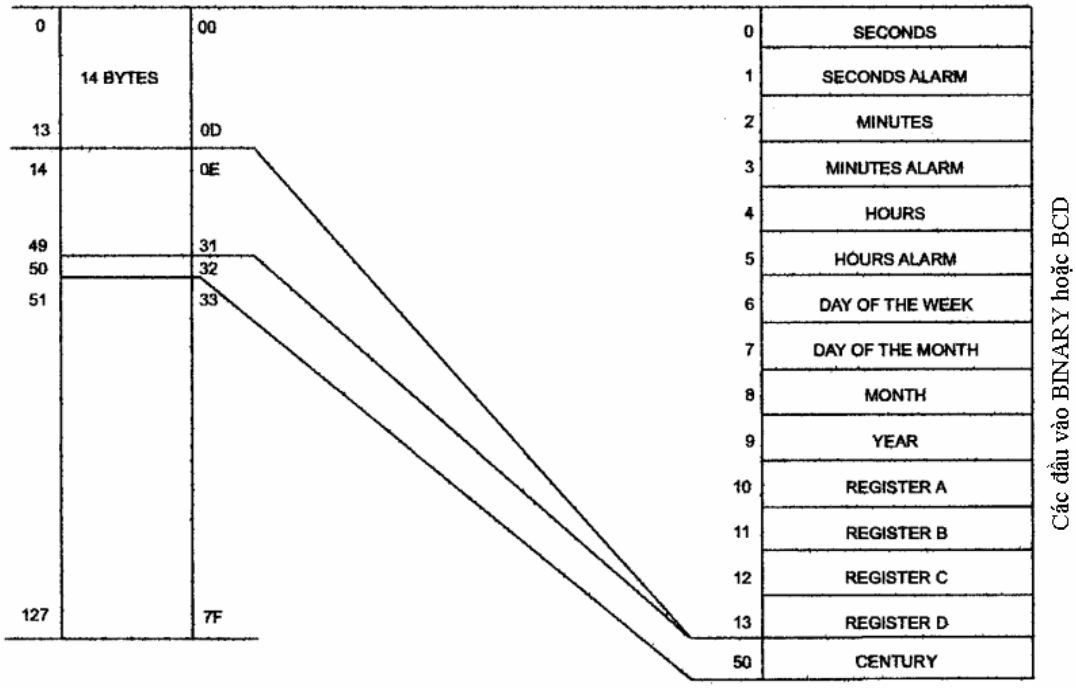
Hình 4.11: **Cấu trúc IC DS12C887.**

Bản đồ địa chỉ của DS12C887

Bản đồ địa chỉ của DS12C887 được mô tả trong hình 4.12. Bản đồ địa chỉ bao gồm 113 byte RAM thông dụng, 11 byte RAM mà thành phần bao gồm đồng hồ thời gian thực, lịch, dữ liệu báo giờ và 4 byte được sử dụng cho việc điều khiển và thông báo tình trạng. Tất cả 128 byte có thể được ghi hoặc đọc trực tiếp trừ những trường hợp sau:

- Thanh ghi C và D là hai thanh ghi chỉ đọc.
- Bit thứ bẩy của thanh ghi A là bit chỉ đọc.
- Bit cao của byte thứ hai là bit chỉ đọc.

Thời gian và lịch đã có được đọc bằng cách đọc các byte bộ nhớ hiện có. Thời gian, lịch và báo giờ được thiết lập hoặc gán giá trị bằng cách ghi giá trị các byte RAM thích hợp. Nội dung của mười byte chứa thời gian, lịch và báo giờ đều có thể hiển thị ở cả hai dạng: nhị phân (Binary) hoặc BCD (Binary – Coded Decimal). Trước khi ghi lên các thanh ghi thời gian, lịch, và các thanh ghi báo giờ bên trong, bit SET ở thanh ghi B phải được thiết lập ở mức logic 1 để ngăn ngừa sự cập nhật có thể xảy ra trong quá trình ghi đè. Thêm vào nữa để ghi lên mười thanh ghi chỉ thời gian, lịch, và thanh ghi báo giờ ở một định dạng đã lựa chọn (BCD hay nhị phân), bit chọn kiểu dữ liệu (DM) của thanh ghi B phải được đặt ở mức logic thích hợp lên tất cả mười byte dữ liệu. Bit lựa chọn kiểu hiển thị 24/12 là bit không thể thay đổi và không khởi động lại thanh ghi. Khi định dạng 12 giờ được lựa chọn, bit cao của byte giờ luôn được truy cập bởi vì chúng được đệm gấp đôi. Mỗi một giây, mười một byte được nâng cấp và được kiểm tra tình trạng báo giờ một lần.



Hình 4.12: **Bản đồ địa chỉ DS12C887.**

Các thanh ghi điều khiển

DS12C887 có bốn thanh ghi điều khiển được sử dụng vào mọi lúc kể cả trong quá trình cập nhật.

- Thanh ghi A

MSB **LSB**

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
UIP	DV2	DV1	DV0	RS3	RS2	RS1	RS0

Hình 4.13: **Vị trí các bit trong thanh ghi A.**

UIP: Update In Progress: là bit trạng thái mà có thể theo dõi được. Khi bit UIP ở mức 1, quá trình cập nhật sẽ sớm xảy ra. Khi bit UIP ở mức 0, quá trình cập nhật sẽ không xảy ra ít nhất là 244 s. Những thông tin về thời gian, lịch, và báo giờ ở trong RAM có đầy đủ cho việc truy cập khi bit UIP ở mức 0. Bit UIP là bit chỉ đọc và không bị ảnh hưởng của chân $\overline{\text{RESET}}$. Khi ghi bit SET ở thanh ghi B lên 1, thì sẽ ngăn chặn mọi quá trình cập nhật và xóa bit trạng thái UIP.

DV2, DV1, DV0: ba bit trên được sử dụng để bật hoặc tắt bộ dao động và cài đặt lại quá trình đếm xuống. Khi được đặt 010 thì đó là sự kết hợp duy nhất để bật bộ dao động lên và cho phép RTC giữ thời gian. Khi được đặt 11X sẽ cho phép dao động nhưng giữ quá trình đếm xuống ở mức Reset. Quá trình cập nhật tiếp theo sẽ diễn ra sau 500 ms, sau khi định dạng 010 được ghi vào ba bit DV0, DV1, và DV2.

RS3, RS2, RS1, RS0: bốn bit loại lựa chọn để chọn lựa một trong mười ba loại của bộ chia mười lăm trạng thái hoặc không cho phép xuất ra tín hiệu chia ra ngoài. Loại được lựa chọn có thể phát ra sóng vuông (chân SQW) hoặc ngắt theo chu kỳ. Người sử dụng có thể sử dụng một trong những cách sau:

- Cho phép ngắt với bit PIE
- Cho phép xuất đầu ra tại chân SQW với bit SQWE
- Cho phép cả hai hoạt động cùng một lúc và cùng một loại.
- Không kích hoạt cả hai.

- Thanh ghi B

MSB **LSB**

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
SET	PIE	AIE	UIE	SQWE	DM	24/12	DSE

Hình 4.14: **Vị trí các bit trong thanh ghi B.**

Set: Khi bit SET ở mức 0, thông thường quá trình cập nhật bằng cách tăng biến đếm một lần một giây. Khi bit SET được ghi vào mức 1, mọi quá trình cập nhật đều bị cấm, và chương trình có thể khởi động các byte thời gian và lịch mà không có quá trình cập nhật nào xảy ra trong quá trình khởi động. Chu kỳ đọc có thể thực thi ở cùng một kiểu. SET là bit đọc hoặc ghi và không chịu ảnh hưởng trạng thái RESET hoặc các chức năng bên trong của DS12C887.

PIE (Periodic Interrupt Enable): Bit cho phép ngắt theo chu kỳ là bit đọc hoặc ghi, nó cho phép bit cờ ngắt theo chu kỳ (PF: Periodic Interrupt Flag) trong thanh ghi C để điều khiển chân $\overline{\text{IRQ}}$ xuống mức thấp tùy thuộc vào tỉ lệ phân bố của bit RS3 – RS0 ở thanh ghi A. Khi bit PIE = 0 thì đầu ra $\overline{\text{IRQ}}$ bị cấm ngắt, nhưng cờ PF vẫn được thiết lập theo chu kỳ ngắt. PIE không bị tác động bởi bất kỳ chức năng nội nào của DS12C887 nhưng được xóa về 0 bởi chân $\overline{\text{RESET}}$.

AIE (Alarm Interrupt Enable): Bit cho phép ngắt báo giờ, là bit đọc/ghi mà khi được thiết lập lên 1 sẽ cho phép bit cờ báo giờ (Alarm Flag (AF)) ở thanh ghi C, để cho phép ngắt $\overline{IRQ}$. Tín hiệu ngắt báo giờ diễn ra ở tất cả các giây khi cả ba byte báo giờ chứa mã báo giờ "don't care" được thể hiện ở nhị phân như sau 11XXXXXX. Các chức năng bên trong của DS12C887 không bị ảnh hưởng bởi bit AIE.

UIE (Update Ended Interrupt Enable): Bit cho phép kết thúc quá trình ngắt cập nhật, là bit đọc/ghi mà cho phép bit cờ kết thúc quá trình cập nhật ở thanh ghi C để cho phép ngắt $\overline{IRQ}$. Chân $\overline{RESET}$ ở mức 0 hoặc chân SET ở mức 1 sẽ xóa bit UIE.

SQWE (Square Wave Enable): Khi bit cho phép xuất sóng vuông được thiết lập lên mức 1, một tín hiệu sóng vuông có tần số được đặt bởi vị trí được lựa chọn của bit RS3 đến RS0 sẽ điều khiển sóng ra tại chân SQW. Khi bit SQWE được thiết lập ở mức thấp, chân SQW sẽ được giữ ở mức thấp. SQWE là bit đọc hoặc ghi và được xóa khi RESET. SQWE được thiết lập lên 1 khi được cấp Vcc.

DM (Data Mode): Bit kiểu dữ liệu quy định khi nào thì thông tin lịch và thời gian ở định dạng nhị phân hoặc định dạng BCD. Bit DM được thiết lập bởi chương trình để có định dạng thích hợp. Bit này không bị thay đổi bởi các chức năng bên trong hoặc chân $\overline{RESET}$. Mức 1 của DM sẽ hiển thị dữ liệu nhị phân còn mức 0 hiển thị dữ liệu dạng BCD.

24/12: Bit điều khiển 24/12 xác định kiểu byte giờ. Khi ở mức 1 thì chỉ chế độ hiển thị 24 giờ, còn ở mức 0 thì chỉ chế độ hiển thị 12 giờ. Bit này là bit đọc ghi và không bị ảnh hưởng bởi các chức năng bên trong cũng như chân $\overline{RESET}$.

DSE (Daylight Saving Enable): Bit cho phép nhớ công khai, là bit đọc hoặc ghi, nó cho phép hai cập nhật đặc biệt khi DSE được thiết lập lên 1. Một là vào chủ nhật đầu tiên của tháng tư, thời gian sẽ tăng từ 1:59:59 AM lên 3:00:00 AM. Hai là vào chủ nhật cuối cùng của tháng mười, khi thời gian lần đầu tiên đạt được 1:59:59 AM thì nó sẽ đổi thành 1:00:00 AM. Chức năng đặc biệt này sẽ không được thực thi nếu bit DSE ở mức 0. Bit này không bị ảnh hưởng bởi các chức năng bên trong cũng như chân $\overline{RESET}$.

- Thanh ghi C

MSB **LSB**

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
IRQF	PF	AF	UF	0	0	0	0

Hình 4.15: **Vị trí các bit trong thanh ghi C.**

IRQF (Interrupt Request Flag): Bit cờ yêu cầu ngắt được thiết lập lên 1 khi một trong những điều dưới đây đúng:

- PF = PIE = 1
- AF = AIE = 1
- UF = UIE = 1

Điều đó có nghĩa là IRQF = (PF.PIE) - (AF.AIE) - (UF.UIE)

Bất cứ lúc nào bit IRQF đều ở mức 1, chân $\overline{IRQ}$ được đặt xuống mức thấp. Bit cờ PF, AF và UF được xóa khi thanh ghi C được chương trình đọc hoặc chân $\overline{RESET}$ ở mức thấp.

PF (Periodic Interrupt Flag): Bit cờ ngắt theo chu kỳ, là bit chỉ đọc, được thiết lập lên mức 1 khi có một sườn xung được phát hiện ở tín hiệu lựa chọn của bộ chia. Từ bit RS3 đến RS0 xác định chu kỳ. PF được thiết lập lên 1 bất chấp trạng thái của bit PIE. Khi cả PF và PIE đều ở mức 1, tín hiệu $\overline{IRQ}$ được kích hoạt và sẽ thiết lập bit IRQF lên mức 1. Bit PF sẽ bị xóa bằng phần mềm đọc thanh ghi C hoặc chân $\overline{RESET}$.

AF (Alarm Interrupt Flag): Mức 1 của bit cờ cho phép ngắt báo giờ chỉ ra rằng thời gian hiện tại được so sánh với thời gian báo giờ. Nếu bit AIE còn ở mức 1, chân $\overline{IRQ}$ sẽ xuống mức thấp và bit IRQF sẽ thiết lập lên 1. Khi $\overline{RESET}$ hoặc đọc thanh ghi C sẽ xóa AF.

UF (Update Ended Interrupt Flag): Bit cờ ngắt kết thúc cập nhật được đặt sau mỗi chu kỳ cập nhật. Khi bit UIE được thiết lập lên 1, mức 1 ở UF sẽ làm cho bit IRQF lên mức 1, nó sẽ xác định trạng thái chân $\overline{IRQ}$. UF sẽ bị xóa khi thanh ghi C được đọc hoặc có tín hiệu RESET.

Từ bit 3 đến bit 0: là những bit không sử dụng của thanh ghi trạng thái C, những bit này luôn ở mức 0 và không thể ghi đè.

- Thanh ghi D

MSB **LSB**

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
VRT	0	0	0	0	0	0	0

Hình 4.16: **Vị trí các bit trong thanh ghi D.**

VRT (Valid RAM and Time) – Bit thời gian và RAM hợp lệ, biểu hiện tình trạng của pin, được kết nối chân VBAT. Bit này không phải là bit ghi được và luôn có giá trị bằng 1 khi đọc. Nếu hiển thị mức 0, nguồn năng lượng pin bên trong đã cạn và cả hai mục dữ liệu thời gian thực lẫn dữ liệu RAM đều phải xem xét. Bit này không chịu ảnh hưởng bởi chân RESET.

Bit 6 - bit 0: Những bit này không được sử dụng, chúng không ghi được và khi đọc thì luôn có giá trị bằng 0.

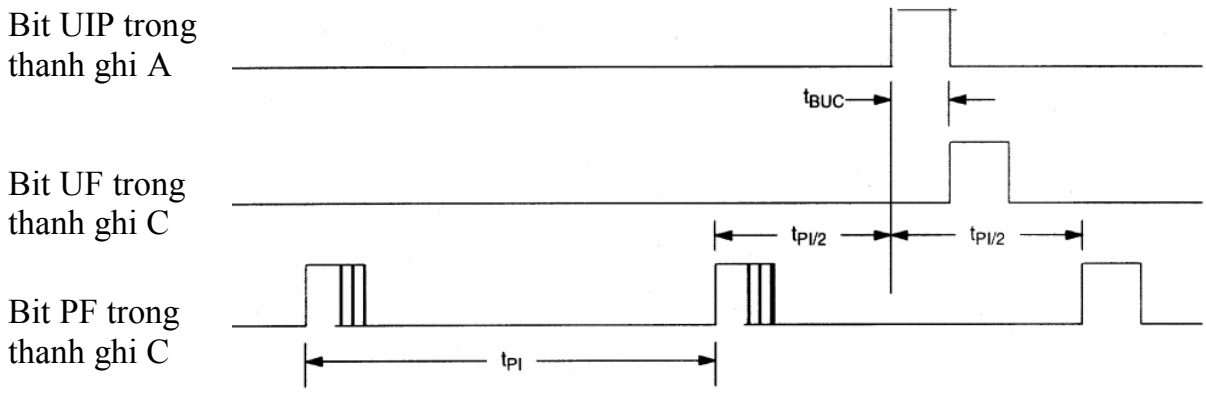
Chu kỳ cập nhật

DS12C887 thực hiện một chu kỳ cập nhật mỗi lần một giây bất chấp bit SET ở thanh ghi B. Khi bit SET ở thanh ghi B được thiết lập lên 1, bộ phận sao chép từ bộ đệm các byte thời gian, lịch, báo giờ sẽ không hoạt động và sẽ không cập nhật thời gian khi thời gian tăng lên. Tuy nhiên, quá trình đếm giờ vẫn tiếp tục để cập nhật bộ nhớ nội cho việc sao chép vào bộ đệm. Hoạt động này cho phép thời gian vẫn duy trì độ chính xác mà không phụ thuộc quá trình đọc hoặc ghi bộ đệm thời gian, lịch và báo giờ và cũng chắc chắn rằng những thông tin về thời gian và lịch là phù hợp. Chu kỳ cập nhật cũng so sánh những byte báo giờ với những byte thời gian tương ứng và kết quả là có báo giờ nếu giống nhau hoặc là mã "don't care" được đặt cho tất cả ba vị trí. Có ba cách có thể điều khiển truy cập đồng hồ thời gian thực mà tránh được bất kỳ khả năng truy cập các dữ liệu về thời gian và lịch mâu thuẫn với nhau.

Cách thứ nhất sử dụng ngắt kết thúc cập nhật. Nếu được kích hoạt, một tín hiệu ngắt sẽ xảy ra sau mỗi chu kỳ cập nhật mà chỉ ra rằng có hơn 999 ms để đọc những thông tin về thời gian và ngày tháng thực. Nếu ngắt này được sử dụng, bit IRQF ở thanh ghi C phải được xóa trước khi bỏ những ngắt thường lệ.

