

The Specification, Design, and Implementation of a

# Home Automation System

Javier Castro and James Psota

{javy,psota}@mit.edu

6.111: Introductory Digital Systems Laboratory Final Project

Massachusetts Institute of Technology

---

This paper presents the specification, design, and implementation of a home automation system that was designed and built as a final project for 6.111 at M.I.T. This system was designed to be flexible and generally programmable, extensible such that adding additional features is relatively simple, and modular and forward-compatible, so that new components can be added without redesigning the entire system. To achieve these goals, the system runs a user-defined program on a special-purpose processor, using real-world sensor inputs as operands. The sensors and other input mechanisms along with user-programmable event schedules allow the user to adjust and customize the home environment. Using sensors that measure temperature, light level, and infrared commands from a remote control, this system will create a comfortable and safe home atmosphere. Design decisions, implementation details, and testing procedures are thoroughly discussed, and the resulting functional system is described.

---

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Design and Implementation</b>	<b>4</b>
2.1	Input/Sensor Layer . . . . .	4
2.1.1	IR Module . . . . .	4
2.1.2	Clock Module . . . . .	9
2.1.3	Temperature Module . . . . .	9
2.1.4	Light Module . . . . .	13
2.2	Control Layer . . . . .	14
2.2.1	Major FSM . . . . .	15
2.2.2	Instruction Format . . . . .	16
2.2.3	Program Format . . . . .	17
2.2.4	Compiler . . . . .	18
2.2.5	Truth Engine Processor . . . . .	18
2.2.6	Status Monitor: Hardware . . . . .	19
2.2.7	Status Monitor: Software . . . . .	20
2.3	Actuator Layer . . . . .	21
<b>3</b>	<b>Testing</b>	<b>22</b>
3.1	IR Module . . . . .	22
3.2	Temperature Module . . . . .	22
3.3	Light Module . . . . .	23
3.4	Control Layer . . . . .	23
3.5	Actuator Layer . . . . .	23
3.6	Integration Testing . . . . .	24
<b>4</b>	<b>Conclusion</b>	<b>24</b>

## List of Figures

1	Three-layer System Architecture . . . . .	5
2	IR Transmitter/Receiver Waveforms . . . . .	5
3	Phillips RC-5 Encoding Protocol . . . . .	6
4	IR Module Block Diagram . . . . .	7
5	IR Module State Transition Diagram . . . . .	8
6	IR Module Simulation Waveform . . . . .	8
7	Clock Module Block Diagram . . . . .	9
8	Clock Module Waveform Simulation . . . . .	10
9	Temperature Module Block Diagram . . . . .	11
10	A/D Interface State Transition Diagram . . . . .	12
11	Sensor Module State Transition Diagram . . . . .	13
12	Temperature Module Simulation Waveform . . . . .	13
13	Light Module Simulation Waveform . . . . .	14
14	Simple Thermostat Program . . . . .	14
15	Top-Level Control Unit Block Diagram . . . . .	15
16	Control Unit Major FSM . . . . .	16

17	64-bit Instruction Format . . . . .	16
18	Compiler Interface . . . . .	18
19	Truth Engine Processor Block Diagram . . . . .	19
20	Status Monitor FSM . . . . .	20
21	Status Monitor Screen Shot . . . . .	21
22	Lab Kit Integration Testing Setup . . . . .	24

#### List of Tables

I	Opcode Map . . . . .	17
---	----------------------	----

## 1. INTRODUCTION

*[Written by James Psota]*

Homes of the 21st century will become more and more self-controlled and automated. Simple devices such as a timer to turn on one's coffee maker in the morning have been around for many years, but much more sophisticated mechanisms will soon be prevalent in homes around the world. Imagine walking into your home and being greeted at the door with lights illuminating your path without you ever having to touch a light switch, with your favorite music streaming through the speakers in whichever room you enter (because your home recognized that it was you and not some other household member), all while having the peace of mind knowing that your home automation system took care of activating your security system. Furthermore, such a system could allow the user to schedule events to occur at recurring intervals (*e.g.*, turn on sprinkler system at 4:30a.m. every Tuesday and Thursday).

This report describes an approximation of such a home automation system that was designed and built as a final project for 6.111 at M.I.T. This system was designed to be flexible and generally programmable, extensible such that adding additional features is relatively simple, and modular and forward-compatible, so that new components can be added without redesigning the entire system. To achieve these goals, the system runs a user-defined program on a special-purpose processor, using real-world sensor inputs as operands.

The rest of this paper is organized as follows. In Section 2, details of the design and implementation are thoroughly discussed. Section 3 discusses the testing strategy that was used to get the system up and running. Finally, Section 4 concludes the paper.

## 2. DESIGN AND IMPLEMENTATION

*[Written by James Psota]*

Section 1 motivated the importance of an extensible and flexible system. This section describes the resulting top-level design that reflects these goals.

The Home Automation System (HAS) has three logical levels, as seen in Figure 1. The top-most layer is the input/sensor layer, which can be from sensors, clock modules, remote controls, or any other kind of input. The control unit comprises the next layer. This layer reads the inputs and performs actions depending on the values of the inputs and the control program specified by the user. Finally, the control unit outputs commands to the actuator layer. The actuator layer is responsible for forming and sending commands to the real-world systems such as lights, HVAC systems, security systems, and music systems. The details and corresponding subcomponents of each of these layers are described in the remainder of this section.

### 2.1 Input/Sensor Layer

The input/sensor layer is responsible for acquiring data and formatting the data to a digital signal that the control layer understands. In our implementation, such a format is simply an 8-bit digital active-high signal. This section describes the design and implementation of each of the sensor inputs in detail.

#### 2.1.1 IR Module. *[Designed, Implemented, and Written by Javier Castro]*

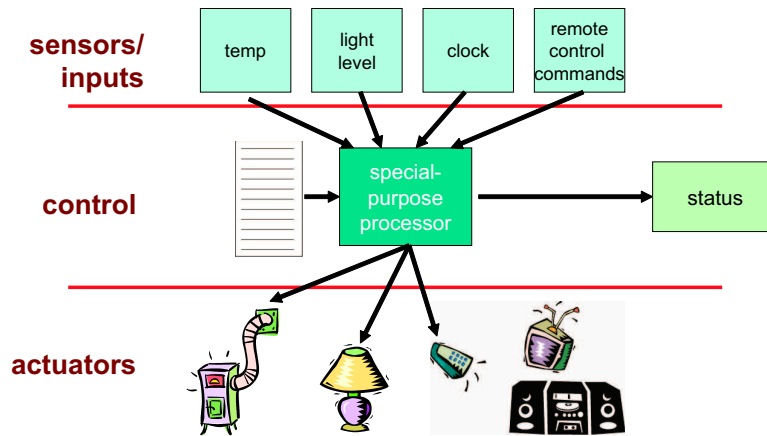


Fig. 1. Three-layer System Architecture

A remote control interface is provided as an input that can be used to interact with the HAS. In order to make this possible, the system interfaces with an infrared receiver which demodulates a 36kHz infrared signal to a digital format that can then be decoded and offered as input to the HAS control layer. The basics of infrared transmission are that when an infrared light is turned on and off at a frequency of 36kHz, the signal is interpreted to be in the on state (See Figure 2). When no light is flashing, the signal is in the off state. There are many different methods that can be used to encode a signal using infrared transmission. The Phillips RC-5 protocol was chosen for the HAS because it is well documented and has a fairly simple protocol.