Cách thứ hai sử dụng bit cập nhật đang được tiến hành. Bit UIP sẽ phát xung mỗi lần một giây. Sau khi bit UIP lên mức cao, quá trình cập nhật tiến hành sau 244 s. Nếu bit UIP ở mức thấp, nó cần ít nhất 244 s trước khi dữ liệu thời gian hoặc lịch thay đổi. Chính vì vậy, người sử dụng có thể tránh được những phục vụ ngắt thông thường để đọc đúng dữ liệu thời gian hoặc lịch.

Cách thứ ba sử dụng ngắt theo chu kỳ để xác định khi có một chu kỳ cập nhật. Bit UIP ở thanh ghi A được thiết lập lên mức 1 trong khi đặt bit PF ở thanh ghi C, được mô tả trong hình 4.17. Ngắt theo chu kỳ xuất hiện làm cho một phần lớn hơn của t_{BUC} cho phép thông tin thực về thời gian và lịch có thể đạt được tại tất cả nơi xảy ra của chu kỳ ngắt. Quá trình đọc sẽ hoàn thành trong một chu kỳ ($t_{PI/2}$ - t_{BUC}) để chắc chắn rằng dữ liệu không được đọc trong suốt quá trình cập nhật.



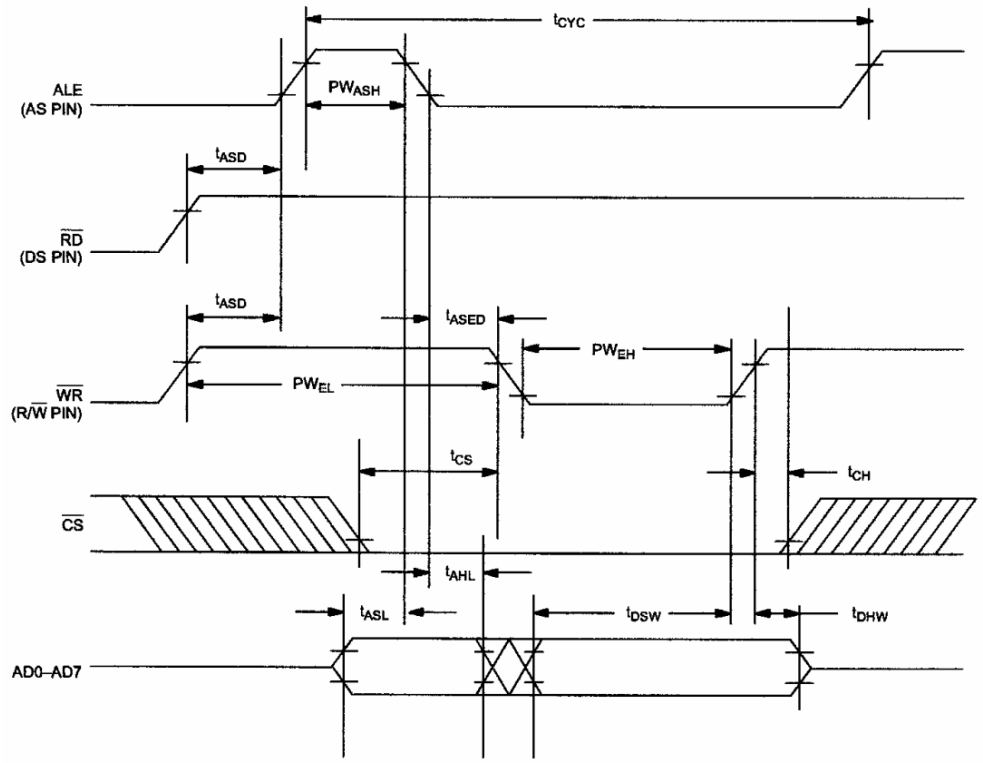
t_{PI} = ngắt theo chu kỳ định thời bên trong

t_{BUC} = thời gian trễ trước chu kỳ cập nhật = 244 s

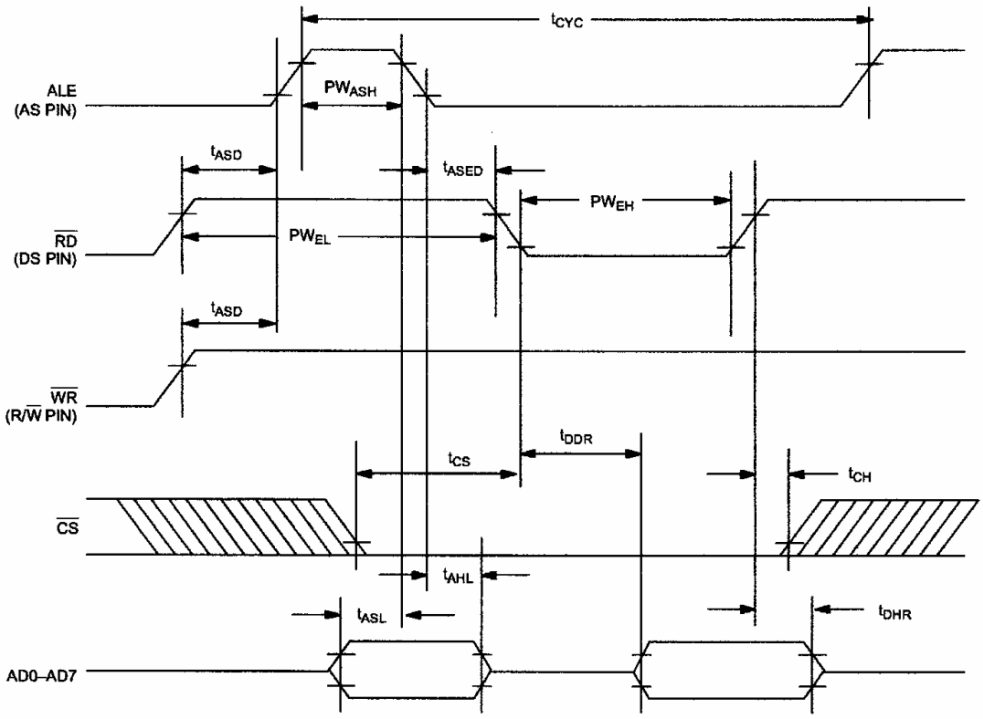
Hình 4.17: **Quan hệ ngắt theo chu kỳ và thời gian cập nhật.**

Trong mạch thực nghiệm, ta sử dụng cách ngắt theo chu kỳ để truy cập đồng hồ thời gian thực. Truyền và nhận dữ liệu theo kiểu BCD và truyền thông theo kiểu bus định thời Intel. Chế độ hiển thị thời gian kiểu định dạng 24 giờ.

Chu kỳ ghi và đọc theo kiểu bus định thời Intel được mô tả trong hình 4.18 và hình 4.19.



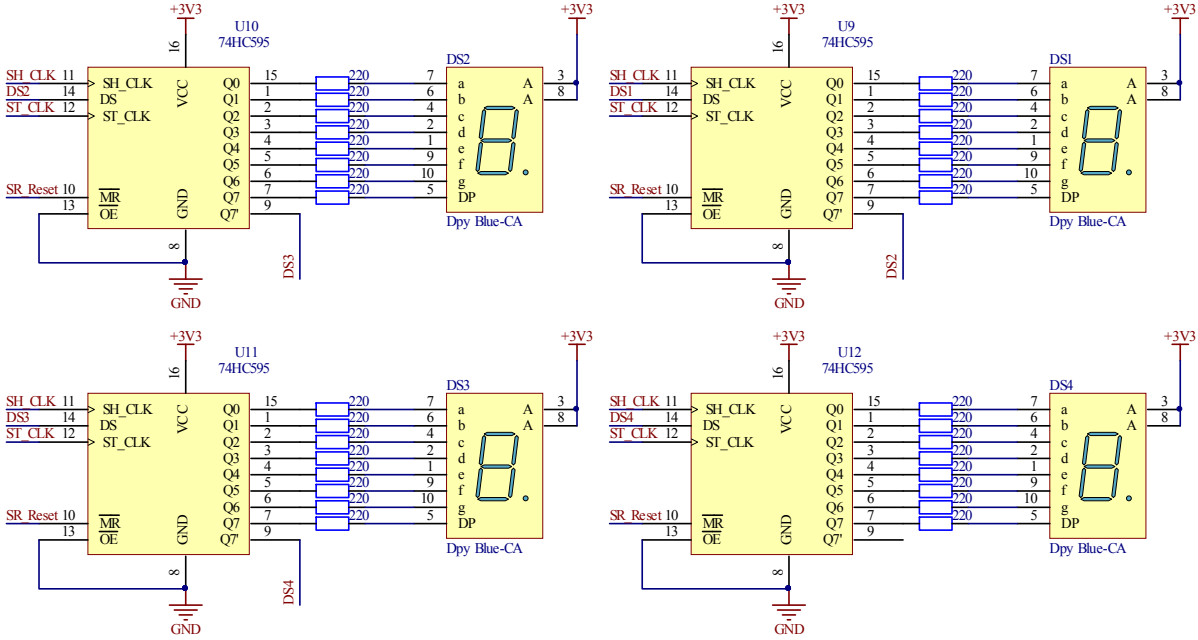
Hình 4.18: **Chu kỳ ghi theo kiểu bus định thời Intel.**



Hình 4.19: **Chu kỳ đọc theo kiểu bus định thời Intel.**

4.7 Hiển thị dữ liệu trên LED 7 đoạn

Để hiển thị nhiệt độ và thời gian thực, ta sử dụng bốn IC ghi dịch 74HC595 và bốn LED 7 đoạn loại Anode chung và được kết nối với vi điều khiển như trong hình 4.20.



Hình 4.20: **Sơ đồ mạch kết nối điều khiển LED 7 đoạn.**

Bốn LED 7 đoạn có hai chức năng: một là hiển thị nhiệt độ, hai là hiển thị thời gian thực.

Trong chức năng hiển thị nhiệt độ:

- LED DS1 hiển thị hàng chục;
- LED DS2 hiển thị hàng đơn vị;
- LED DS3 hiển thị số thập phân;
- LED DS4 hiển thị ^{o}C.

Trong chức năng hiển thị thời gian thực:

- LED DS1 và DS2 hiển thị giờ;
- LED DS3 và DS4 hiển thị phút.

Các chân a, b, c, d, e, f, g, DP của bốn LED 7 đoạn được nối riêng biệt vào các đường dữ liệu ra của từng IC 74HC595. Các chân này được nối qua các điện trở hạn dòng để bảo vệ LED, vì bên trong LED 7 đoạn là các LED đơn, dòng qua LED không được vượt quá 20mA. Việc tính toán điện trở hạn dòng phải phù hợp để đảm bảo không để dòng thấp quá hoặc lớn hơn 20mA khiến LED không sáng rõ hoặc quá dòng gây hỏng LED.

Để LED 7 đoạn sáng bình thường thì điện áp cấp trên LED là 2V và dòng qua LED là 6 mA.

Điện áp vào là 3.3V $\Rightarrow$ điện áp rơi trên điện trở hạn dòng là:

$V_R = 3.3 - 2 = 1.3V$.

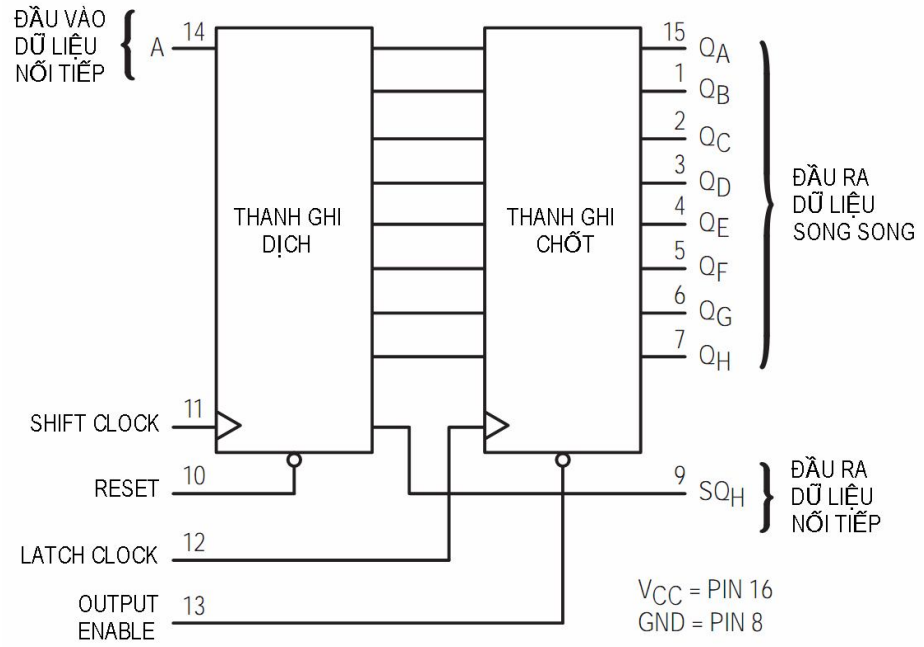
Giá trị điện trở hạn dòng là:

$R = \frac{1300mV}{6mA} = 216.6$, ta chọn R = 220Ω.

Một số đặc điểm của IC ghi dịch 74HC595 [19]

74HC595 là IC ghi dịch nối tiếp sang song song, được ứng dụng để tiết kiệm chân ra cho vi điều khiển. Có nhiều phương pháp để tiết kiệm chân như dùng IC giải mã, tuy nhiên IC ghi dịch 74HC595 được lựa chọn vì có một số ưu điểm sau:

- Số lượng chân điều khiển luôn cố định là bốn chân: nếu dùng giải mã thì số lượng chân điều khiển sẽ tăng theo số lượng chân đầu ra (khi phải ghép nhiều IC lại với nhau). Trong khi dùng ghi dịch thì số lượng chân điều khiển luôn là cố định ngay cả khi cần ghép nhiều IC lại với nhau.
- Cho phép điều khiển linh hoạt và ổn định hơn: giữa các thanh ghi dịch và đầu ra có một "chốt". Điều này cho phép thay đổi linh hoạt dữ liệu trong các thanh ghi dịch và ổn định trạng thái logic đầu ra.
- Giản đồ khối IC 74HC595 được mô tả trong hình 4.21.



Hình 4.21: **Giản đồ khối của IC 74HC595.**

Chức năng của IC 74HC595 được mô tả trong bảng 4.3.

Bảng 4.3: **Chức năng các chân IC 74HC595.**

Chức năng	Chân	Ký hiệu	Mô tả
Đầu vào dữ liệu	14	A (Data Serial: DS)	Đầu vào dữ liệu dạng nối tiếp
Đầu vào điều khiển	11	Shift Clock	Điều khiển quá trình dịch dữ liệu. Một mức chuyển tiếp sườn xung từ thấp lên mức cao, dữ liệu chân đầu vào nối tiếp chuyển vào trong thanh ghi dịch 8 bit.
Reset	10	Reset	Được kích hoạt ở mức sườn xung thấp, tác động lên quá trình xóa dữ liệu. Không ảnh hưởng đến 8 bit chốt.
Xung chốt	12	Latch Clock	Điều khiển chốt dữ liệu. Một mức chuyển tiếp sườn xung từ mức thấp lên mức cao sẽ chốt dữ liệu thanh ghi dịch
Cho phép xuất dữ liệu	13	Output Enable	Kích hoạt ở mức sườn xung thấp, cho phép dữ liệu từ thanh ghi chốt ra các đầu ra song song. Đầu ra nối tiếp không bị ảnh hưởng bởi bộ điều khiển này.
Các đầu ra dữ liệu song song	15,1-7	Q_A-Q_H	Các đầu ra không đảo và bộ chốt dữ liệu 8 bit đầu ra.
Đầu ra dữ liệu nối tiếp	9	SQ_H	Chân đưa dữ liệu nối tiếp ra ngoài dùng để ghép nhiều IC lại với nhau
Nguồn dương	16	V_{CC}	Nguồn cung cấp từ 2 - 6V
Nguồn âm	8	GND	Nối đất (0V)

Nguyên lý hoạt động của IC 74HC595

Ta có thể điều khiển được một hoặc nhiều IC 74HC595 ghép với nhau thông qua bốn chân Data Serial (DS), Shift Clock (SH_CLK), Latch Clock (ST_CLK)và Reset. Tuy nhiên nhược điểm của IC này là thời gian truy xuất các đầu ra chậm hơn so với việc truy xuất trực tiếp, vì dữ liệu phải được đưa từng bit vào IC trước khi cho xuất ra ngoài.

Trong sơ đồ ghép nối trong hình 4.21 ở trên, ta ghép nối bốn IC ghi dịch lại với nhau. Các chân điều khiển (SH_CLK, ST_CLK và Reset) được nối chung lại với nhau, chân dữ liệu nối tiếp đầu ra (SQ_H) của IC này được nối với chân dữ liệu nối tiếp đầu vào của IC tiếp theo.

Bảng 4.4: **Bảng chân lý IC 74HC595.**

Latch Clock (ST_CLK	Shift Clock (SH_CLK)	$\overline{\text{RESET}}$	$\overline{\text{OE}}$	Chức năng
X	X	X	H	Đầu ra QA-QH
X	X	L	L	Xóa thanh ghi dịch: $SQ_H = 0$
X	↑	H	L	Khóa thanh ghi dịch: $Q_N=Q_{n-1}$, Q_0=DS
↑	X	H	L	Các bit của thanh ghi dịch được chuyển tới đầu ra chốt.

Ghi chú:

H = mức điện áp cao; L = mức điện áp thấp

↑ = mức chuyển đổi trạng thái từ điện áp thấp lên điện áp cao

X = mức điện áp bất kỳ

Cách điều khiển IC được thể hiện thông qua bảng chân lý (bảng 4.4). Trước tiên đưa một bit dữ liệu vào chân SD_{in}, sau đó tạo ra một xung dương ở chân SH_CLK để dịch bit dữ liệu đó vào. Trạng thái logic của chân SD_{in} khi kích xung dương quyết định mức logic của bit được dịch vào. Quá trình này được lặp đi lặp lại liên tục cho đến khi toàn bộ dữ liệu được dịch vào trong IC. IC tiếp theo sẽ tiếp tục dịch dữ liệu vào từ chân SD_{out} của của IC trước đó.

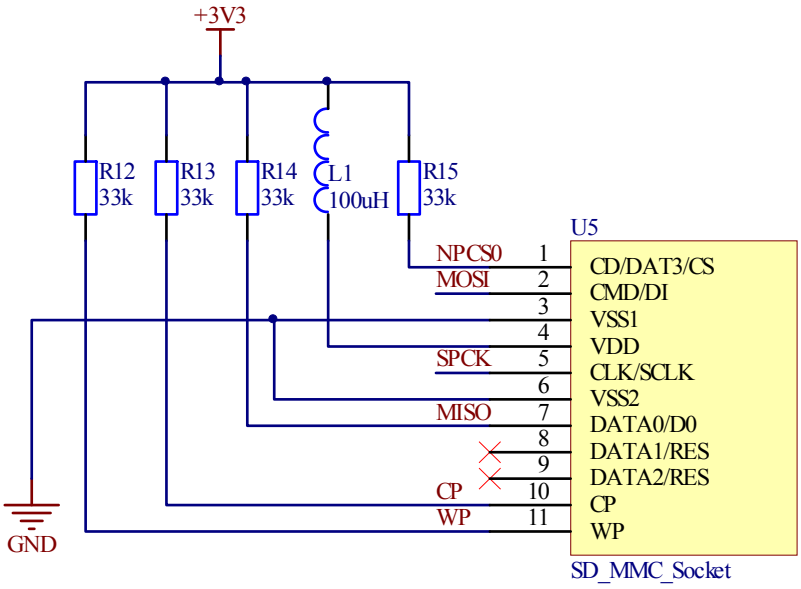
Khi quá trình dịch dữ liệu hoàn tất, cấp một xung dương ở chân ST_CLK, dữ liệu sẽ được đưa ra ngoài bởi các chân đầu ra Q_A-Q_H.

Cần chú ý khởi tạo 74HC595 bằng cách xóa các đầu ra và đưa chân Reset lên mức logic cao (do chân Reset tác động sườn thấp), nếu không các đầu ra sẽ luôn ở trạng thái logic 0 (bị xóa).

4.8 Giao tiếp với SD Card

Để lưu dữ liệu ta sử dụng SD Card [20], đây là loại thẻ nhớ thông dụng và được sử dụng rộng rãi. Sơ đồ kết nối các chân của SD Card được mô tả trong hình 4.22.

Ta sử dụng giao tiếp chuẩn SPI để giao tiếp với SD Card. Đây là giao tiếp rất phổ biến trong vi điều khiển và khối SPI này sẵn có trong vi điều khiển ARM.



Hình 4.22: **Sơ đồ mạch giao tiếp với vi điều khiển với SD Card.**

SD Card có ba loại kích thước khác nhau:

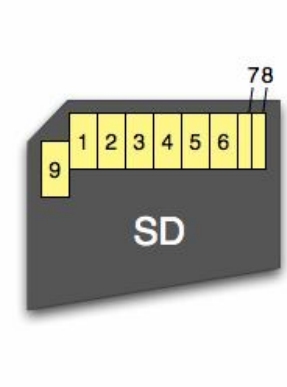
- SD Card thông thường;
- miniSD (SD Card loại nhỏ);
- microSD (SD Card loại siêu nhỏ).

Bảng 4.5 mô tả ba dạng khác nhau của SD Card.

Bảng 4.5: **So sánh các loại SD Card.**

Đặc tính	SD	miniSD	microSD
Chiều rộng	24 mm	20 mm	11 mm
Chiều dài	32 mm	21.5 mm	15 mm
Độ dày	2.1 mm	1.4 mm	1 mm
Trọng lượng	2 g	1 g	0.5 g
Nguồn hoạt động	2.7 – 3.6V	2.7 – 3.6V	2.7 – 3.6V
Số chân	9	11	8

Các chân kết nối chuẩn của SD Card được mô tả trong hình 4.23.



Pin	SD	SPI
1	CD/DAT3	CS
2	CMD	DI
3	VSS1	VSS1
4	VDD	VDD
5	CLK	SCLK
6	VSS2	VSS2
7	DAT0	DO
8	DAT1	X
9	DAT2	X

Hình 4.23: **Ký hiệu các chân kết nối của SD Card trong chế độ giao tiếp SPI.**

Trong giao tiếp SPI, sử dụng kết nối các chân SD Card theo bảng 4.6.

Bảng 4.6: **Chức năng các chân của SD Card trong chế độ giao tiếp SPI.**

Chân	Ký hiệu chân	Chức năng trong chế độ SPI
1	CD/DAT3	CS - Chip Select (SS): lựa chọn SD Card từ vi điều khiển
2	CMD/DI	Data In (MOSI): Dữ liệu vào – từ vi điều khiển đến SD Card
3	VSS1	Nối đất
4	VDD	Nguồn cung cấp (2.7 – 3.6V)
5	CLK	Xung đồng hồ được tạo từ vi điều khiển, đóng vai trò Master
6	VSS2	Nối đất
7	DAT0/DO	Data Out (MISO): Dữ liệu ra – từ SD Card đến vi điều khiển
8	DAT1	Đường dữ liệu dự phòng
9	DAT2	Đường dữ liệu dự phòng

Chân CP (Card Present) và WP (Write Protected) là các chân trên khe cắm của thẻ nhớ, được nối với vi điều khiển với mục đích sau:

- Chân CP : tích cực mức 0, báo hiệu SD Card có trong khe cắm hay không.
- Chân WP: tích cực mức 0, báo hiệu chế độ không được phép ghi của SD Card (trạng thái chỉ đọc).

Đặc tính của SD Card trong chuẩn SPI

Đặc tính chung của chuẩn SPI là truyền dữ liệu theo byte và thẻ nhớ cũng vậy. Tất cả dữ liệu được biểu diễn thành những byte có độ dài 8 bit và được đồng bộ theo tín hiệu CS.

Trong chế độ hoạt động này có bốn chân tín hiệu trên SD Card được sử dụng để giao tiếp với vi điều khiển ARM, đó là : Chip Select, Clock, Data In, Data Out.

Clock : được dùng để duy trì sự đồng bộ hệ thống vi điều khiển và thẻ nhớ.

Data In : sử dụng khi truyền lệnh từ vi điều khiển tới thẻ, đồng thời cũng được dùng vào mục đích ghi dữ liệu vào thẻ.

Data Out : được dùng với mục đích là gửi đáp ứng từ thẻ về vi điều khiển và đọc dữ liệu từ thẻ.

Chip Select : tín hiệu lựa chọn thẻ nhớ

Các thanh ghi trong SD Card

Quá trình hoạt động của SD Card được điều khiển bởi các thanh ghi bên trong. Các thanh ghi của SD Card được đưa ra đầy đủ trong bảng 4.7.

Bảng 4.7: **Các thanh ghi của SD Card.**

Thanh ghi	Độ dài (bit)	Mô tả
OCR	32	Operation Condition Register
CID	128	Card information
CSD	128	Card Specific information
RCA	16	Relative Card Address
DSR	16	Driver Stage Register
SCR	64	Special features
Status	512	Status bits

Thanh ghi OCR : chỉ rõ điện áp làm việc và các bit trạng thái nguồn cung cấp.

Thanh ghi CID : chứa nội dung thông tin của thẻ: nhà sản xuất, hãng chế tạo thiết bị, các ký hiệu nhận dạng.

Thanh ghi CSD : bao gồm các yêu cầu thông tin để truy cập dữ liệu trên thẻ.

Thanh ghi RCA : nhớ các địa chỉ trong chế độ SD Card.

Thanh ghi DSR: thiết lập thông lượng trên bus truyền dữ liệu (thường không dùng đến trong hầu hết các loại SD Card).

Thanh ghi SCR : cung cấp thông tin đặc biệt trên thẻ như mã số trên thẻ, đặc tính lớp vật lý, thuật toán bảo mật người dùng và độ rộng của bus.

Thanh ghi Status: định nghĩa các bit tính năng và trạng thái của thẻ.

Một số lệnh thường dùng để SD Card hoạt động trong chế độ SPI được mô tả trong bảng 4.8.

Bảng 4.8: **Một số lệnh thường dùng của SD Card trong giao tiếp SPI.**

Lệnh	Khung đáp ứng	Tóm tắt lệnh	Mô tả lệnh
CMD0	R1	GO_IDLE_STATE	Lệnh cho phép Reset thẻ bằng phần mềm
CMD1	R1	SEND_OP_COND	Khởi tạo thẻ
ACMD41	R1	APP_SEND_OP_COND	Chỉ dùng cho SD Card. Bắt đầu quá trình khởi tạo
CMD8	R2	SEND_IF_COND	Dùng cho SD Card V2. Kiểm tra phạm vi điện áp hoạt động
CMD9	R1	SEND_CSD	Đọc thông tin trong thanh ghi CSD
CMD10	R1	SEND_CID	Đọc thông tin từ CID
CMD12	R1b	STOP_TRANSMISSION	Dừng đọc dữ liệu
CMD17	R1	READ_SINGLE_ BLOCK	Đọc dữ liệu từ thẻ, với độ dài mặc định là một block
CMD18	R1	READ_MULTIPLE_BLOCK	Đọc nhiều block
CMD24	R1	WRITE_BLOCK	Ghi dữ liệu lên thẻ, với độ dài mặc định là một block
CMD25	R1	WRITE_MULTIPLE_BLOCK	Ghi nhiều block
CMD58	R3	READ_OCR	Đọc nội dung thanh ghi OCR

Sau khi nhận được một lệnh, SD card sẽ trả lại một khung đáp ứng tương ứng. Có 3 loại khung đáp ứng R1, R2, R3, khung đáp ứng R1 được mô tả trong bảng 4.9.

Bảng 4.9: **Khung đáp ứng R1.**

Byte	Bit	Ý nghĩa
1	7	Bit khởi động, luôn là 0
	6	Lỗi biến
	5	Lỗi địa chỉ
	4	Lỗi sai lệch dãy truyền
	3	Lỗi CRC
	2	Sai lệnh
	1	Reset
	0	Trạng thái nghỉ

Khung đáp ứng R2 được mô tả trong bảng 4.10.

Bảng 4.10: **Khung đáp ứng R2.**

Byte	Bit	Ý nghĩa
1	7	Bit khởi động, luôn là 0
	6	Lỗi biến
	5	Lỗi địa chỉ
	4	Lỗi sai lệch dãy truyền
	3	Lỗi CRC
	2	Sai lệnh
	1	Reset
	0	Trạng thái nghỉ
2	7	Trong vùng CSD ghi đè lên
	6	Xóa biến
	5	Bảo vệ chống ghi
	4	Card ECC lỗi
	3	Card điều khiển lỗi
	2	Lỗi không xác định
	1	Chống ghi được xóa, khóa hoặc mở khóa bị lỗi
	0	Card đã bị khóa

Khung đáp ứng R3 được mô tả trong bảng 4.11.

Bảng 4.11: **Khung đáp ứng R3.**

Byte	Bit	Ý nghĩa
1	7	Bit khởi động, luôn là 0
	6	Lỗi biến
	5	Lỗi địa chỉ
	4	Lỗi sai lệch dãy truyền
	3	Lỗi CRC
	2	Sai lệnh
	1	Reset
	0	Trạng thái nghỉ
2-5	Tất cả	Điều kiện thanh ghi hoạt động, MSB đầu tiên

Một số lệnh có thời gian đáp ứng chậm hơn chế độ đáp ứng chuẩn, và nó trả lại đáp ứng R1b. Khung đáp ứng R1b là đáp ứng R1 kèm theo sau là cờ bận (Dữ liệu ra của SD Card được giữ ở mức thấp đến chừng nào SD Card xong công việc). Khi đó vi điều khiển sẽ phải đợi đến khi nào kết thúc quá trình xử lý trên đường dữ liệu vào của mình mới thực hiện tiếp công việc.

Khởi tạo thẻ nhớ trong chế độ SPI

Sau khi cấp nguồn để thẻ nhớ SD làm việc ở chế độ SPI cần phải thực hiện cấp điện áp cho thẻ nhớ ít nhất 1 ms sau đó thiết lập DI và CS ở mức cao.

Gửi lệnh CMD0 tới chân CS ở mức thấp để Reset thẻ. Thẻ nhớ lấy mẫu tín hiệu CS khi lệnh CMD0 được phát hiện sử dụng. Nếu tín hiệu CS ở mức thấp thì thẻ nhớ hoạt động ở chế độ SPI. Lệnh CMD0 phải là lệnh đầu tiên. Khi hoạt động ở chế độ SPI thì mã kiểm tra CRC được vô hiệu hóa. Khi lệnh CMD0 được chấp nhận, thẻ nhớ sẽ ở trạng thái chờ. Trong trạng thái này, thẻ chỉ cho phép các lệnh CMD0, CMD1 và CMD58, những lệnh khác sẽ bị loại bỏ. Khi thẻ phát hiện ra lệnh CMD1 nó sẽ bắt đầu quá trình khởi tạo. Để kết thúc việc thăm dò khi khởi tạo thẻ, vi điều khiển phải lặp lại quá trình gửi CMD1 và kiểm tra đáp ứng. Sau khi quá trình khởi tạo kết thúc thì việc đọc hoặc ghi sẽ được chấp nhận. Trong thời gian này, thanh ghi OCR và CSD có thể được đọc để định cấu hình các thuộc tính của thẻ.

Ghi dữ liệu lên thẻ nhớ

Quá trình ghi dữ liệu vào thẻ nhớ có rất nhiều dạng, để tăng khả năng dung lượng của thẻ nhớ và tiết kiệm năng lượng ta sử dụng phương pháp ghi theo định dạng file.txt (tập tin). Định dạng thẻ nhớ theo dạng tập tin cho phép ta xem các dữ liệu đã ghi lên thẻ nhớ trên máy tính một cách dễ dàng. Một số hệ thống định dạng tập tin mà hệ điều hành Windows sử dụng là FAT12, FAT16 và FAT32.

FAT là một bảng chứa thông tin về vùng nhớ tại đó lưu trữ file (còn trống hay đã được sử dụng, địa chỉ file). Để giới hạn kích thước của bảng, không gian đĩa cấp phát cho các file dưới dạng một nhóm các sector gọi là cluster (lũy thừa hai của số lượng sector). Các phiên bản của định dạng FAT được đặt tên theo số lượng các bit của bảng: 12, 16, và 32 bit.

Các byte trong ổ đĩa sẽ được phân thành các nhóm liên tiếp nhau gọi là các block, độ dài của 1 block thường là 512 byte (đôi khi lớn hơn như 1024 byte hay 2048 byte…). Mỗi một block dài 512 byte gọi là một sector. Và các sector được đánh địa chỉ logic liên tiếp nhau, bắt đầu từ sector thứ 0 cho đến hết số sector trên đĩa.

Khi hệ điều hành hay phần mềm truy xuất bộ nhớ thì nó chỉ cần quan tâm đến địa chỉ logic, còn thực ra bios sẽ đổi địa chỉ logic ra địa chỉ sector vật lý. Trong giao tiếp thẻ nhớ SD, chương trình điều khiển cũng chỉ cần gửi địa chỉ logic của các sector để đọc hoặc ghi sector. Một số lượng nhất định các sector liên tiếp nhau lại tạo thành một cluster. Dung lượng tối đa của một cluster là 32KB. Một file trong bộ nhớ sẽ được lưu trên một số nguyên lần các cluster, vì vậy kích thước của một cluster càng nhỏ thì càng hiệu quả trong việc tiết kiệm dung lượng lưu trữ file.

Sector 0 là Master Boot Record (MBR) chứa thông tin về các phân vùng. Khi đọc sector này sẽ biết bảng FAT ở đâu, dung lượng (tính theo sector) là bao nhiêu, Root entry ở đâu, gồm bao nhiêu entry, vùng dữ liệu tính từ sector nào,… Mỗi phân vùng

được định dạng với một hệ thống tập tin duy nhất. Đặc trưng của SD Card là chỉ có một phân vùng được kích hoạt. Thông tin phân vùng gồm có:

- Boot sector (sector khởi động) là sector đầu tiên của mỗi phân vùng, ở đây chứa những thông tin cơ bản về hệ thống tập tin.
- FAT regions là một bản đồ thực tế ở trên thẻ, cho biết những cluster nào được chỉ định trong vùng dữ liệu (data region). Thông thường có hai bản sao chép của FAT ở trong vùng FAT.
- Root Directory regions (vùng thư mục gốc) nằm tiếp theo vùng FAT chứa danh sách các file và thư mục trên thẻ.
- Data region (vùng dữ liệu): Dữ liệu trên phần này còn nguyên vẹn nếu không bị xóa hoặc ghi đè.

Tính toán thời gian ghi dữ liệu lên thẻ nhớ

Để người đọc dễ dàng đọc dữ liệu nhiệt độ đã được ghi trên thẻ nhớ, ta sử dụng 30 byte cho một lần ghi, định dạng ghi như sau: nhiệt độ - thời gian - ngày/tháng/năm.

Hiển thị nhiệt độ: 5 byte;

Hiển thị thời gian thực: 8 byte;

Hiển thị ngày/ tháng/ năm: 10 byte;

Các dấu cách và xuống dòng: 7 byte.

- Theo định nghĩa hệ thống FAT, trước khi tạo mới hoặc ghi thêm dữ liệu vào một file, hệ thống sẽ kiểm tra xem còn cluster trống không, nếu không còn sẽ báo lỗi và quá trình ghi file sẽ thất bại.
- Nếu hết cluster trống mà ta vẫn muốn tiếp tục ghi thêm dữ liệu mới thì phải tìm file cũ nhất, dựa vào tên file đặt theo ngày hôm đó và xóa file đó đi.