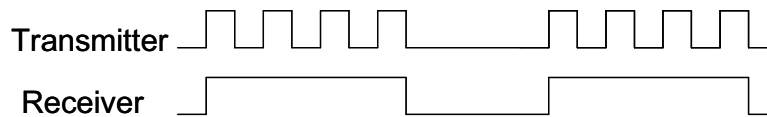


Fig. 2. IR Transmitter/Receiver Waveforms

Rather than using a variable bit-length encoding scheme, as is used by other protocols, the RC-5 protocol uses a form of Manchester Encoding to represent 1's and 0's. This means that in order to transmit a 1, the transmitter is silent for the first half of a bit, and blinking at 36kHz for the second half of the bit. A 0 is transmitted by a similar sequence: an on signal followed by an off signal.

Every message in the RC-5 protocol is 14 bits in length and is repeated every 114ms as long as a key is pressed. Each bit is 1.8ms in length meaning that each half bit is 0.9ms long. A message takes 25.2ms to transmit and is repeated every 114ms while a button is pressed. Repetition occurs in case the signal was not received correctly the first time. Each message takes the form that is shown in Figure 3. The first two bits are called the start bits and are always 1's. The next bit is called the check bit and is used to signal when a new button has been pressed. The check bit is in one state as long as a button is pressed so that an IR receiver can determine whether a message is new or a repetition of the previous

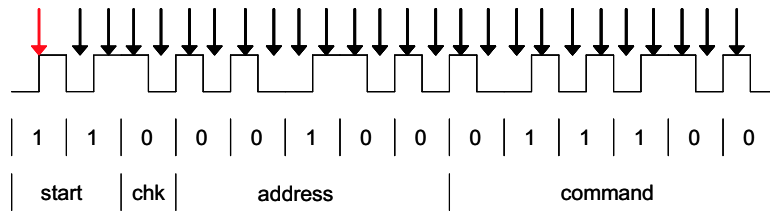


Fig. 3. Philips RC-5 Encoding Protocol

message. This allows users some room for error. For example, if a person holds down the “7” button on a remote control for longer than 114ms, the receiving system will understand that only a “7” was intended and that “77” is not an intended input. The next button press will toggle the state of the check bit. This change in bit is to be expected by the receiving system and indicates that a new button has been pressed. After the check bit, five address bits are transmitted. These bits are used to tell a system that the message is directed to it. For example, address 0 is reserved for TV1 and address 1 is reserved for TV2. This means that people can have two TV’s in the same room, and commands intended for one TV are not processed by the other. The last six bits of the message are the command bits. These bits specify which button has been pressed. A television, for example, could then turn its monitor on or change channels. The Home Automation System checks the address bits of a message only to ensure that a message is still being transmitted. This means that it does not matter what the address bits actually are, just that they abide by the RC-5 protocol in that the state of the first half of a bit is different from the state of the second half of the bit.

Responsible for receiving the infrared signal from a remote control is a TSOP2236 chip from Vishay electronics. The circuit is intended to detect a 36kHz infrared signal and interpret it as a valid 1. The TSOP2236 actually inverts the signal that is received so that a valid 1 corresponds to the low state and a valid 0 corresponds to the high state. Following the instructions found on the data sheet, a  $4.7\mu\text{F}$  capacitor was placed between the power and ground leads and a  $100\Omega$  resistor was placed between the power lead and Vdd. The purpose of the capacitor and resistor is to suppress power supply disturbances (Please refer to Figure 4).

The IR Decoding FSM is responsible for taking the demodulated signal from the IR receiver and parsing it for the command bits (See Figure 5). If the signal that is being received does not follow the RC-5 protocol, the FSM resets and waits for the next signal. A negative edge in the input signal asserts the start signal output of the FSM. Every positive clock edge signals the IR Decoding FSM to check whether the read signal which is provided by the timer is asserted. If the read signal is asserted, the module will check to see if it needs to change state; otherwise no state change will occur.

The FSM begins in the wait state. Since the read signal will only be asserted if the module has already received a negative edge as input from the IR receiver, the wait state checks to see that the second start bit is being received at this point. If the input signal is low, the FSM moves to the start state, otherwise the timer is reset and the FSM remains in the wait state. If in any of the states, the RC-5 protocol is broken, the FSM resets the timer and returns to the wait state. The start state checks that the input signal is high (second half of the second start bit) when the read signal is asserted and proceeds to the toggle1 state. In the toggle1 state, the FSM checks that the current message that is being received is not the