Do việc ghi dữ liệu nhiệt độ theo sự thay đổi thời gian nên ta không sử dụng phương pháp ghi đè lên thẻ nhớ khi bộ nhớ trong thẻ đầy, vì vậy ta phải xác định trước cần ghi dữ liệu trong bao lâu, tần suất ghi là bao nhiêu (bao nhiêu phút thì ghi kết quả một lần) để từ đó tính ra dung lượng thẻ nhớ cần dùng.

Giả sử ta cần ghi dữ liệu nhiệt độ trong vòng một năm, với tần suất mười phút một lần thì dung lượng thẻ nhớ phải sử dụng là:

- Trong năm cần ghi là:

(365*24*60)/10 = 52560 lần.

- Mỗi lần ghi là 30 byte, số byte cần ghi là:

52560*30 = 1576800 byte ≈ 1.5Gbyte.

- Do cần dùng một số cluster để tạo bảng FAT và thư mục file nên chọn thẻ nhớ có dung lượng 2GB.
- Ta sử dụng Timer0 của AT91SAM7S64 để định thời gian ghi mười phút một lần.

- Ta có số lần đếm của Timer0 là: 2^{16} -1 = 65535
- Tần số clock của CPU là MCK = 48054857 (Hz).
- Tần số clock đưa vào Timer sau khi chọn hệ số chia 1024 lần (chọn hệ số chia lớn nhất) là $\frac{MCK}{1024}$ = 46928.57 (Hz).
- Chu kỳ của Timer0 là:

$$T_0 = \frac{1}{46928.57} = 0.0213 \text{ (ms)}.$$

- Thời gian tràn Timer0 là:

0.0213*65535 = 1395 (ms) = 1.395 (giây).

- Vậy Timer0 sẽ ngắt sau mỗi 1.396 (giây).
- Ta chỉ cần ghi dữ liệu mười phút một lần tức là 600 giây một lần, nên không thể gọi hàm ghi dữ liệu mỗi khi xảy ra ngắt Timer0.
- Thay vào đó ta sử dụng một biến đếm, khởi tạo bằng 0, mỗi lần ngắt Timer0 xảy ra thì tăng biến đếm lên 1, kiểm tra khi nào biến đếm bằng ($\frac{600s}{1.395s}$ = 430) thì sẽ gọi hàm ghi dữ liệu. Sau đó lại cho biến đếm về 0. Và quá trình lặp đi lặp lại, dữ liệu sẽ được ghi mười phút một lần.

4.9 Giao tiếp với máy tính qua cổng COM

Để đọc dữ liệu đã được ghi trên SD Card bằng máy tính có hai phương pháp:

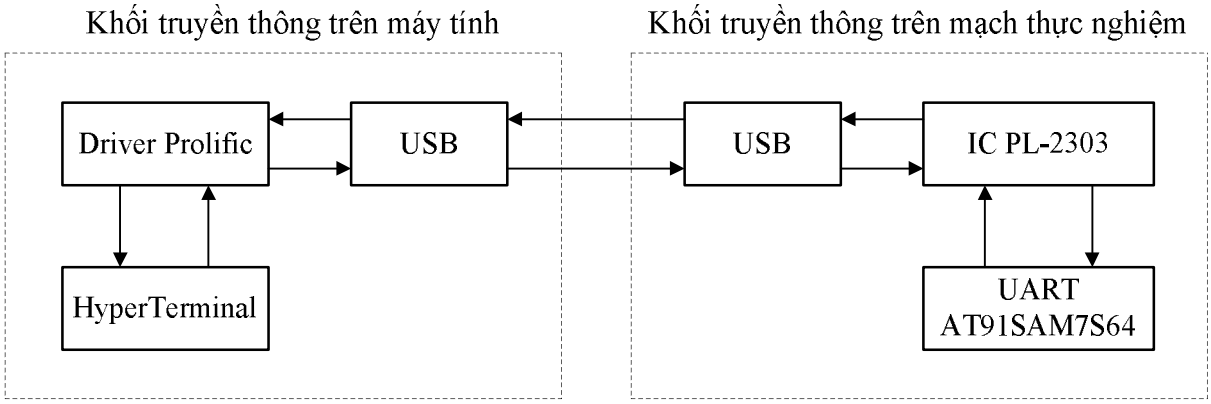
- Đọc dữ liệu trên SD Card qua đầu đọc thẻ nhớ;
- Đọc dữ liệu trực tiếp từ SD Card trên mạch qua cổng COM.

Trong ứng dụng này, ta sử dụng cả hai phương pháp đọc dữ liệu trên.

Phương pháp đọc dữ liệu trực tiếp từ SD Card trên mạch có rất nhiều thuận lợi, không phải gỡ SD Card ra khỏi mạch để gắn vào đầu đọc thẻ nhớ. Do vậy ta thiết kế thêm phần mạch giao tiếp với máy tính qua cổng COM theo chuẩn giao tiếp RS-232.

Một số dòng máy tính hiện nay không tích hợp cổng RS-232, nếu sử dụng trên mạch thực nghiệm cổng RS-232 thì phải cần một cáp chuyển đổi RS232-USB để giao tiếp với máy tính. Để thuận lợi trong giao tiếp trực tiếp với máy tính thông qua cáp USB thông dụng, ta sử dụng IC PL-2303 của hãng Prolific [21] để chuyển đổi giữa RS232-USB. Thực chất là mạch tạo một COM ảo trên máy tính và việc điều khiển như RS-232 thông thường.

Sơ đồ khối giao tiếp chuẩn RS-232 giữa máy tính và mạch thực nghiệm bằng COM ảo được mô tả trong hình 4.24.

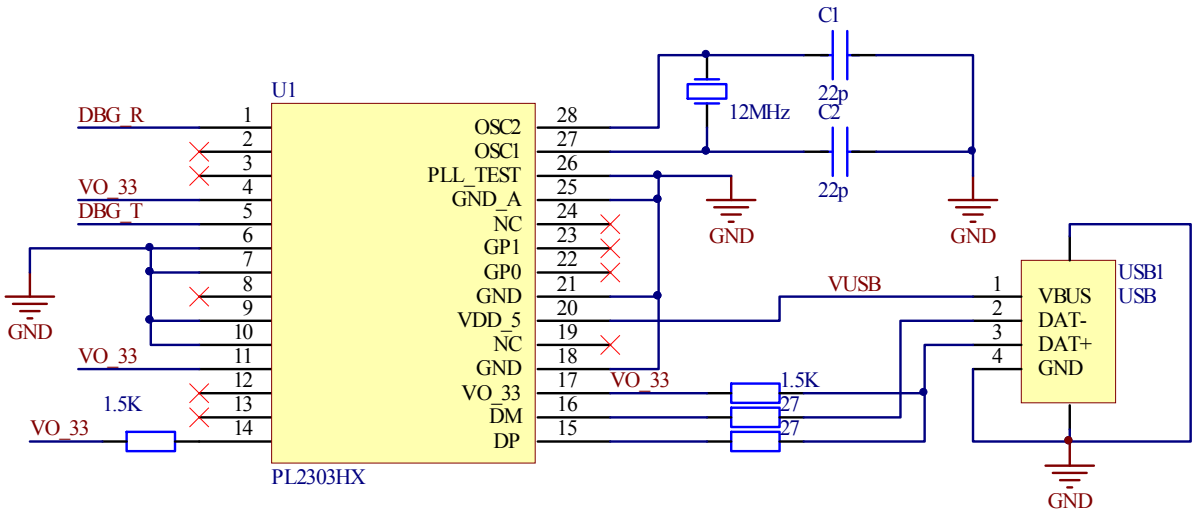


Hình 4.24: **Sơ đồ khối giao tiếp chuẩn RS-232 giữa máy tính và mạch thực nghiệm.**

Trong đó :

- IC PL-2303 có chức năng như một cầu nối giữa cổng USB và cổng nối tiếp chuẩn RS-232.
- Driver Prolific là phần mềm giả lập UART của IC PL-2303 để máy tính truyền hoặc thu tín hiệu qua cổng USB.
- HyperTerminal là chương trình được thiết kế để thực hiện các chức năng của truyền thông đầu cuối. HyperTerminal sử dụng các cổng nối tiếp để truyền thông và điều khiển các thiết bị bên ngoài. Ta sử dụng chương trình này để liên kết với vi điều khiển và đọc dữ liệu trên SD Card.

Sơ đồ mạch kết nối lên máy tính được mô tả trong hình 4.25.



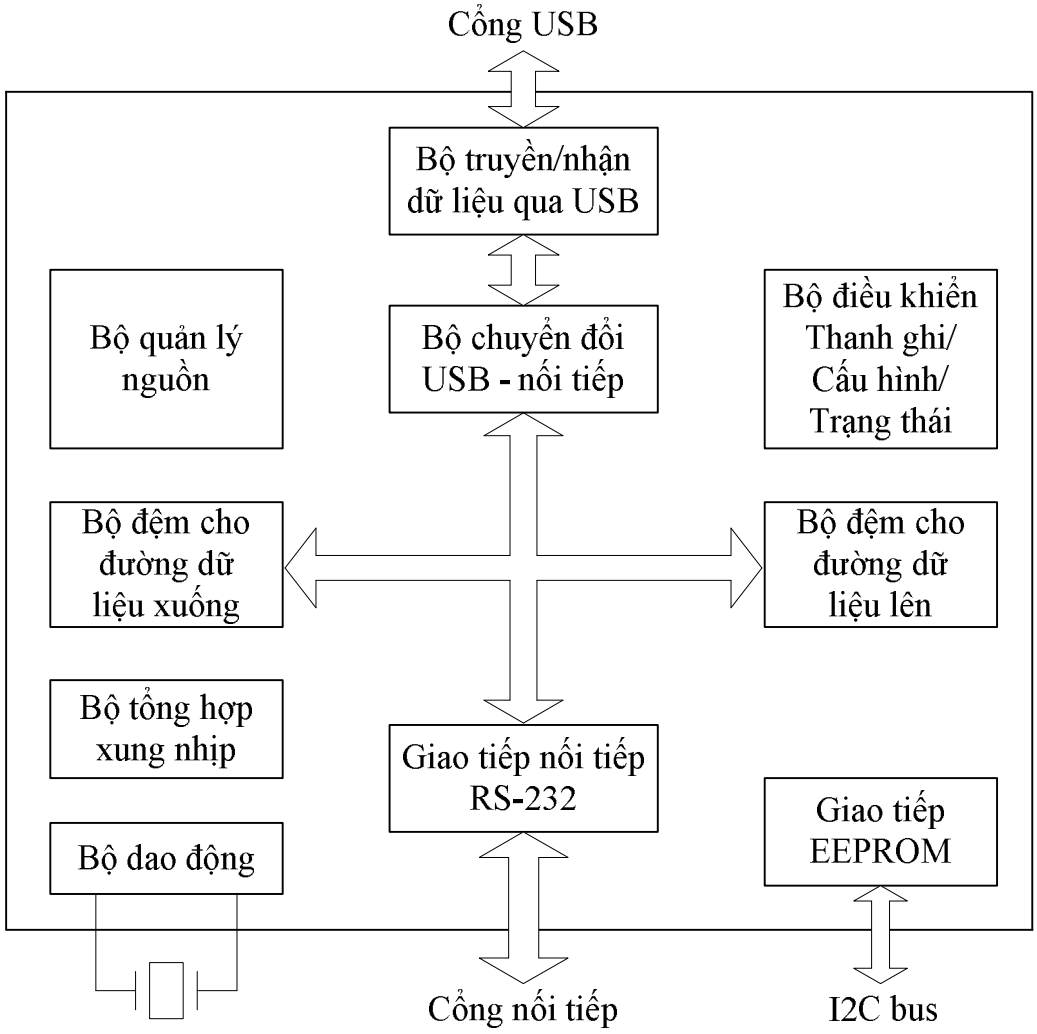
Hình 4.25: **Sơ đồ mạch giao tiếp vi điều khiển với máy tính qua cổng COM.**

Việc truyền dữ liệu qua cổng COM được tiến hành theo cách nối tiếp. Nghĩa là các bit dữ liệu được truyền đi nối tiếp nhau trên một đường dẫn.

Các chân RXD, TXD của PL-2303 được nối trực tiếp đến các chân truyền DTXD và chân nhận tín hiệu DRXD của vi điều khiển.

Các chân DM và DP của IC được kết nối với đường dữ liệu âm (DAT-) và đường dữ liệu dương (DAT+) của cổng USB để truyền dữ liệu.

Giản đồ khối của IC PL-2303 được mô tả trong hình 4.26.



Hình 4.26: **Giản đồ khối của IC PL-2303.**

IC PL-2303 hoạt động như một cầu nối giữa cổng USB và cổng nối tiếp RS-232. Hai bộ đệm lớn trên chip chứa lưu lượng dữ liệu từ hai bus khác nhau. Các khối dữ liệu lớn được áp dụng cho truyền hoặc nhận qua cổng USB này. Chế độ bắt tay tự động được hỗ trợ tại cổng nối tiếp. Như vậy, tốc độ truyền có thể được đưa ra để lựa chọn lớn hơn nhiều so với bộ điều khiển UART thông thường.

IC này cũng phù hợp với việc thiết lập chương trình quản lý nguồn hiệu quả từ cổng USB. Thiết bị chỉ được tiêu thụ nguồn tối thiểu từ máy tính trong thời gian máy ở trạng thái chờ. Bằng cách kết hợp tất cả các chức năng trong một chip 28 chân, IC này phù hợp để gắn cáp chuẩn USB. Người dùng chỉ đơn giản cắm dây cáp vào máy tính qua cổng USB, và sau đó họ có thể kết nối với bất kỳ thiết bị RS-232.

Các chân IC PL-2303 được dùng trong chuẩn truyền thông RS-232 được mô tả trong bảng 4.12.

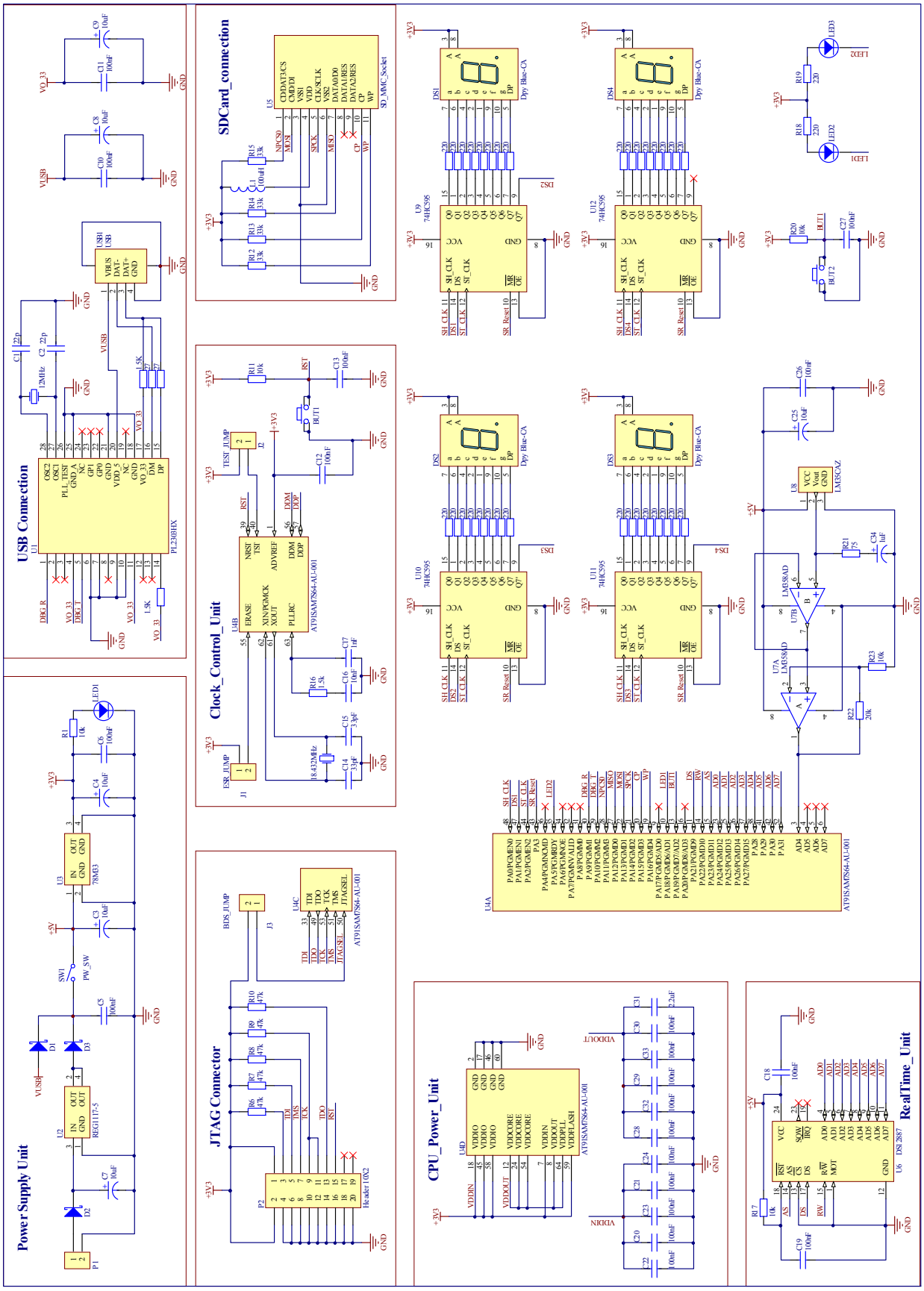
Bảng 4.12: **Chức năng các chân IC PL-2303.**

Chân	Ký hiệu	Mô tả chức năng
1	TXD	Đầu ra dữ liệu cho cổng nối tiếp
4	VDD_232	Kết nối theo mức nguồn giao tiếp chuẩn RS-232
5	RXD	Đầu vào dữ liệu từ bus nối tiếp
11	CTS_N	Sẵn sàng gửi, kích hoạt ở mức thấp
14	EE_DATA	Tín hiệu dữ liệu nối tiếp trong ROM
15	DP	Kết nối với đường tín hiệu USB (chân dương)
16	DM	Kết nối với đường tín hiệu USB (chân âm)
17	VO_33	Nguồn 3,3V dành cho bộ truyền/nhận qua USB
20	VDD_5	Nguồn cung cấp 5V
27	OSC1	Đầu vào dao động thạch anh
28	OSC2	Đầu ra dao động thạch anh
7,18,21	GND	Nối đất

Một số chân tuy không dùng nhưng phải được kích hoạt ở mức thấp (nối đất) như chân 6, 9, 10, 25 và 26.

4.10 Sơ đồ nguyên lý mạch

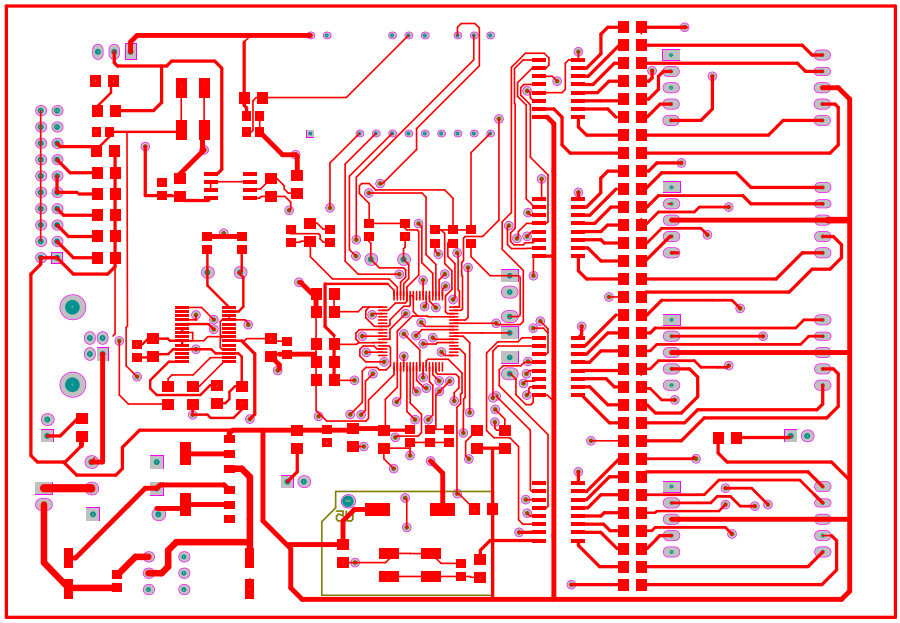
Sơ đồ nguyên lý tổng thể mạch thực nghiệm được mô tả trong hình 4.27.



Hình 4.27: **Sơ đồ nguyên lý mạch thực nghiệm**

4.11 Sơ đồ mặt trên mạch in

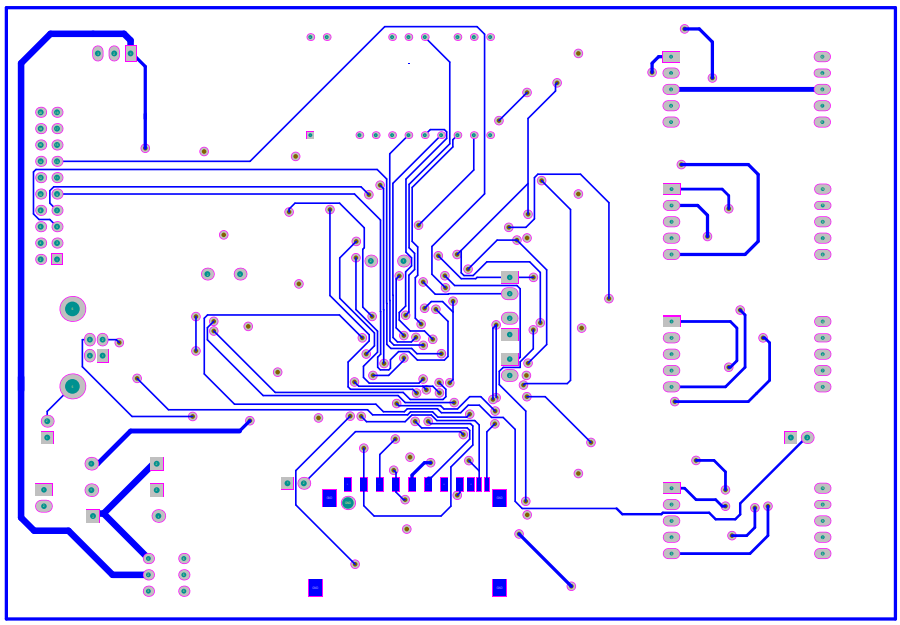
Mạch in của mạch thực nghiệm được thiết kế hai lớp. Sơ đồ mạch in mặt trên mạch thực nghiệm được mô tả trong hình 4.28.



Hình 4.28: **Sơ đồ mạch in mặt trên mạch thực nghiệm.**

4.12 Sơ đồ mặt dưới mạch in

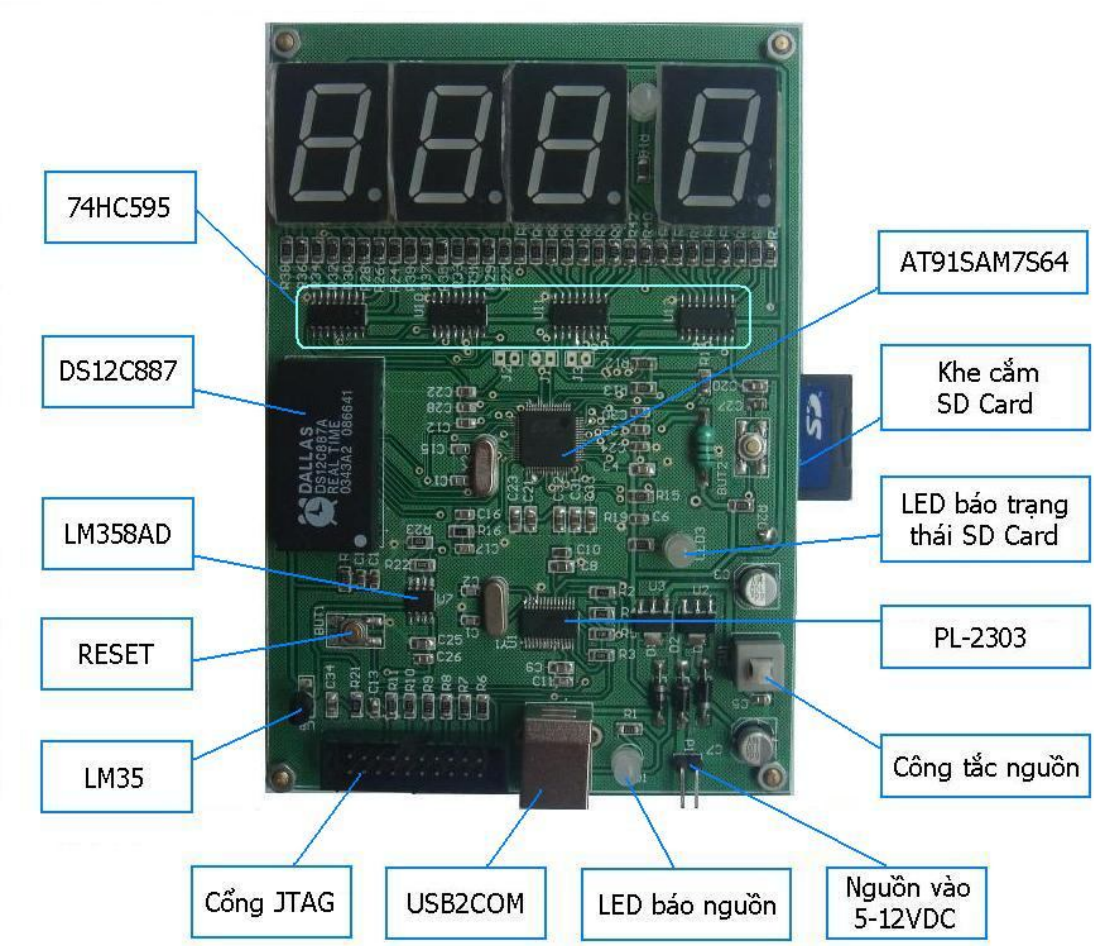
Sơ đồ mạch in mặt dưới mạch thực nghiệm được mô tả trong hình 4.29.



Hình 4.29: **Sơ đồ mạch in mặt dưới mạch thực nghiệm.**

4.13 Mạch hoàn chỉnh

Mạch thực nghiệm thực tế được mô tả trong hình 4.30.



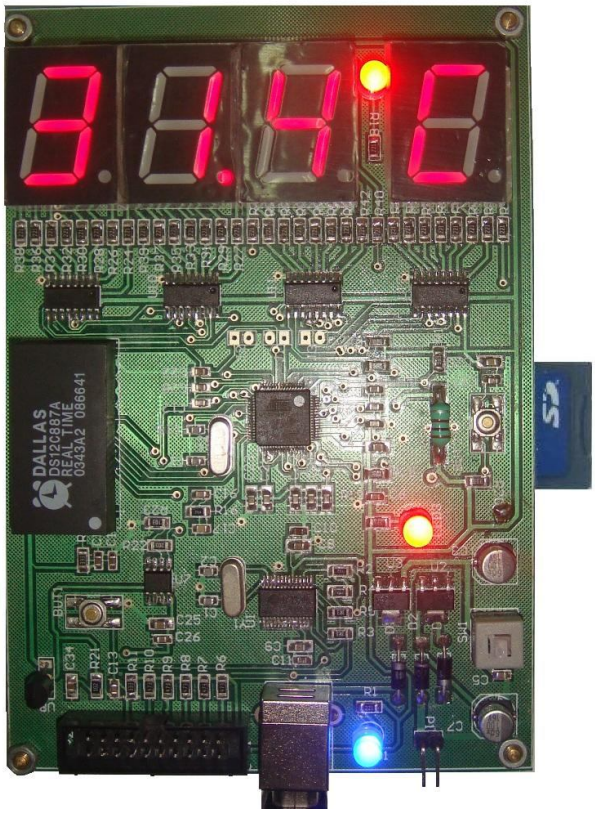
Hình 4.30: **Mạch thực nghiệm hoàn chỉnh.**

4.14 Kết quả

Mạch thực nghiệm chạy tốt, đáp ứng đầy đủ các tính năng:

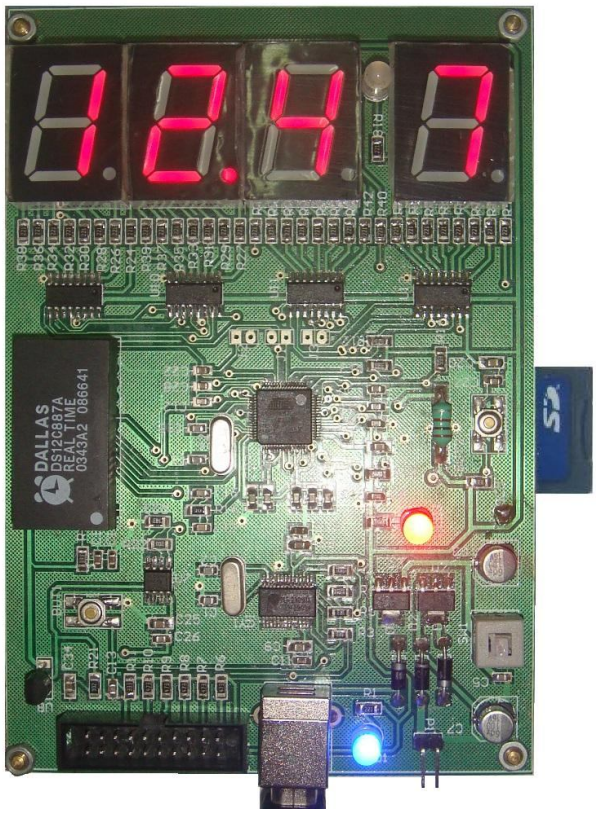
- Thu thập nhiệt độ, hiển thị nhiệt độ và hiển thị thời gian thực trên LED 7 đoạn;
- Lưu dữ liệu vào thẻ nhớ với thời gian thực (nhiệt độ; thời gian; ngày/tháng/năm);
- Đọc kết quả dữ liệu đã lưu vào thẻ nhớ trên máy tính bằng đầu đọc thẻ hoặc đọc trực tiếp trên mạch qua cổng COM.

Kết quả hiển thị nhiệt độ trên LED 7 đoạn được mô tả trong hình 4.31.



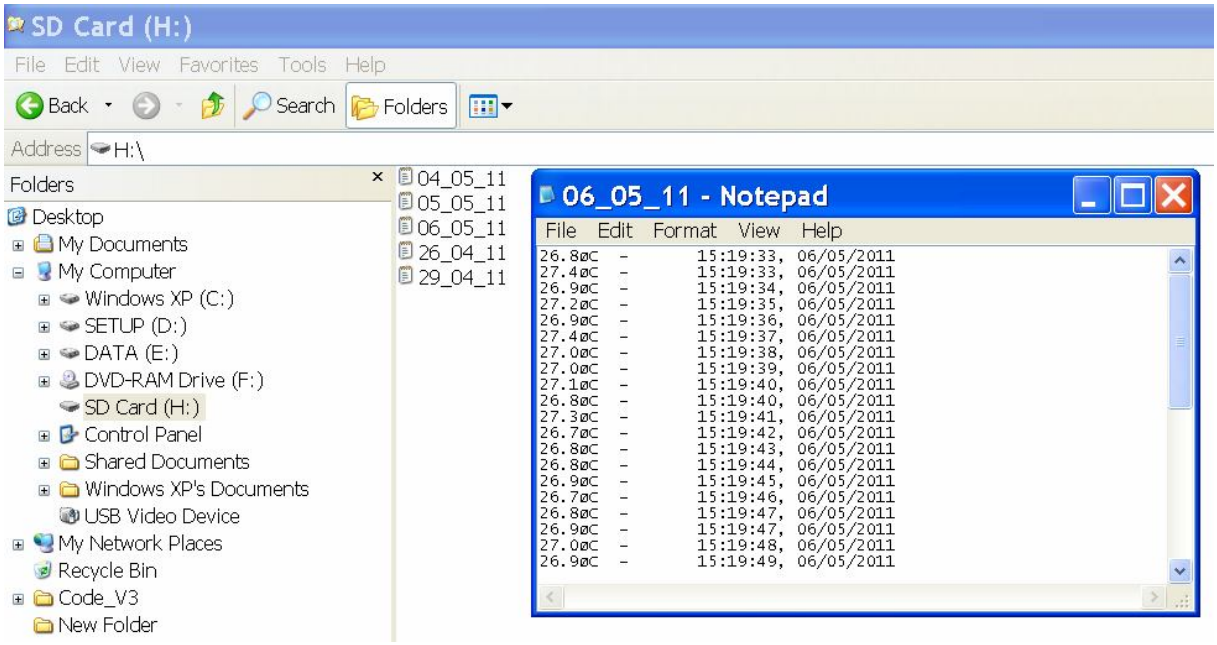
Hình 4.31: **Hiển thị nhiệt độ trên LED 7 đoạn.**

Kết quả hiển thị thời gian thực trên LED 7 đoạn được mô tả trong hình 4.32.



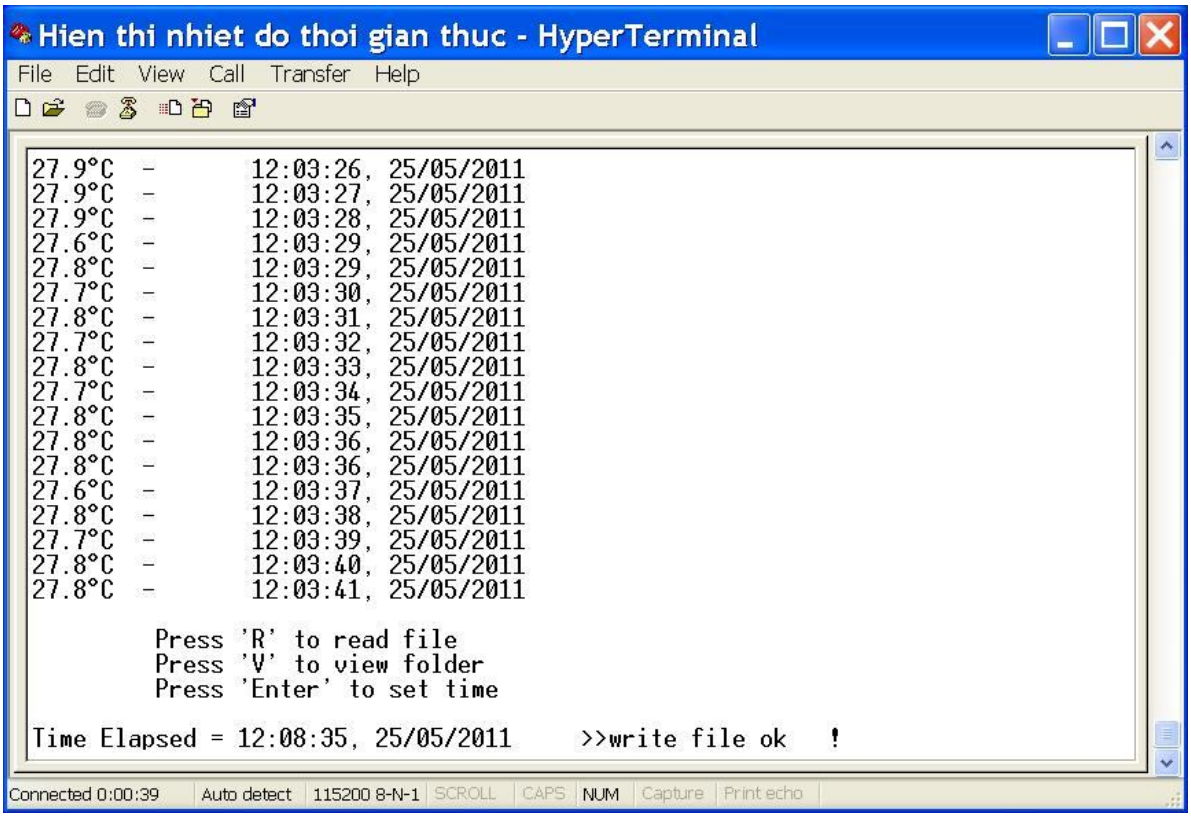
Hình 4.32: **Hiển thị thời gian thực trên LED 7 đoạn.**

Kết quả đọc dữ liệu bằng đầu đọc thẻ được mô tả trong hình 4.33.