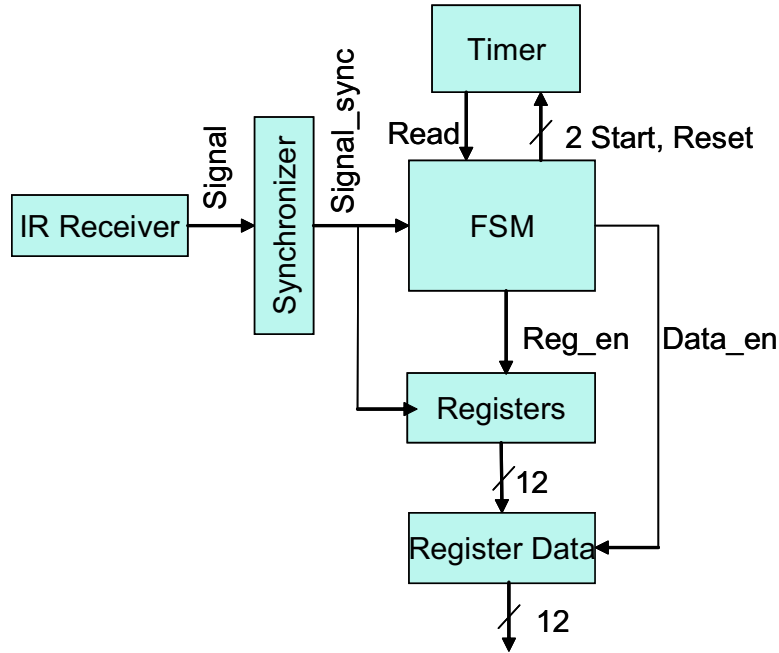


Fig. 4. IR Module Block Diagram

same as the previous message that was received. If the current message is a new message, the state of the input signal is saved to two registers. One is for reference on the next FSM iteration. The other register is the generic prev register where the state of the first half of each bit is stored. The FSM now proceeds to the toggle2 state. If the state of the input signal, at this point is different from the value held in the prev register, the FSM proceeds to the address1 state. On assertion of the read signal, the FSM will store the state of the input signal to the prev register, and move to the address2 state. While in the address2 state, if the state of the input signal is different from the value held in the prev register the FSM will assert the data\_en signal so that the address bit that is being provided as an input signal can be stored if desired. If five address bits have been received, the FSM now moves to the command1 state; otherwise it returns to the address1 state and repeats the process described above. The two command states (command1 and command2) behave the same way as the two address states, except that on the last command bit, the reg\_int signal is asserted. The reg\_int signal is used to enable a register which will hold the parsed command bits. The reg\_en signal gets the value of reg\_int one clock cycle later so that all of the command bits can be registered in the series of registers before being placed in a 6 bit register all together.

The decoder timer module has a start and reset signal as controls and a read signal as output. The timer module is a small state machine that waits for a start signal. Once the start signal is asserted, it proceeds to a delay state where it counts the length of a quarter bit (.45ms). Once this delay has occurred the timer moves to the count state where the timer counts .9ms and then asserts the read signal for the length of the period of the system clock (10MHz is expected). This sequence is repeated 26 times: the number of half bits in the 14

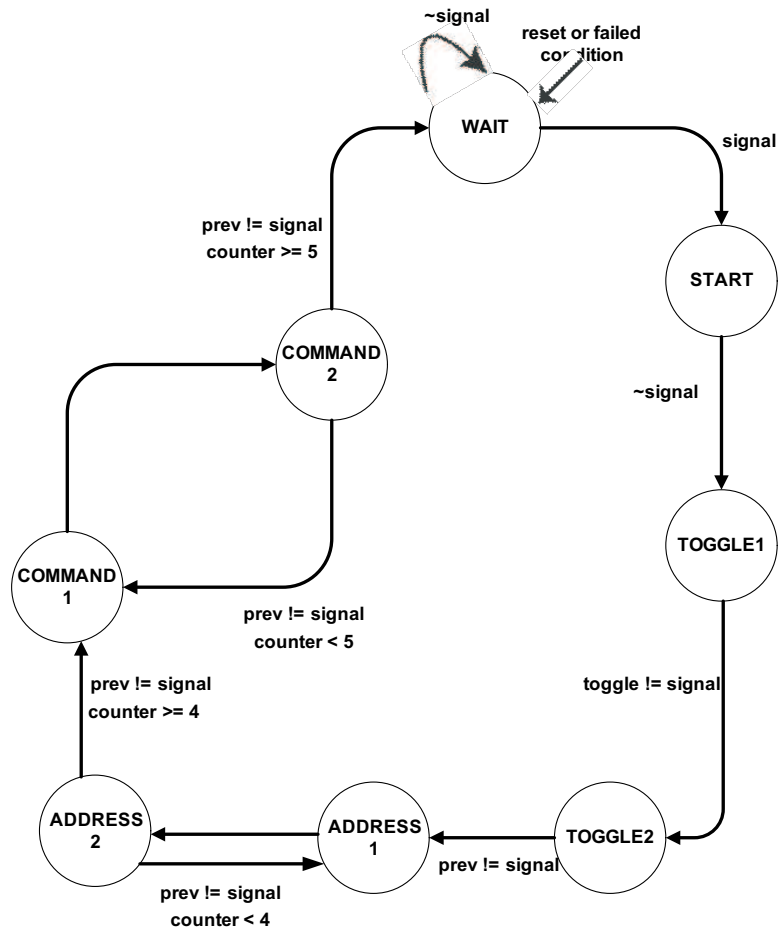


Fig. 5. IR Module State Transition Diagram

bit message when excluding the first bit. This is the number of times the IR Decoding FSM needs to check the state of the input signal. Whenever the IR Decoder Timer is reset, or completes the count state, it returns to the wait state, where it does nothing until it receives an asserted start signal. A waveform of the hardware simulation can be seen in Figure 6.



Fig. 6. IR Module Simulation Waveform



### 2.1.2 Clock Module. *[Designed, Implemented, and Written by James Psota]*

One of the goals of the HAS was to be able to schedule actions and events for certain times during the day or days of the week, for instance. Because of the HAS's flexible design, incorporating such functionality was simply a matter of adding another set of inputs that provided the current time and date.

The clock module keeps track of the following fields of the current date:

- seconds
- minutes
- hours
- day of week
- date
- month
- year

The system takes into account the number of days in each month, as well as leap years. The block diagram of the system is shown in Figure 7. Note the interdependence of each sub-unit on one another. There are some obvious unidirectional dependencies, such as that from the minute counter to the second counter. One of the dependencies, however, is bi-directional; the dependency between the date counter and the month counter is two-way, as the date counter must be aware of which month it is to determine when to rollover, and the month counter must know when to increment to the next month.

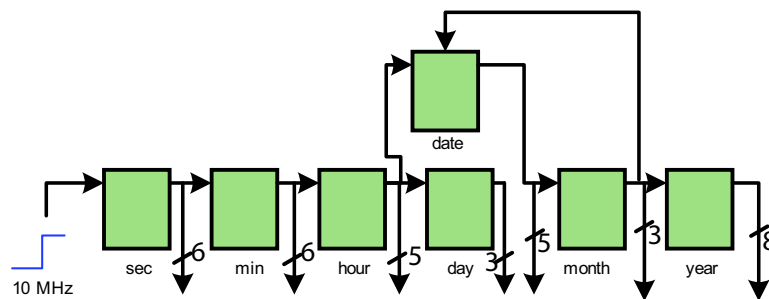


Fig. 7. Clock Module Block Diagram

Figure 8 shows a simulation of the clock module. Note that the date in the simulation is initialized at February 28, 2004 23:59:45, and that the date transitions to the 29th of February, as 2004 is a leap year.