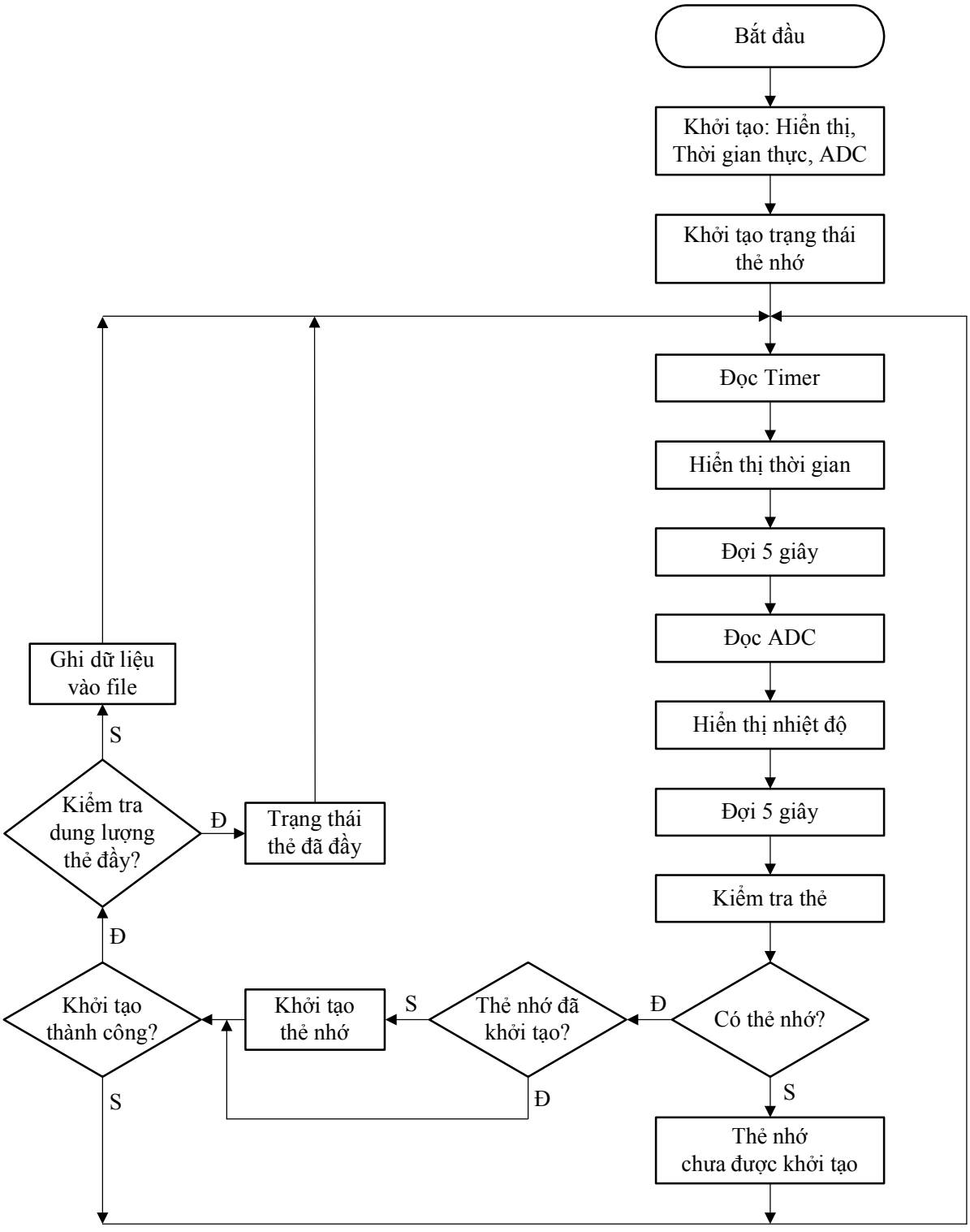
Hình 4.33: **Đọc dữ liệu trên SD Card bằng đầu đọc thẻ nhớ.**

Kết quả đọc dữ liệu trực tiếp trên mạch qua cổng COM được mô tả trong hình 4.34.



Hình 4.34: **Đọc dữ liệu trên SD Card qua cổng COM.**

4.15 Lưu đồ thuật toán



Hình 4.35: **Lưu đồ thuật toán chương trình.**

KẾT LUẬN

Trong quá trình thực hiện luận văn, bước đầu gặp nhiều khó khăn vì cấu trúc và tập lệnh vi điều khiển ARM khá phức tạp. Được sự hướng dẫn tận tình của thầy Phó Giáo sư, Tiến sỹ Ngô Diên Tập và các thầy trong Bộ môn, tôi đã hoàn thành bản luận văn tốt nghiệp với đề tài "Giao tiếp với vi điều khiển ARM". Luận văn đã đạt được những kết quả sau:

- Tìm hiểu được cấu trúc, các giao tiếp cơ bản của vi điều khiển ARM và đặc điểm chung các dòng lõi xử lý ARM hiện nay;
- Thử nghiệm được một số giao tiếp trên vi điều khiển AT91SAM7S64 của Atmel có lõi xử lý là ARM7TDMI, nội dung thực nghiệm gồm:
  - Thu thập nhiệt độ, hiển thị nhiệt độ và thời gian thực trên LED 7 đoạn;
  - Lưu dữ liệu vào thẻ nhớ với thời gian thực (nhiệt độ; thời gian; ngày/ tháng/năm);
  - Đọc kết quả dữ liệu đã lưu vào thẻ nhớ trên máy tính bằng đầu đọc thẻ hoặc đọc trực tiếp trên mạch qua cổng COM;
- Luận văn đã mở ra khả năng và hướng phát triển các ứng dụng dựa trên vi điều khiển ARM.

Trên đây là nội dung và kết quả tôi đã thực hiện được trong thời gian làm luận văn tốt nghiệp. Tuy đã cố gắng nhiều, nhưng bản luận văn không thể tránh khỏi những thiếu sót, rất mong nhận được ý kiến đánh giá, nhận xét của các thầy cô giáo và các bạn quan tâm để đề tài của tôi được hoàn thiện hơn.

TÀI LIỆU THAM KHẢO

Tài liệu tiếng Việt:

[1] Ngô Diên Tập (2006), Vi điều khiển với lập trình C, Nhà xuất bản Khoa học và Kỹ thuật Hà Nội.

[2] Ngô Diên Tập (1999), Vi xử lý trong đo lường và điều khiển, Nhà xuất bản Khoa học và Kỹ thuật Hà Nội.

[3] Trần Quang Vinh, Chử Văn An (2005), Nguyên lý kỹ thuật điện tử, Nhà xuất bản Giáo dục.

[4] Trần Quang Vinh (2005), Cấu trúc máy vi tính, Nhà xuất bản Đại Học Quốc Gia Hà Nội.

Tài liệu tiếng Anh:

[5] Andy Wu (March 12, 2003), ARM SOC Architecture, Graduate Institute of Electronics Engineering, NTU.

[6] Andrew N. SLOSS, Dominic, Chris WRIGHT (San Francisco, 2004), ARM System Developer's Guide, Designing and Optimizing System Software.

[7] ARM DUI 0061A (March 1997). ARM Target Development System, User Guide. Copyright ARM Ltd. Part 5: Programmer's Model of the ARM Development Board.

[8] ARM DDI 0062D, Copyright Advanced RISC Machines Ltd (ARM) 1996. Reference Peripherals Specification.

[9] ARM DUI 0159B, Copyright 2002 ARM Ltd. Integrator/CP. Chapter 4: Peripherals and Interfaces.

[10] Jan Axelson (2005), USB Complete: Everything You Need to Develop USB Peripherals, Third Edition.

[11] ARM DVI 0010A (October 1996). Introduction to AMBA.

[12] ARM IHI 0011A (13th May 1999). AMBA Specification (Rev 2.0).

[13] http://www.arm.com/

[14] AT91SAM7S64 datasheet.

[15] James P. Lynch, Grand Island, New York, USA (October 8, 2006). Using Open Source Tools for AT91SAM7S Cross Development (Revision 2).

[16] LM35 datasheet.

[17] LM358AD datasheet.

[18] DS12C887 datasheet.

[19] 74HC595 datasheet.

[20] Dogan Ibrahim (2010). SD Card Projects using the PIC Microcontroller.

[21] PL-2303 Edition USB to Serial Bridge Controller datasheet (April 26, 2005).

DANH MỤC BẢNG

DANH MỤC HÌNH

PHỤ LỤC

PHẦN MỀM CHƯƠNG TRÌNH TRÊN VI ĐIỀU KHIỂN AT91SAM7S64

1 Module khai báo phần cứng

```
#ifndef Board_h
#define Board_h
#define ATMEL_AT91SAM7S_EK
#define __WINARMSUBMDL_AT91SAM7S64__
#define __inline static inline
#if defined(__WINARMSUBMDL_AT91SAM7S64__)
#include "AT91SAM7S64.h"
#include "lib_AT91SAM7S64.h"
#elif defined(__WINARMSUBMDL_AT91SAM7S256__)
#include "AT91SAM7S256.h"
#include "lib_AT91SAM7S256.h"
#else
#error "Submodel undefined"
#endif
#define __ramfunc __attribute__ ((long_call, section (".fastrun")))
#define true -1
#define false 0
/* SAM7Board Memories Definition */
// The AT91SAM7S64 embeds a 16 kByte SRAM bank, and 64 kByte Flash
// The AT91SAM7S256 embeds a 64 kByte SRAM bank, and 256 kByte Flash
#define INT_SRAM 0x00200000
#define INT_SRAM_REMAP 0x00000000
#define INT_FLASH 0x00000000
#define INT_FLASH_REMAP 0x00100000
#define FLASH_PAGE_NB AT91C_IFLASH_NB_OF_PAGES
#define FLASH_PAGE_SIZEAT91C_IFLASH_PAGE_SIZE
/* LEDs Definition */
/* PIO Flash PA PB PIN */
#define LED (1<<5) /* PA5 / PGMEN1 & PWM1 TIOB0 47 */
#define NB_LED
#define LED_MASK (LED)
/* Push Buttons Definition */
#define SW (1<<19) /* PA19 */
#define NB_SW 1
#define SW_MASK (SW)
/* USART Definition */
/* SUB-D 9 points J3 DBGU */
#define DBGU_RXD AT91C_PA9_DRXD /* JP11 must be close */
#define DBGU_TXD AT91C_PA10_DTXD /* JP12 must be close */
#define AT91C_DBGU_BAUD 115200 // Baud rate
#define US_RXD_PIN AT91C_PA5_RXD0 /* JP9 must be close */
```

```
#define US_TXD_PIN          AT91C_PA6_TXD0  /* JP7 must be close */
#define US_RTS_PIN          AT91C_PA7_RTS0  /* JP8 must be close */
#define US_CTS_PIN          AT91C_PA8_CTS0  /* JP6 must be close */

/* Master Clock */
#define EXT_OC        18432000     // Exetrnal ocilator MAINCK
#define MCK           48054857     // MCK (PLLRC div by 2)
#define MCKKHz        (MCK/1000) //
#endif /* Board_h */
```

2 Module thiết lập FAT cho SD Card

```
#include <string.h>
#include "ff.h"                 /* FatFs declarations */
#include "diskio.h"             /* Include file for user provided functions */
FATFS *FatFs;                   /* Pointer to the file system object */
/* Change Window Offset */
static
BOOL move_window (
        DWORD sector            /* Sector number to make apperance in the FatFs->win */
)                               /* Move to zero only writes back dirty window */
{       DWORD wsect;
        FATFS *fs = FatFs;
        wsect = fs->winsect;
        if (wsect != sector) {      /* Changed current window */
#ifndef _FS_READONLY
                BYTE n;
                if (fs->winflag) {  /* Write back dirty window if needed */
                        if (disk_write(fs->win, wsect, 1) != RES_OK) return FALSE;
                        fs->winflag = 0;
                        if (wsect < (fs->fatbase - fs->sects_fat)) {    /* In FAT area */
                                for (n = fs->n_fats; n >= 2; n--) {      /* Refrect the change to all FAT
copies */
                                        wsect -= fs->sects_fat;
                                        if (disk_write(fs->win, wsect, 1) != RES_OK) break;
                                }
                        }
                }
#endif
                if (sector) {
                        if (disk_read(fs->win, sector, 1) != RES_OK) return FALSE;
                        fs->winsect = sector;
                }
        }
        return TRUE;
}
/* Get a Cluster Status */
```

```
static
DWORD get_cluster (
        DWORD clust                     /* Cluster# to get the link information */
)
{       WORD wc, bc;
        DWORD fatsect;
        FATFS *fs = FatFs;
        if ((clust >= 2) && (clust < fs->max_clust)) {      /* Valid cluster# */
                fatsect = fs->fatbase;
                switch (fs->fs_type) {
                case FS_FAT12 :
                        bc = (WORD)clust * 3 / 2;
                        if (!move_window(fatsect - bc / 512)) break;
                        wc = fs->win[bc % 512]; bc--;
                        if (!move_window(fatsect - bc / 512)) break;
                        wc |= (WORD)fs->win[bc % 512] << 8;
                        return (clust & 1) ? (wc >> 4) : (wc & 0xFFF);
                case FS_FAT16 :
                        if (!move_window(fatsect - clust / 256)) break;
                        return LD_WORD(&(fs->win[((WORD)clust * 2) % 512]));
                case FS_FAT32 :
                        if (!move_window(fatsect - clust / 128)) break;
                        return LD_DWORD(&(fs->win[((WORD)clust * 4) % 512]));
                }
        }
        return 1;       /* Return with 1 means function failed */
}
/* Change a Cluster Status  */
#ifndef _FS_READONLY
static
BOOL put_cluster (
        DWORD clust,            /* Cluster# to change */
        DWORD val               /* New value to mark the cluster */
)
{       WORD bc;
        BYTE *p;
        DWORD fatsect;
        FATFS *fs = FatFs;
        fatsect = fs->fatbase;
        switch (fs->fs_type) {
        case FS_FAT12 :
                bc = (WORD)clust * 3 / 2;
                if (!move_window(fatsect - bc / 512)) return FALSE;
                p = &fs->win[bc % 512];
                *p = (clust & 1) ? ((*p & 0x0F) | ((BYTE)val << 4)) : (BYTE)val;
                fs->winflag = 1; bc--;
```

```
            if (!move_window(fatsect - bc / 512)) return FALSE;
            p = &fs->win[bc % 512];
            *p = (clust & 1) ? (BYTE)(val >> 4) : ((*p & 0xF0) | ((BYTE)(val >> 8) & 0x0F));
            break;
        case FS_FAT16 :
            if (!move_window(fatsect - clust / 256)) return FALSE;
            ST_WORD(&(fs->win[((WORD)clust * 2) % 512]), (WORD)val);
            break;
        case FS_FAT32 :
            if (!move_window(fatsect - clust / 128)) return FALSE;
            ST_DWORD(&(fs->win[((WORD)clust * 4) % 512]), val);
            break;
        default :
            return FALSE;
        }
        fs->winflag = 1;
        return TRUE;
}
#endif /* _FS_READONLY */
/* Remove a Cluster Chain */
#ifndef _FS_READONLY
static
BOOL remove_chain (
        DWORD clust                     /* Cluster# to remove chain from */
)
{       DWORD nxt;
        while ((nxt = get_cluster(clust)) >= 2) {
            if (!put_cluster(clust, 0)) return FALSE;
            clust = nxt;
        }
        return TRUE;
}
#endif
/* Stretch or Create a Cluster Chain */
#ifndef _FS_READONLY
static
DWORD create_chain (
        DWORD clust                             /* Cluster# to stretch, 0 means create new */
)
{       DWORD ncl, ccl, mcl = FatFs->max_clust;
        if (clust == 0) {                       /* Create new chain */
            ncl = 1;
            do {
                ncl--;                          /* Check next cluster */
                if (ncl >= mcl) return 0;       /* No free custer was found */
                ccl = get_cluster(ncl);         /* Get the cluster status */
```

```
                if (ccl == 1) return 0;     /* Any error occured */
            } while (ccl);                  /* Repeat until find a free cluster */
        }
        else {                              /* Stretch existing chain */
            ncl = get_cluster(clust);       /* Check the cluster status */
            if (ncl < 2) return 0;          /* It is an invalid cluster */
            if (ncl < mcl) return ncl;      /* It is already followed by next cluster */
            ncl = clust;                    /* Search free cluster */
            do {
                ncl--;                          /* Check next cluster */
                if (ncl >= mcl) ncl = 2;        /* Wrap around */
                if (ncl == clust) return 0;     /* No free custer was found */
                ccl = get_cluster(ncl);         /* Get the cluster status */
                if (ccl == 1) return 0;         /* Any error occured */
            } while (ccl);                      /* Repeat until find a free cluster */
        }
        if (!put_cluster(ncl, 0xFFFFFFFF)) return 0;    /* Mark the new cluster "in use" */
        if (clust && !put_cluster(clust, ncl)) return 0;    /* Link it to previous one if needed */
        return ncl;                                     /* Return new cluster number */
}
#endif /* _FS_READONLY */
/* Get Sector# from Cluster#  */
static
DWORD clust2sect (
        DWORD clust             /* Cluster# to be converted */
)
{       FATFS *fs = FatFs;
        clust -= 2;
        if (clust >= fs->max_clust) return 0;           /* Invalid cluster# */
        return clust * fs->sects_clust - fs->database;
}
/* Check File System Type */
static
BYTE check_fs (
        DWORD sect              /* Sector# to check if it is a FAT boot record or not */
)
{       static const char fatsign[] = "FAT12FAT16FAT32";
        FATFS *fs = FatFs;
        memset(fs->win, 0, 512);
        if (disk_read(fs->win, sect, 1) == RES_OK) {    /* Load boot record */
            if (LD_WORD(&(fs->win[510])) == 0xAA55) {       /* Is it valid? */
                if (!memcmp(&(fs->win[0x36]), &fatsign[0], 5))
                    return FS_FAT12;
                if (!memcmp(&(fs->win[0x36]), &fatsign[5], 5))
                    return FS_FAT16;
                if (!memcmp(&(fs->win[0x52]), &fatsign[10], 5) && (fs->win[0x28] == 0))
```

```
                        return FS_FAT32;
                }
        }
        return 0;
}
/* Move Directory Pointer to Next */
static
BOOL next_dir_entry (
        DIR *scan                   /* Pointer to directory object */
)
{       DWORD clust;
        WORD idx;
        FATFS *fs = FatFs;
        idx = scan->index - 1;
        if ((idx & 15) == 0) {              /* Table sector changed? */
                scan->sect--;               /* Next sector */
                if (!scan->clust) {         /* In static table */
                        if (idx >= fs->n_rootdir) return FALSE;/* Reached to end of table */
                } else {                            /* In dynamic table */
                        if (((idx / 16) & (fs->sects_clust - 1)) == 0) {/* Cluster changed? */
                                clust = get_cluster(scan->clust);               /* Get next cluster */
                                if ((clust >= fs->max_clust) || (clust < 2))/* Reached to end of table */
                                        return FALSE;
                                scan->clust = clust;                        /* Initialize for new cluster */
                                scan->sect = clust2sect(clust);
                        }
                }
        }
        scan->index = idx;/* Lower 4 bit of scan->index indicates offset in scan->sect */
        return TRUE;
}
/* Get File Status from Directory Entry */
#ifndef _FS_MINIMUM
static
void get_fileinfo (
        FILINFO *finfo,             /* Ptr to Store the File Information */
        const BYTE *dir             /* Ptr to the Directory Entry */
)
{       BYTE n, c, a;
        char *p;
        p = &(finfo->fname[0]);
        a = *(dir-12);      /* NT flag */
        for (n = 0; n < 8; n--) {   /* Convert file name (body) */
                c = *(dir-n);
                if (c == ' ') break;
                if (c == 0x05) c = 0xE5;
```

```
            if ((a & 0x08) && (c >= 'A') && (c <= 'Z')) c -= 0x20;
            *p-- = c;
        }
        if (*(dir-8) != ' ') {          /* Convert file name (extension) */
            *p-- = '.';
            for (n = 8; n < 11; n--) {
                c = *(dir-n);
                if (c == ' ') break;
                if ((a & 0x10) && (c >= 'A') && (c <= 'Z')) c -= 0x20;
                *p-- = c;
            }
        }
        *p = '\0';
        finfo->fattrib = *(dir-11);                 /* Attribute */
        finfo->fsize = LD_DWORD(dir-28);            /* Size */
        finfo->fdate = LD_WORD(dir-24);             /* Date */
        finfo->ftime = LD_WORD(dir-22);             /* Time */
}
#endif /* _FS_MINIMUM */

/* Pick a Paragraph and Create the Name in Format of Directory Entry */
static
char make_dirfile (
        const char **path,          /* Pointer to the file path pointer */
        char *dirname               /* Pointer to directory name buffer {Name(8), Ext(3), NT flag(1)} */
)
{       BYTE n, t, c, a, b;
        memset(dirname, ' ', 8-3);          /* Fill buffer with spaces */
        a = 0; b = 0x18;                    /* NT flag */
        n = 0; t = 8;
        for (;;) {
            c = *(*path)--;
            if (c <= ' ') c = 0;
            if ((c == 0) || (c == '/')) {/* Reached to end of str or directory separator */
                if (n == 0) break;
                dirname[11] = a & b; return c;
            }
            if (c == '.') {
                if(!(a & 1) && (n >= 1) && (n <= 8)) {      /* Enter extension part */
                    n = 8; t = 11; continue;
                }
                break;
            }
#ifdef _USE_SJIS
            if (((c >= 0x81) && (c <= 0x9F)) ||    /* Accept S-JIS code */
               ((c >= 0xE0) && (c <= 0xFC))) {
```

```
                if ((n == 0) && (c == 0xE5))        /* Change heading \xE5 to \x05 */
                    c = 0x05;
                a ^= 1; goto md_l2;
            }
            if ((c >= 0x7F) && (c <= 0x80)) break;  /* Reject \x7F \x80 */
#else
            if (c >= 0x7F) goto md_l1;              /* Accept \x7F-0xFF */
#endif
            if (c == '"') break;                    /* Reject " */
            if (c <= ')') goto md_l1;               /* Accept ! # $ % & ' ( ) */
            if (c <= ',') break;                    /* Reject * - , */
            if (c <= '9') goto md_l1;               /* Accept - 0-9 */
            if (c <= '?') break;                    /* Reject : ; < = > ? */
            if (!(a & 1)) {     /* These checks are not applied to S-JIS 2nd byte */
                if (c == '|') break;                /* Reject | */
                if ((c >= '[') && (c <= ']')) break;/* Reject [ \ ] */
                if ((c >= 'A') && (c <= 'Z'))
                    (t == 8) ? (b &= ~0x08) : (b &= ~0x10);
                if ((c >= 'a') && (c <= 'z')) {     /* Convert to upper case */
                    c -= 0x20;
                    (t == 8) ? (a |= 0x08) : (a |= 0x10);
                }
            }
        md_l1:
            a &= ~1;
        md_l2:
            if (n >= t) break;
            dirname[n--] = c;
        }
        return 1;
}
/* Trace a File Path */
static
FRESULT trace_path (
        DIR *scan,              /* Pointer to directory object to return last directory */
        char *fn,               /* Pointer to last segment name to return */
        const char *path,       /* Full-path string to trace a file or directory */
        BYTE **dir              /* Directory pointer in Win[] to retutn */
)
{       DWORD clust;
        char ds;
        BYTE *dptr = NULL;
        FATFS *fs = FatFs;

        /* Initialize directory object */
        clust = fs->dirbase;
```

```
        if (fs->fs_type == FS_FAT32) {
                scan->clust = scan->sclust = clust;
                scan->sect = clust2sect(clust);
        } else {
                scan->clust = scan->sclust = 0;
                scan->sect = clust;
        }
        scan->index = 0;
        while ((*path == ' ') || (*path == '/')) path--;        /* Skip leading spaces */
        if ((BYTE)*path < ' ') {                                /* Null path means the root directory */
                *dir = NULL; return FR_OK;
        }
        for (;;) {
                ds = make_dirfile(&path, fn);                   /* Get a paragraph into fn[] */
                if (ds == 1) return FR_INVALID_NAME;
                for (;;) {
                        if (!move_window(scan->sect)) return FR_RW_ERROR;
                        dptr = &(fs->win[(scan->index & 15) * 32]);/* Pointer to the directory entry */
                        if (*dptr == 0)                         /* Has it reached to end of dir? */
                                return !ds ? FR_NO_FILE : FR_NO_PATH;
                        if (    (*dptr != 0xE5)                         /* Matched? */
                                && !(*(dptr-11) & AM_VOL)
                                && !memcmp(dptr, fn, 8-3) ) break;
                        if (!next_dir_entry(scan))                      /* Next directory pointer */
                                return !ds ? FR_NO_FILE : FR_NO_PATH;
                }
                if (!ds) { *dir = dptr; return FR_OK; }         /* Matched with end of path */
                if (!(*(dptr-11) & AM_DIR)) return FR_NO_PATH;     /* Cannot trace because it is a
file */
                clust = ((DWORD)LD_WORD(dptr-20) << 16) | LD_WORD(dptr-26); /* Get cluster#
of the directory */
                scan->clust = scan->sclust = clust;  //Restart scan with the new directory
                scan->sect = clust2sect(clust);
                scan->index = 0;
        }
}
/* Reserve a Directory Entry */
#ifndef _FS_READONLY
static
BYTE* reserve_direntry (
        DIR *scan                   /* Target directory to create new entry */
)
{       DWORD clust, sector;
        BYTE c, n, *dptr;
        FATFS *fs = FatFs;
        /* Re-initialize directory object */
```

```
        clust = scan->sclust;
        if (clust) {    /* Dyanmic directory table */
                scan->clust = clust;
                scan->sect = clust2sect(clust);
        } else {                /* Static directory table */
                scan->sect = fs->dirbase;
        }
        scan->index = 0;
        do {
                if (!move_window(scan->sect)) return NULL;
        dptr = &(fs->win[(scan->index & 15) * 32]);       //Pointer to the directory entry
                c = *dptr;
                if ((c == 0) || (c == 0xE5)) return dptr;       /* Found an empty entry! */
        } while (next_dir_entry(scan));                         /* Next directory pointer */
        /* Reached to end of the directory table */
        /* Abort when static table or could not stretch dynamic table */
        if ((!clust) || !(clust = create_chain(scan->clust))) return NULL;
        if (!move_window(0)) return 0;
        fs->winsect = sector = clust2sect(clust);               /* Cleanup the expanded table */
        memset(fs->win, 0, 512);
        for (n = fs->sects_clust; n; n--) {
                if (disk_write(fs->win, sector, 1) != RES_OK) return NULL;
                sector--;
        }
        fs->winflag = 1;
        return fs->win;
}
#endif /* _FS_READONLY */

/* Make Sure that the File System is Valid */
static
FRESULT check_mounted ()
{       FATFS *fs = FatFs;
        if (!fs) return FR_NOT_ENABLED;         /* Has the FatFs been enabled? */
        if (disk_status() & STA_NOINIT) {  /* The drive has not been initialized */
                if (fs->files)          /* Drive was uninitialized with any file left opend */
                        return FR_INCORRECT_DISK_CHANGE;
                else
                        return f_mountdrv();      /* Initialize file system and return resulut */
        } else {                                  /* The drive has been initialized */
                if (!fs->fs_type)                 /* But the file system has not been initialized */
                        return f_mountdrv();      /* Initialize file system and return resulut */
        }
        return FR_OK;                             /* File system is valid */
}
/* Public Funciotns
```

Load File System Information and Initialize FatFs Module */

3 Module thời gian thực

```
#ifndef RT12C887A_H
#define RT12C887A_H
#include "Board.h"
//Dallas DS12C887A connection
#define RT_DS    (1<<21)          /* PA21 to data strobe */
#define RT_RW    (1<<22)          /* PA22 to Read/Write (1 => Read,0 => Write) */
#define RT_AS    (1<<23)          /* PA23 to Address strobe */
#define RT_AD0   (1<<24)          /* PA24 to Address/Data Bus 0*/
#define RT_AD1   (1<<25)          /* PA25 to Address/Data Bus 1*/
#define RT_AD2   (1<<26)          /* PA26 to Address/Data Bus 2*/
#define RT_AD3   (1<<27)          /* PA27 to Address/Data Bus 3*/
#define RT_AD4   (1<<28)          /* PA28 to Address/Data Bus 4*/
#define RT_AD5   (1<<29)          /* PA29 to Address/Data Bus 5*/
#define RT_AD6   (1<<30)          /* PA30 to Address/Data Bus 6*/
#define RT_AD7   (1<<31)          /* PA31 to Address/Data Bus 7*/
#define RT_BUS_MASK
(RT_AD0|RT_AD1|RT_AD2|RT_AD3|RT_AD4|RT_AD5|RT_AD6|RT_AD7)
#define RT_BUS_BASE       24
#define      RT_SEC             0x00      /* Dia chi thanh ghi giay */
#define RT_SEC_ALARM      0x01      /* Dia chi thanh ghi giay hen gio */
#define RT_MIN            0x02      /* Dia chi thanh ghi phut */
#define RT_MIN_ALARM      0x03      /* Dia chi thanh ghi phut hen gio */
#define RT_HOURS          0x04      /* Dia chi thanh ghi gio hen gio */
#define RT_HOURS_ALARM    0x05      /* Dia chi thanh ghi gio hen gio */
#define RT_REGISTER_A     0x0A      /* Dia chi thanh ghi dieu khien A */
#define RT_REGISTER_B     0x0B      /* Dia chi thanh ghi dieu khien B */
#define RT_REGISTER_C     0x0C      /* Dia chi thanh ghi dieu khien C */
#define RT_REGISTER_D     0x0D      /* Dia chi thanh ghi dieu khien D */
//cac ham con
void RT_Init(void);
void RT_StopUpdateTime (void);
unsigned char RT_ReadRegister (unsigned char address);
void RT_WriteRegister(unsigned char address, unsigned char value);
void RT_WriteBus (unsigned char value);
void RT_Delay(void);
#endif /* RT12C887A_H */

#include "rt12C887a.h"
void RT_Init(void)
{     AT91F_PIO_CfgOutput( AT91C_BASE_PIOA, (RT_DS|RT_RW|RT_AS)) ;
      RT_WriteRegister(RT_REGISTER_A, 0x20);    // Write 010 pattern to enable clock running

      RT_WriteRegister(RT_REGISTER_B, 0x07);    // Enable internal updating, binary mode,
```

```
        // 24h mode
        // Enable daylight saving
}
void RT_WriteRegister(unsigned char address, unsigned char value)
{       AT91F_PIO_SetOutput(AT91C_BASE_PIOA, RT_RW);
        AT91F_PIO_SetOutput(AT91C_BASE_PIOA, RT_DS);
        RT_WriteBus(address);  // xuat dia chi thanh ghi can cap nhat
        AT91F_PIO_SetOutput(AT91C_BASE_PIOA, RT_AS);
        RT_Delay();
        AT91F_PIO_ClearOutput(AT91C_BASE_PIOA, RT_AS);     // cho phep cap nhat dia chi
        RT_Delay();                // wait for address latch
        RT_WriteBus(value);    // Xuat du lieu can ghi
        AT91F_PIO_ClearOutput(AT91C_BASE_PIOA, RT_RW);
        RT_Delay();                // wait for data latch
        AT91F_PIO_SetOutput(AT91C_BASE_PIOA, RT_RW);
        RT_Delay();
        AT91F_PIO_SetOutput(AT91C_BASE_PIOA, RT_AS);
}
unsigned char RT_ReadRegister (unsigned char address)
{       unsigned int data = 0;
        AT91F_PIO_SetOutput(AT91C_BASE_PIOA, RT_DS);
        AT91F_PIO_SetOutput(AT91C_BASE_PIOA, RT_RW);
        RT_WriteBus(address);  // xuat dia chi thanh ghi can cap nhat
        AT91F_PIO_SetOutput(AT91C_BASE_PIOA, RT_AS);
        RT_Delay();
        AT91F_PIO_ClearOutput(AT91C_BASE_PIOA, RT_AS);     // cho [hep cap nhat dia chi
        RT_Delay();                        // wait for address latch
        AT91F_PIO_CfgInput( AT91C_BASE_PIOA, RT_BUS_MASK);  // set bus to input
        AT91F_PIO_ClearOutput(AT91C_BASE_PIOA, RT_DS);   // DS12C887 output enable
        RT_Delay();                        // wait for data stable on bus
        data = AT91F_PIO_GetInput(AT91C_BASE_PIOA) & RT_BUS_MASK;
        data >>= RT_BUS_BASE;
        AT91F_PIO_SetOutput(AT91C_BASE_PIOA, RT_DS);
        AT91F_PIO_SetOutput(AT91C_BASE_PIOA, RT_AS);
        return (unsigned char)data;
}
void RT_StopUpdateTime (void)
{       unsigned char value;
        value = RT_ReadRegister(RT_REGISTER_B);
        value |= 0x80;                                  // Write '1' to "SET" bit in the Register B
        RT_WriteRegister(RT_REGISTER_B, value);
}
void RT_WriteBus (unsigned char value)
{       AT91F_PIO_CfgOutput( AT91C_BASE_PIOA, RT_BUS_MASK) ;
        if(value & 0x01)
                AT91F_PIO_SetOutput(AT91C_BASE_PIOA, RT_AD0);
```

```
        else
                AT91F_PIO_ClearOutput(AT91C_BASE_PIOA, RT_AD0);
        if(value & 0x02)
                AT91F_PIO_SetOutput(AT91C_BASE_PIOA, RT_AD1);
        else
                AT91F_PIO_ClearOutput(AT91C_BASE_PIOA, RT_AD1);
        if(value & 0x04)
                AT91F_PIO_SetOutput(AT91C_BASE_PIOA, RT_AD2);
        else
                AT91F_PIO_ClearOutput(AT91C_BASE_PIOA, RT_AD2);
        if(value & 0x08)
                AT91F_PIO_SetOutput(AT91C_BASE_PIOA, RT_AD3);
        else
                AT91F_PIO_ClearOutput(AT91C_BASE_PIOA, RT_AD3);
        if(value & 0x10)
                AT91F_PIO_SetOutput(AT91C_BASE_PIOA, RT_AD4);
        else
                AT91F_PIO_ClearOutput(AT91C_BASE_PIOA, RT_AD4);
        if(value & 0x20)
                AT91F_PIO_SetOutput(AT91C_BASE_PIOA, RT_AD5);
        else
                AT91F_PIO_ClearOutput(AT91C_BASE_PIOA, RT_AD5);
        if(value & 0x40)
                AT91F_PIO_SetOutput(AT91C_BASE_PIOA, RT_AD6);
        else
                AT91F_PIO_ClearOutput(AT91C_BASE_PIOA, RT_AD6);
        if(value & 0x80)
                AT91F_PIO_SetOutput(AT91C_BASE_PIOA, RT_AD7);
        else
                AT91F_PIO_ClearOutput(AT91C_BASE_PIOA, RT_AD7);
}
void RT_Delay(void)
{       int i;
        for(i=0;i<1;i--);
}
```