### 2.1.3 Temperature Module. *[Designed, Implemented, and Written by Javier Castro]*

The Temperature Module is responsible for providing the control layer with a representation of the ambient temperature. The module takes a voltage, which represents the ambient temperature, as input and converts the information to a 2's complement binary number which represents the temperature in degrees Fahrenheit (See Figure 9). The fact that the temperature is initially represented as a voltage and then converted to a binary

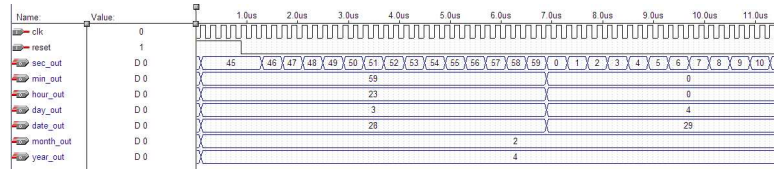


Fig. 8. Clock Module Waveform Simulation

number implies that there is an analog as well as a digital component to this module. The analog component provides a voltage to an AD670 analog to digital (A/D) converter. The AD670 is controlled by a finite state machine which is located within the FPGA. There is a Sensor FSM which controls the A/D Interface FSM, and is triggered by a sample timer. When the sample signal is asserted by the Timer module, the previously converted Fahrenheit temperature and the previously computed binary voltage value are registered and a new A/D conversion is started.

The analog part of the temperature module consists of a thermistor, 62k $\Omega$  resistor, and an LM741 Op Amp. A thermistor is a component that has a variable resistance which depends on the ambient temperature. It is made of D14.0 material and has a resistance of 100k $\Omega$  at 25°C. A voltage divider was created by placing the thermistor in series with the 62k $\Omega$  resistor and measuring the voltage across the non-variable resistor. The range of valid temperatures to 50-95°F, which in turn restricted the resistance of the thermistor to a range from 61k $\Omega$  to approximately 222k $\Omega$ . The reason for choosing a 62k $\Omega$  resistor for the voltage divider was so that the range of input voltages to the A/D converter would fall into the range of 0-2.55V. Due to a 10k $\Omega$  load resistance located in the AD670, the voltage from the voltage divider had to be passed through an operational amplifier which was arranged in a voltage follower configuration. This caused the voltage, which the AD670 was experiencing across its leads, to be the voltage that was expected from the calculations of the voltage divider.

The A/D Interface FSM allows the temperature module to obtain a binary representation of the temperature dependent voltage from an AD670 which is setup so that the output is formatted in straight binary representation and the input that is expected is from 0V to 2.55V. Please refer to Figure 10 for a state transition diagram. The A/D Interface has a go and status signal as inputs, and outputs a busy signal, a reg\_en signal, and three control signals (cs\_bar, ce\_bar, and r\_w\_bar). The go signal is connected to a sensor FSM and is asserted whenever an A/D conversion and reading should take place. This submodule has nine states: an idle state, four

read states, three conversion states, and a wait state. The idle state is the initial as well as reset state of the FSM. This state waits for the go signal to be asserted in order to proceed to the next state; otherwise nothing occurs in this state. There are four read states so that the setup and hold time requirements for the read cycle of the AD670 are met. The busy signal is asserted during the four read states, and the reg\_en signal is asserted during only the fourth read state. In order for the AD670 to begin reading the data, the cs\_bar and ce\_bar signals must be held low and the r\_w\_bar signal must be high for at least 250ns. As a result, there must be three read states, with the fourth allowing for pin stabilization and registering of the value which is being read. The reg\_en signal is used to enable a register which holds the binary representation of the newest voltage sample. The A/D

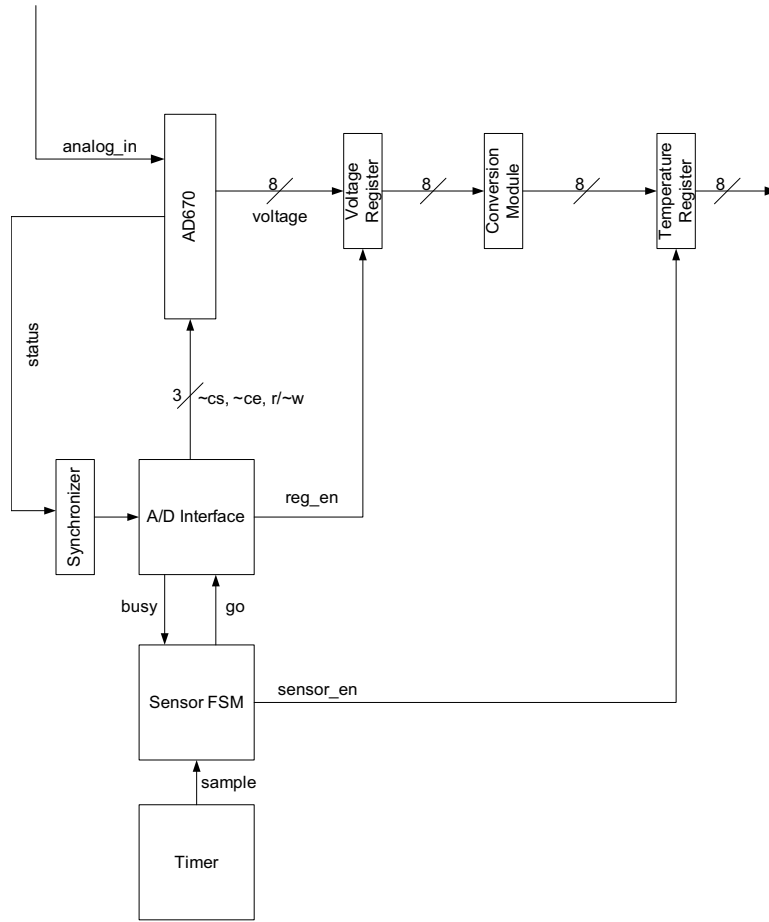


Fig. 9. Temperature Module Block Diagram

Interface FSM then continues to the three conversion states. Again, these states satisfy the setup time requirement for the write cycle of the AD670. The final wait state of this FSM is provided so that the **cs\_bar**, **ce\_bar**, and **r\_w\_bar** signals remain low until the AD670 actually begins a new write cycle. The wait state is waiting for the status signal from the AD670 to be asserted before returning the FSM to the idle state.

The Sensor FSM is responsible for registering the newest voltage to Fahrenheit conversion as well as starting a new A/D conversion whenever a sample is desired. The Sensor FSM has five states (See Figure 11). The idle state is where the Sensor FSM waits for the next sample time. Once the sample signal is asserted, the FSM proceeds to the register data state where it enables the sensor register so that the previous Temperature calculation is available for the control layer to read. The next three states concern A/D conversion and are provided so that the Sensor FSM will not return to the idle state until an A/D conversion has been initiated. The start reading state sends a start signal to the A/D Interface FSM. Next, the Sensor FSM waits for the A/D Interface to signal that it is no longer busy before

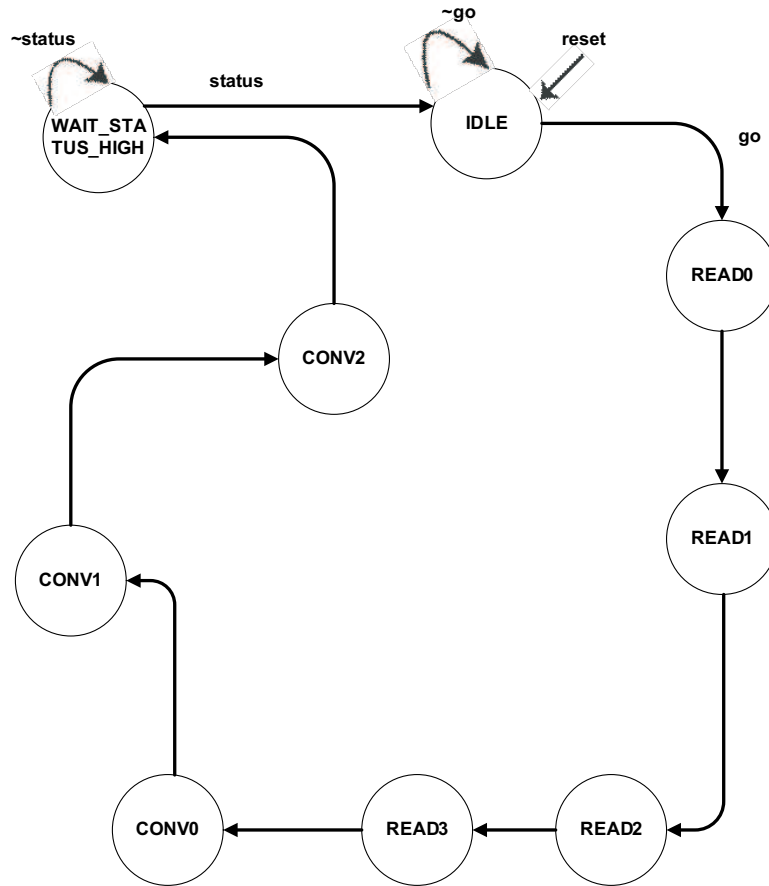


Fig. 10. A/D Interface State Transition Diagram

returning to the idle state.

The voltage which is sampled by the AD670 needs to be converted to a Fahrenheit value so that the control layer is able to process the data. Using matlab, the voltages which appear across the voltage divider at different temperatures from the range of 50-95°F were computed and converted to binary. Using this information, a look up table, implemented by a Verilog if tree, was generated within a clocked module so that whenever a voltage provided as input, the module would produce the corresponding temperature, in degrees Fahrenheit, as output. The reason for using a look up table, rather than some other method, was because the resistance of the thermistor, which was used in the analog component of the temperature module, is non-linear.

The timer block of the Temperature module is a clock divider that takes a 10MHz clock signal, and slows it down to an arbitrary length periodic signal. For the purposes of the HAS, the timer was set to assert its sample signal once every second. This is because temperature is not a rapidly changing environment variable and instead is fairly constant in the home environment. Figure 12 contains an image with a simulation waveform of this

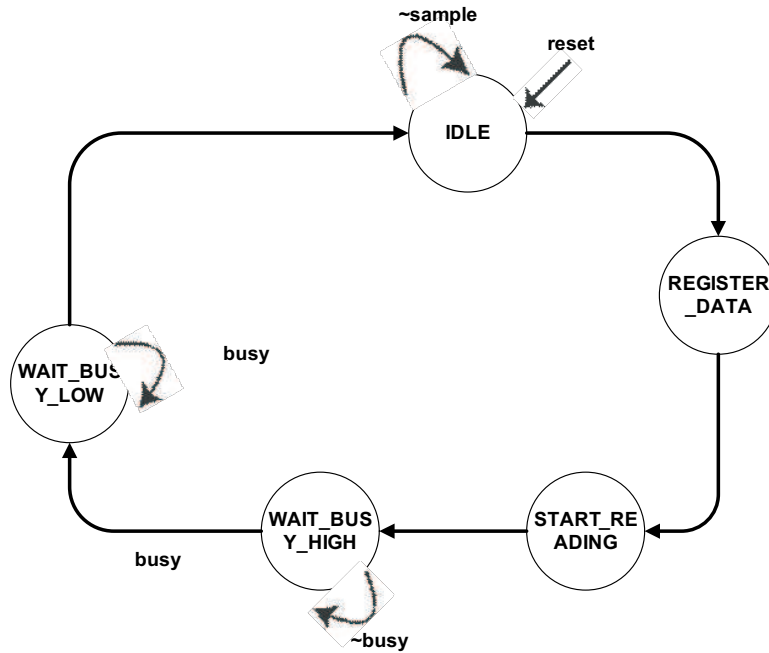


Fig. 11. Sensor Module State Transition Diagram

module.

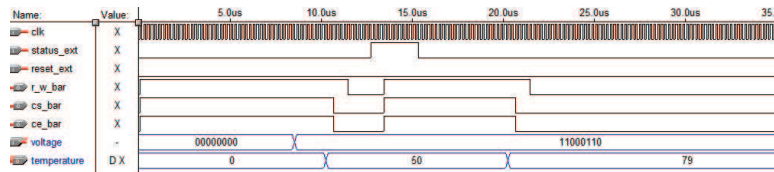


Fig. 12. Temperature Module Simulation Waveform

#### 2.1.4 Light Module. [Designed, Implemented, and Written by Javier Castro]

The Light Module was created by placing a PDV-P5001 photoresistor from Photonic Detectors Inc. in series with a  $2k\Omega$  resistor and measuring the voltage across the non-variable resistor. The result is a voltage divider similar to the one used in the Temperature Sensor. Once again, the voltage had to be passed through an LM741 Op Amp in order to avoid the voltage drop across the resistance load of the AD670. The two systems, Light Module and Temperature Module, were similar enough to use all of the same components with the exception of a substituted conversion module. The reason for choosing a  $2k\Omega$  resistor as opposed to any other resistance was so that the voltage measured by the AD670 would fall into the range of 0-2.55V. At ambient light levels, the voltage across the non-variable resistor is approximately 2.46V.

Since light level is measured in lumens, it seemed to make more sense to have a scale from 0-9 to represent different levels of light. This is because, unlike temperature, people do not have a general idea for how many lumens a particular level of light may be. With this in mind, the Light Module takes a voltage that ranges from 0-2.55V that is represented in straight binary returns which decade the voltage falls into with 9 corresponding to 2.46V and higher, and 0 corresponding to 0.36V and lower. This module was implemented using a series of if statements. Figure 13 depicts a simulation waveform of this module. For associated A/D interface and state transition diagrams, please refer to Figure 10 and 11. The block diagram is similar to that depicted in Figure 9, but has a different conversion module.