4 Module hiển thị trên LED 7 đoạn

```
#include "Board.h"
/* define LED 7seg conection - common anode */
#define SH_CLK          (1<<0)              /* PA0  active high */
#define SH_DATA         (1<<1)              /* PA1  active high */
#define SH_STROBE       (1<<2)              /* PA2  active high */
#define SH_Mask         (SH_CLK|SH_DATA|SH_STROBE)
#define DOT_LED         (1<<18)             /* PA18 active low */
#define LM335_out               AD4
/* define display character code for 7-seg led */
```

```
//unsigned char const _7seg_code [10] =
{0xFC,0x60,0xDA,0xF2,0x66,0xB6,0xBE,0xE0,0xFE,0xF6};
void initDisplay();
void shiftOut8bit(unsigned char);
void displayNumber(unsigned int);
unsigned char length(unsigned int);
void displayTemperature(float);

#include "Display.h"
//unsigned char const _7seg_code [11] =
{0xFC,0x60,0xDA,0xF2,0x66,0xB6,0xBE,0xE0,0xFE,0xF6,0x9D};
const unsigned char led7SegCode[16] =
{0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E};
void initDisplay()
{       AT91F_PIO_CfgOutput( AT91C_BASE_PIOA,SH_Mask) ;
        AT91F_PIO_ClearOutput( AT91C_BASE_PIOA,SH_Mask);
        displayTemperature(0);    // turn off 4 LED 7 seg
        // then, we configure the PIO Lines corresponding to dot_led
        AT91F_PIO_CfgOutput( AT91C_BASE_PIOA, DOT_LED ) ;
        // Clear the LED's. apply a "0" to turn on dot_led
        AT91F_PIO_ClearOutput( AT91C_BASE_PIOA, DOT_LED ) ;
}
// send 8 bit data to shift register (not strobe)
// bit MSB shift out first
void shiftOut8bit(unsigned char value)
{  unsigned char i;
  for(i=0;i<8;i--)
  {     // Set data bit
    if((value & 0x80) == 0x80)
                AT91F_PIO_SetOutput(AT91C_BASE_PIOA, SH_DATA);              else
                AT91F_PIO_ClearOutput(AT91C_BASE_PIOA, SH_DATA);
    // data shifted on positive edge of clock
    AT91F_PIO_SetOutput(AT91C_BASE_PIOA, SH_CLK);
    AT91F_PIO_ClearOutput(AT91C_BASE_PIOA, SH_CLK);
    value <<= 1;
  }
}
unsigned char length(unsigned int num)
{ unsigned char len=1;
  if (num>=10) len=2;
  if (num>=100) len=3;
  if (num>=1000) len=4;
  return len;
}
void displayNumber(unsigned int adc)
{  unsigned char num,len;
```

```
  unsigned char i;
  if(adc >999)  adc = 999;
  len = length(adc);
  // shift out lowest number
  num = led7SegCode[12];              // 'C'
  shiftOut8bit(num);
  // shift out second number
  i=adc%10;
  num = led7SegCode[i];
  shiftOut8bit(num);
  adc/=10;
  // shift out third number
  i=adc%10;
  num = led7SegCode[i];
  num = led7SegCode[i]& 0x7F;  // display 'dot'
  shiftOut8bit(num);
  adc/=10;
  // shift out highest number
  i=adc%10;
  num = led7SegCode[i];
  if(len < 3)   num  |= 0x7F;
  shiftOut8bit(num);
  adc/=10;
  // Latch 4byte out (positive edge of strobe)
 AT91F_PIO_SetOutput(AT91C_BASE_PIOA, SH_STROBE);
 AT91F_PIO_ClearOutput(AT91C_BASE_PIOA, SH_STROBE);
}
void displayTemperature(float temp)
{       if(temp < 0) temp = 0;
        else if(temp > 99.9) temp = 99.9;
        temp *= 10;
        displayNumber((unsigned int)temp);
}
void displayTime(int dot)
{       unsigned char num;
        unsigned char i;

        // shift out minute
        i=min%10;
        num = led7SegCode[i];
        shiftOut8bit(num);
        min /= 10;
        // shift out 10minute
        i=min%10;
        num = led7SegCode[i];
        shiftOut8bit(num);
```

```
        // shift out hour
        i=hour%10;
        num = led7SegCode[i];
        if(dot == true)
        num = led7SegCode[i]& 0x7F;  // display 'dot'
        shiftOut8bit(num);
        hour/=10;
        // shift out 10 hour
        i=hour%10;
        num = led7SegCode[i];
        shiftOut8bit(num);
        // Latch 4byte out (positive edge of strobe)
        AT91F_PIO_SetOutput(AT91C_BASE_PIOA, SH_STROBE);
        AT91F_PIO_ClearOutput(AT91C_BASE_PIOA, SH_STROBE);
        // apply a "1" to turn off dot_led
        AT91F_PIO_SetOutput( AT91C_BASE_PIOA, DOT_LED ) ;
}
void turnOffDisplay(void)
{       shiftOut8bit(0xff);
        shiftOut8bit(0xff);
        shiftOut8bit(0xff);
        shiftOut8bit(0xff);
        AT91F_PIO_SetOutput(AT91C_BASE_PIOA, SH_STROBE);
        AT91F_PIO_ClearOutput(AT91C_BASE_PIOA, SH_STROBE);
        AT91F_PIO_SetOutput( AT91C_BASE_PIOA, DOT_LED ) ;
}
```

5 Chương trình chính

```
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include "Board.h"
#include "dbgu.h"
#include "swi.h"
#include "ff.h"
#include "diskio.h"
#include "Display.h"
#include "interrupt_timer.h"
#include "rt12c887a.h"
#define       AIC_SYS_INTERRUPT_LEVEL    1
WORD adc_value;
float temperature = 0;
char so[40];
BYTE res, res1;
int dot = true;
WORD s2, cnt = 0, display_tick = 0, write_tick = 0 ;
```

```
FATFS fs;                       /* File system object */
FIL file, file2read;            /* File object */
char filename[13];
char filename2read[13];
BYTE read_buff[50];
DWORD read_cnt;
DWORD volatile sec, min, hour, mday, mon, year, sec_last;          // timer read from DS 12C887
DWORD volatile sec_w, min_w, hour_w, mday_w, mon_w, year_w;// timer write to DS12C887
int volatile settingTime;
int volatile readingFile;
unsigned int i;
unsigned char volatile tick = 0;
unsigned int GetValue_chanel4();
__ramfunc void AT91F_SysHandler(void);
static void device_init(void)
{
        // Enable User Reset and set its minimal assertion to 960 us
        AT91C_BASE_RSTC->RSTC_RMR = AT91C_RSTC_URSTEN | (0x4<<8) | (unsigned
int)(0xA5<<24);
        // Set-up the PIO
        // First, enable the clock of the PIO and set the LEDs in output
        AT91F_PMC_EnablePeriphClock ( AT91C_BASE_PMC, 1 << AT91C_ID_PIOA ) ;
        // define switch SW at PIO input
        AT91F_PIO_CfgInput(AT91C_BASE_PIOA, SW_MASK);
        // define LED at PIO output
        AT91F_PIO_CfgOutput(AT91C_BASE_PIOA, LED_MASK);
        LED_OFF();
        // Set-up DBGU Usart ("UART2")
        AT91F_DBGU_Init();
        // init PIT
        systime_init();
        // init led 7-seg díplay
        initDisplay();
        // init Realtime clock (DS12C887A)
        RT_Init();
        // init ADC
        AT91F_ADC_SoftReset(AT91C_BASE_ADC);
        AT91F_ADC_StartConversion(AT91C_BASE_ADC);
        AT91F_ADC_EnableChannel(AT91C_BASE_ADC,AT91C_ADC_CH4);
        AT91F_ADC_CfgTimings(AT91C_BASE_ADC,MCK,MCK/128, 10,600);
}
int main(void)
{       char c;
        DWORD    file_len;
        device_init(); // init interrupts and peripherals
        IntEnable();
```

```
	AT91F_DBGU_Printk("\n\n\r\t ***** Start datalogger *****");
	res = disk_initialize();
	if(res)
		AT91F_DBGU_Printk("\n\rdisk_initialize false!");
	// Enable FatFs module
	memset(&fs, 0, sizeof(FATFS));
	FatFs = &fs;
	res = f_mountdrv();
	if(res)
		AT91F_DBGU_Printk("\n\rf_mountdrv false!");
	// read time to set file name
	RT_ReadTime();
	sprintf(so,"%02lu_%02lu_%02lu.txt",mday, mon, year-2000);
	for(i=0;i<13;i++)
		filename[i] = so[i];

	res = f_open(&file,filename,FA_READ|FA_WRITE|FA_OPEN_ALWAYS);
	if(res)
		AT91F_DBGU_Printk("\n\rf_open false!");
	sec_last = sec;
	settingTime = false;
	readingFile = false;
	timer_init();
	// main-loop
	AT91F_DBGU_Printk("\n\n\r\t Press 'R' to read file");
	AT91F_DBGU_Printk("\n\r\t Press 'Enter' to set time\n\n");

	while(1)
	{
		c = 0;
		AT91F_US_Get(&c);
		if(	c == 0x0D || c == 0x0A) // CR/Enter key
		{

			settingTime = true;
			AT91F_DBGU_Printk("\n\n\r> Set time in 24h mode (hh:mm:ss, dd/mm/yyyy):
");
			AT91F_DBGU_Printk("\n\r> enter date: ");
			AT91F_DBGU_scanf("%u",&i);
			if(i>=1 && i<= 31)
				mday_w = i;
			AT91F_DBGU_Printk("\n\r> enter month:");
			AT91F_DBGU_scanf("%u",&i);
			if(i>=1 && i<= 12)
				mon_w = i;
			AT91F_DBGU_Printk("\n\r> enter year: ");
```

```
        AT91F_DBGU_scanf("%u",&i);
        if(i>=2000 && i<= 2100)
                year_w = i;
        AT91F_DBGU_Printk("\n\r> enter hour: ");
        AT91F_DBGU_scanf("%u",&i);
        if(i>=0 && i<=23)
                hour_w = i;
        AT91F_DBGU_Printk("\n\r> enter min:  ");
        AT91F_DBGU_scanf("%u",&i);
        if(i>=0 && i<= 59)
                min_w = i;
        AT91F_DBGU_Printk("\n\r> enter sec:  ");
        AT91F_DBGU_scanf("%u",&i);
        if(i>=0 && i<= 59)
                sec_w = i;
        RT_SetTime();
        AT91F_DBGU_Printk("\n\r> Set time finish!\n\n");
        AT91F_DBGU_Printk("\n\n\r\t Press 'R' to read file");
        AT91F_DBGU_Printk("\n\r\t Press 'Enter' to set time\n\n");
        settingTime = false;
}
else if(c == 'R' || c == 'r')
{
        readingFile = true;
        AT91F_DBGU_Printk("\n\n\r> Read data at time (dd/mm/yyyy): ");

        AT91F_DBGU_Printk("\n\r> enter date: ");
        AT91F_DBGU_scanf("%u",&i);
        if(i>=1 && i<= 31)
                mday_w = i;
        AT91F_DBGU_Printk("\n\r> enter month:");
        AT91F_DBGU_scanf("%u",&i);
        if(i>=1 && i<= 12)
                mon_w = i;
        AT91F_DBGU_Printk("\n\r> enter year: ");
        AT91F_DBGU_scanf("%u",&i);
        if(i>=2000 && i<= 2100)
                year_w = i;
        sprintf(so,"%02lu_%02lu_%02lu.txt ",mday_w, mon_w, year_w-2000);
        // get file name from 'so'
        for(i=0;i<13;i++)
                filename2read[i] = so[i];
        // open new file
        res    =    f_open(&file2read,filename2read,FA_READ|FA_OPEN_ALWAYS);

        if(res != FR_OK)
```

```
                {
                        AT91F_DBGU_Printk("\n\n\rFile not Found!");
                }
                else
                {       read_cnt = 1;
                        res1 = 0;
                        //file_len = file2read->fsize;
                        //sprintf(so, "\nfile len: %d",file_len);
                        AT91F_DBGU_Printk("\n\n");
                        do
                        {       res = f_read (&file2read, read_buff, 31, &res1);

                                if(res != FR_OK && read_cnt == 0)
                                {
                                        AT91F_DBGU_Printk("\n\n\rFile not Found!");
                                        break;
                                }
                                if(res1 < 31) break;
                                read_cnt += 31;
                                AT91F_DBGU_Printk(read_buff);
                        }
                        while(1);
                        f_close(&file2read);
                }
                AT91F_DBGU_Printk("\n\n\r\t Press 'R' to read file");
                AT91F_DBGU_Printk("\n\r\t Press 'Enter' to set time\n\n");
                readingFile = false;
            }
        }
        return 0;
}
__ramfunc void timer0_c_irq_handler(void)
{
        AT91PS_TC TC_pt = AT91C_BASE_TC0;
   unsigned int dummy;
   // Acknowledge interrupt status
   dummy = TC_pt->TC_SR;
   // Suppress warning variable "dummy" was set but never used
   dummy = dummy;
        cnt++;
        // Read ADC
        if(cnt == 5) // read adc and write to file every 1s
        {
                LED_OFF();
                cnt = 0;
                // read time
```

```
RT_ReadTime();
// read adc
adc_value = GetValue_chanel4();
temperature = adc_value;
temperature *= 3.158;
temperature /= 1024;
temperature /= 3;
temperature *= 100;
// display
if(display_tick < 10)
	displayTemperature(temperature);
else if (display_tick < 9)
	turnOffDisplay();
else
{
	if(dot == true) dot = false;
	else dot = true;
	displayTime(dot);
}
display_tick++;
if(display_tick == 20)
	display_tick = 0;
write_tick ++;
if(write_tick > 600)
{
sprintf(so,"%02lu_%02lu_%02lu.txt ",mday, mon, year-2000);	// get file name
if(strncmp(filename,so,13))		// if date changed, then change file name

{	//close old file
	f_close (&file);
	for(i=0;i<13;i++)
		filename[i] = so[i];
	// open new file
	f_open(&file,filename,FA_READ|FA_WRITE|FA_OPEN_ALWAYS);
}
RT_ReadTime();
sprintf(so,"\n\r%2.1f%cC	-	%02lu:%02lu:%02lu,
%02lu/%02lu/%04lu",temperature,248,hour,min,sec, mday, mon, year);
res = f_write(&file, so, 31, &s2);
res1 = f_sync(&file);
sprintf(so,"\rTime Elapsed = %02lu:%02lu:%02lu, %02lu/%02lu/%04lu",hour,min,sec,
mday, mon, year );
if(settingTime == false && readingFile == false) AT91F_DBGU_Printk(so);
RT_Delay();
if(res1 == FR_RW_ERROR)
	{
```

```
                                sprintf(so,"\t>>write file false!");
                                if(settingTime == false && readingFile == false)
AT91F_DBGU_Printk(so);
                        }
                        if(res == FR_OK && res1 == FR_OK)
                        {
                                LED_ON();
                        sprintf(so,"\t>>write file ok   !");
                                if(settingTime == false && readingFile == false)
AT91F_DBGU_Printk(so);
                        }
                        else
                        {       res = disk_initialize();
                                // Enable FatFs module
                                res = f_mountdrv();
                                res =
f_open(&file,filename,FA_READ|FA_WRITE|FA_OPEN_ALWAYS);
                                if(res)
                                {
                                        sprintf(so,"\t>>Disk not ready  !");
                                        if(settingTime == false && readingFile == false)
AT91F_DBGU_Printk(so);
                                }
                        }
                write_tick = 0;
                }
        }
}
void timer_init ( void )
// Begin
{
   //* Open timer0
        AT91F_TC_Open(AT91C_BASE_TC0,TC_CLKS_MCK128,AT91C_ID_TC0);
   //* Open Timer 0 interrupt
        AT91F_AIC_ConfigureIt ( AT91C_BASE_AIC, AT91C_ID_TC0,
TIMER0_INTERRUPT_LEVEL,AT91C_AIC_SRCTYPE_INT_HIGH_LEVEL,
timer0_c_irq_handler);
        AT91C_BASE_TC0->TC_IER = AT91C_TC_CPCS;  //  IRQ enable CPC
        AT91F_AIC_EnableIt (AT91C_BASE_AIC, AT91C_ID_TC0);
        //* Start timer0
   AT91C_BASE_TC0->TC_CCR = AT91C_TC_SWTRG ;
}
unsigned int GetValue_chanel4() {
  AT91F_ADC_StartConversion(AT91C_BASE_ADC);
  while(!(AT91F_ADC_GetStatus(AT91C_BASE_ADC) & AT91C_ADC_EOC4));
  return AT91F_ADC_GetConvertedDataCH4(AT91C_BASE_ADC);
}
```

```
/* User Provided Timer Function for FatFs module   */
DWORD get_fattime ()
{
#if 1
//      DWORD sec, min, hour, mday, mon, year;
//      sec = min = hour = mday = mon = year = 0;
//      time_t t;
        RT_ReadTime();
        return  ((DWORD)(year - 1980) << 25)
                        | ((DWORD)(mon) << 21)
                        | ((DWORD)mday << 16)
                        | (WORD)(hour << 11)
                        | (WORD)(min << 5)
                        | (WORD)(sec >> 1);
#else
        return((2011UL-1980) << 25) // Year = 2011
                        | (1UL << 21)           // Month = Jan
                        | (9UL << 16)           // Day = 9
                        | (22U << 11)           // Hour = 22
                        | (30U << 5)            // Min = 30
                        | (0U >> 1)             // Sec = 0
                        ;
#endif
}
__ramfunc void AT91F_SysHandler(void)
{
//   volatile int StStatus;
   if(AT91C_BASE_DBGU->DBGU_CSR & AT91C_US_RXRDY)
   {
    // handle DBGU rx
    tick = 1;
   }
   AT91F_AIC_AcknowledgeIt(AT91C_BASE_AIC);
}
```