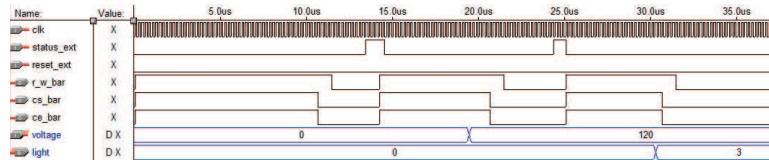


Fig. 13. Light Module Simulation Waveform

## 2.2 Control Layer

*[Designed, Implemented, and Written by James Psota]*

Before diving into the lower-level details of the control system, here is a primitive example of how the HAS would be used as a thermostat. Assume the system has an input that specifies the current temperature in the room that is to be climate-controlled, and has an actuator output that is connected to a heating unit. Also assume that the user desires to turn the heat on when the temperature falls below 66°F, and turn the heater back off when the temperature raises above 68°F. The user would then have to write a program to specify these actions, with one instruction to turn the heater on when the “on” condition is true, and one instruction to turn the program off when the “off” condition is true. The specific instruction format is described in section 2.2.2, but a textual description of each of the instructions as seen in Figure 14 is illustrative. For this example, assume the temperature sensor plugged into sensor input slot 7, and the heater is plugged into actuator output slot 3.

**Instruction-I** If sensor 7 < 66, set actuator 3 to 1

**Instruction-II** If sensor 7 > 68, set actuator 3 to 0

Fig. 14. Simple Thermostat Program

This program, when run by the control unit, performs the user’s desired action.

The top-level block diagram of the control unit can be seen in Figure 15. The cu\_control module is the major FSM that prompts the other components to stop and go. The truth\_engine block is the main processing unit. This block reads instructions from the instruction memory (implemented in an internal RAM) and executes it using sensor inputs

as operands. Finally, the status monitor provides the user with a real-time view of the system's inputs and outputs via the RS232 port of a host PC.

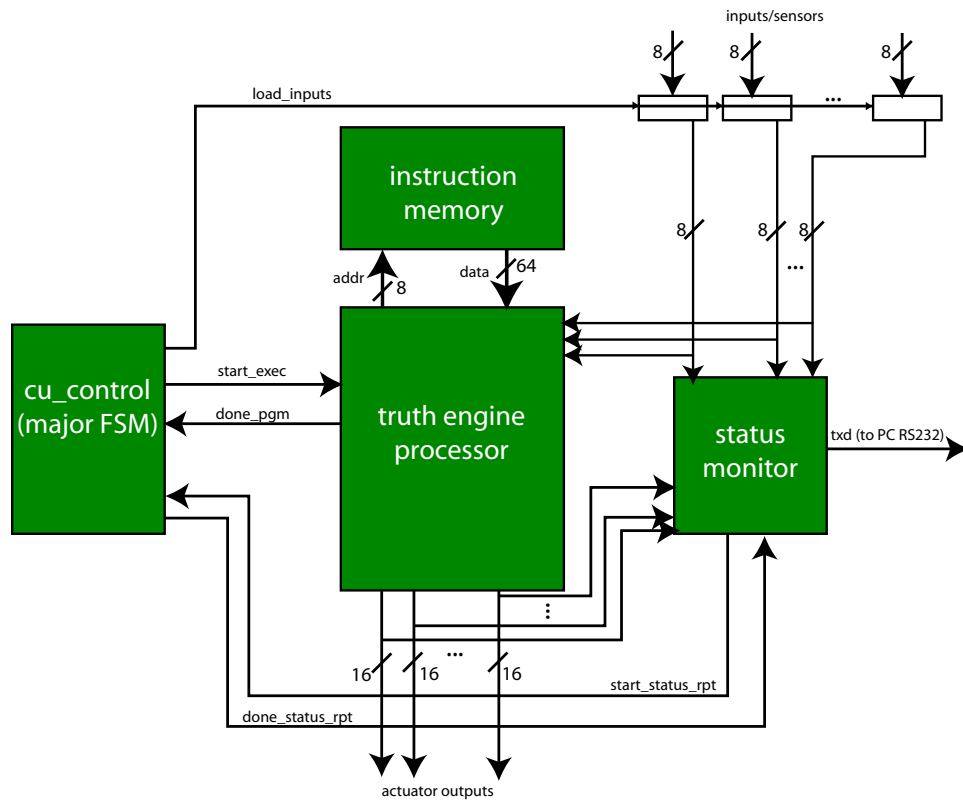


Fig. 15. Top-Level Control Unit Block Diagram

### 2.2.1 Major FSM. [Designed, Implemented, and Written by James Psota]

The control unit's major FSM can be seen in Figure 16. The system completes an entire loop of the control flow about 100 times a second, as dictated by an internal counter. The system waits in the INIT state until the .01 second pulse is asserted. Then, it enters into the LOAD\_INPUTS state. This state loads the inputs (*e.g.*, from the sensors) so that they stay constant throughout the entire current run of the program. It's important that the inputs remain constant throughout the program run for reasons that will be explained in Section 2.2.5. After the inputs are registered, the system enters the EXEC\_PGM state, which prompts the truth engine to begin processing the program stored in memory. When the program has completed execution, the system transitions into the DONE\_PGM state. Finally, the system enters the REPORT\_STATUS state, where it cycles through the inputs and outputs and reports their current state via the RS232 port of a host PC.

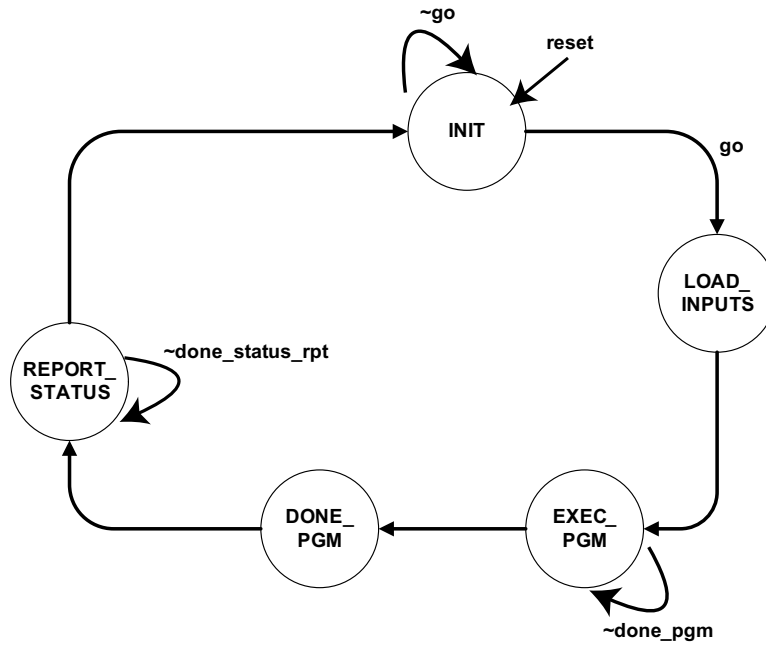


Fig. 16. Control Unit Major FSM

### 2.2.2 Instruction Format. *[Designed, Implemented, and Written by James Psota]*

The example shown in Figure 14 generally illustrated the types of actions and addressing the program needed to perform. Recall that the user needed to specify the following information: which operands to use, which operation to perform on the operands, which output value to set, which value to set the output value to. Additionally, the user needs to be able to specify if the operands inputs or immediates. The resulting 64-bit instruction format can be seen in Figure 17.

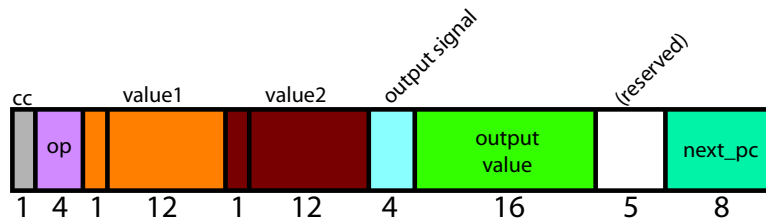


Fig. 17. 64-bit Instruction Format

The first field is called the cumulative condition field, which specifies if this instruction depends on the result of one or more previous instructions. This feature was added because each instruction can only specify a single operation, but there are times when the user may want to execute an action if *multiple* conditions are true. For instance, the user may want



to turn on the lights at 18:00 if it's Friday or Saturday, but not any other day. In this case, the user would specify the two conditions in separate instructions, specify a null action on the first instruction, and the desired action on the second instruction which has the cumulative condition bit set. This feature substantially increases the scope of the system's programmability.

The next four bits specify the opcode. The available opcodes can be seen in Table I. Note that the truth engine only executes boolean operations.

value	operation
0	less than
1	greater than
2	equal
3	not equal
4	less than or equal
5	greater than or equal
6	(reserved)
7	(reserved)
8	(reserved)
9	(reserved)
10	(reserved)
11	(reserved)
12	(reserved)
13	zero (nop)
14	one (always true)
15	done_pgm

Table I. Opcode Map

The next field specifies the first operand. The first bit specifies if the operand is an immediate, in which case the remaining 12-bit value field is all used as an immediate. If the top bit specifies that this operand is an input, the low four bits of the value field are used as an index into a multiplexor which selects the appropriate input. The next 13 bits are analogous to the first operand field, but specify the second operand field.

The next four bits specify which output should be asserted. The 16 subsequent bits specify the value the output should be set to if the condition is true. 16 bits of output value provide a wide range of options for the end user, such as sending IR signals or simply specifying a wide range of output values.

Finally, the last used field of the instruction is the next\_pc field. This field specifies the unconditional address of the instruction that should be executed after this one.

### 2.2.3 Program Format. *[Designed, Implemented, and Written by James Psota]*

The program is specified in the instruction memory, which is initialized by a memory initialization file. As in any computer system, the truth engine processor expects programs to be in a certain format. The processor will always start by executing the instruction at address 0, and unconditionally follow the next\_pc fields of each instruction. The program terminates when the processor sees a done\_pgm instruction. Thus, all programs must obey the following two constraints:

- (1) The first instruction must exist at PC=0
- (2) The last instruction must contain the done\_pgm instruction (opcode 15)

#### 2.2.4 Compiler. [Designed, Implemented, and Written by James Psota]

The HAS comes with a reasonably user-friendly interface that allows users to create programs in an intuitive way. The interface, which runs any host PC capable of executing a Perl script, steps the user through a series of simple questions. The user's input is passed to a compiler which in turn packages up 64-bit instruction words that implement the user's request. The resulting output of the program is a memory initialization file that is used to program the instruction memory. A screenshot of the compiler can be seen in Figure 18.

```
James.Psota@epsilon /cygdrive/d/MyDocuments/Courses/6111/final/src/hw/truth-engine/programs
$ ./progGen.pl 8 64
Enter output program name: [temp_program.nif]: myprog.nif
Do you want to create an instruction? [y]: y
The PC for this instruction will be 0
Does this instruction depend on the previous? [n]: y
Which operation [lt,gt,eq,neq,lte,gte,zero,one,done]? : eq
Is v1 a signal or imm [sig,imm]? [sig]: sig
Specify the value v1: 3
Is v2 a signal or imm [sig,imm]? [sig]: imm
Specify the value v2: 66
Specify which output signal should be activated: 3
Specify what the output value should be set to: 1
Specify the unconditional next PC [l]: 1
>> 0 : 100101000000000011000000100001000110000000000000010000000000001; % <dep> sig3 eq 66, set act3 -> 1; jump to 1 %
create another instruction? [y]: n
Done. Wrote program to myprog.nif.
```

Fig. 18. Compiler Interface

#### 2.2.5 Truth Engine Processor. [Designed, Implemented, and Written by James Psota]

The truth engine accounts for a large amount of the control unit's complexity. As mentioned above, it has the job of executing the program using sensor inputs as operands, and writing outputs to the actuator layer. Figure 19 shows the top-level datapath design.

The 64-bit instruction is read in from the instruction memory and placed into a register. The instruction completely dictates the program's operation – which operands to use, which operation to execute, which instruction to execute next, etc. The opcode field is passed into a decode module, which simply reports to the ALU which operation it should execute, and produces the done\_pgm signal if a done\_pgm opcode is seen. The value fields are used to either select which sensor input to choose, or to specify which value immediate to use. The values are passed to the ALU, which performs the appropriate operation. The result of the ALU operation (either a 1 or a 0) is used to enable loading of the appropriate actuator register. The output\_signal field is fed to a demultiplexor which asserts exactly one of 16 enable signals. This signal is another one of the few signals used to enable loading of the actuator registers. The top bit of the instruction along with the alu\_result is used to determine the value to write to the cumulative condition register, which keeps track of previous alu results so that multiple condition results are stored. This ultimately implements the cumulative condition feature described above, and is used in the enable logic that loads the actuator registers.

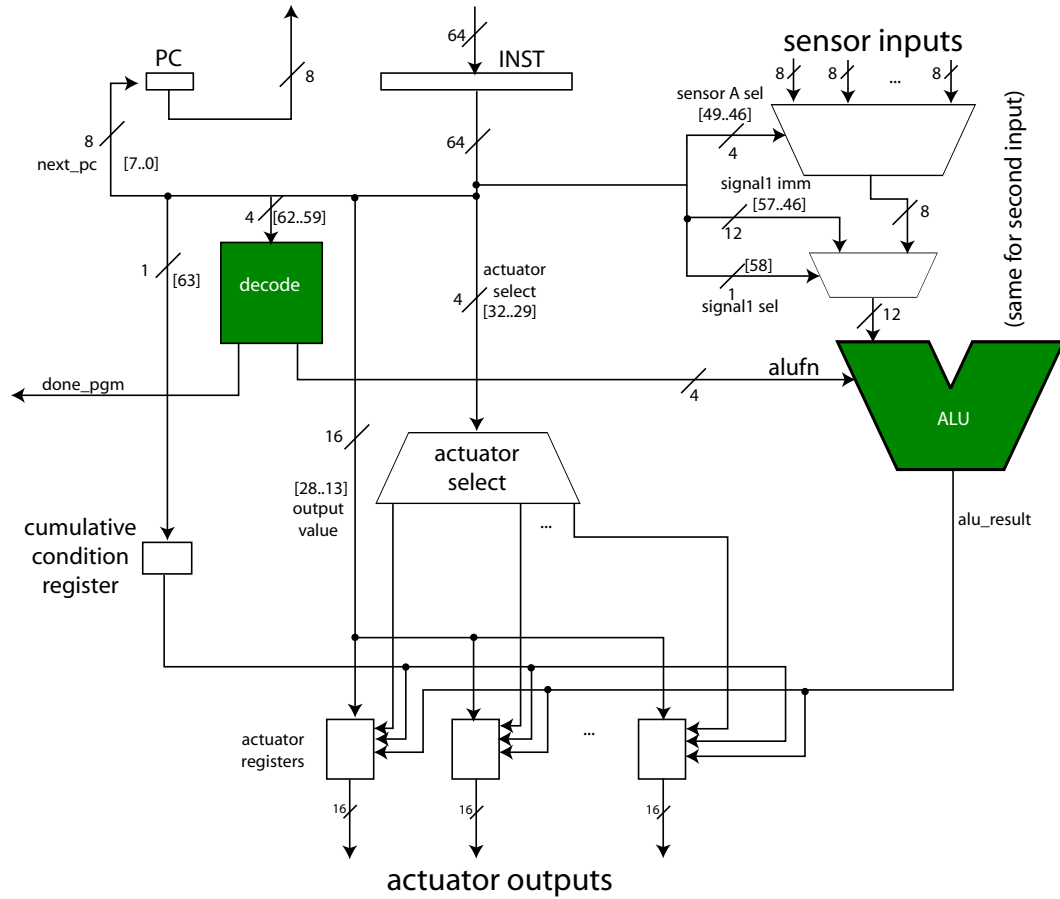


Fig. 19. Truth Engine Processor Block Diagram

### 2.2.6 Status Monitor: Hardware. [Designed, Implemented, and Written by James Psota]

The status monitor displays the current values of the system's inputs and outputs. This feature provides the user with a view of current conditions in the home environment, such as which lights are on and what the temperature is. It is also a useful tool in helping the user program and test the system. The system interacts with a host PC to display the status. The FPGA sends the data to the PC via the RS232 serial interface

As can be seen in Figure 16, the status is reported after the program has completed execution. Note that each sensor input is 1 byte wide, and each actuator output is 2 bytes wide. As the serial interface module<sup>1</sup> accepts 1 byte of data at a time, the status monitor has to cycle through the data, sending one byte after another. As can be seen in Figure 20, the values of each of the signals are sent first, and the values of the actuators are sent second. Note that the actuators' values are sent in two halves – the low order byte is sent first, and the high order byte is sent second.

<sup>1</sup>provided by HyungBin Son

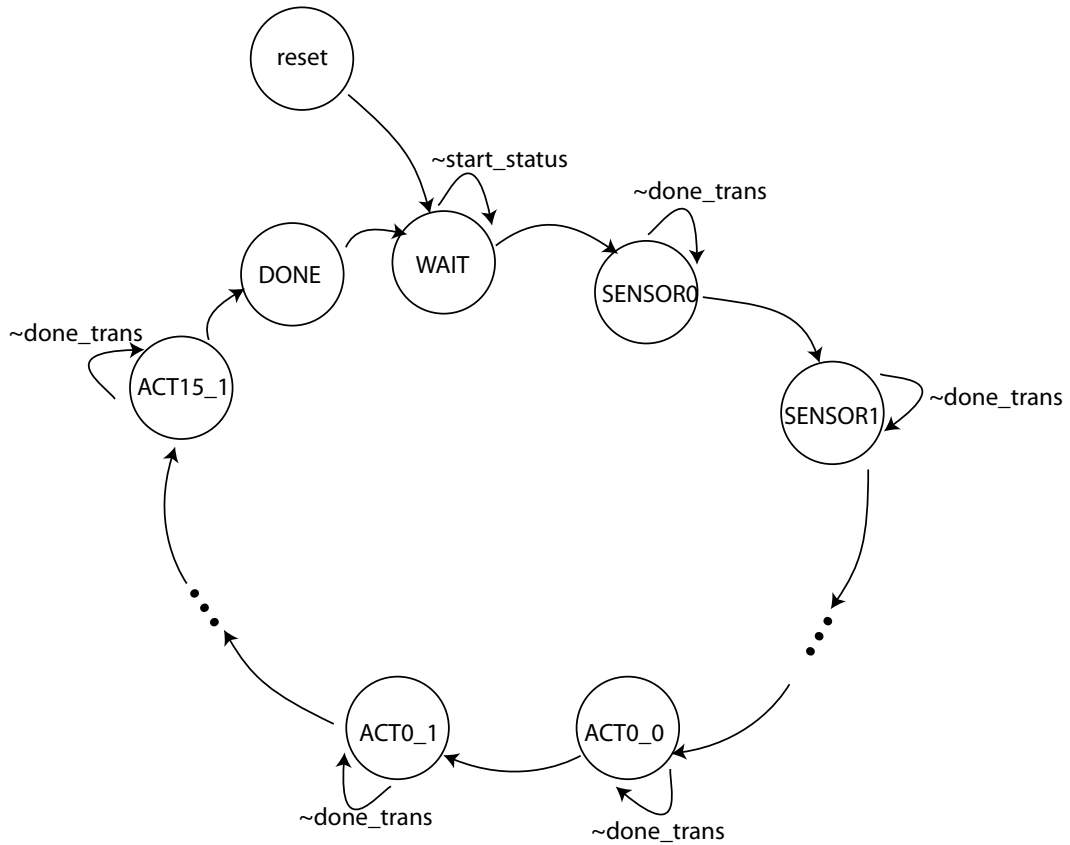


Fig. 20. Status Monitor FSM

### 2.2.7 Status Monitor: Software. [Designed, Implemented, and Written by Javier Castro]

The software of the status monitor was implemented using the Java Swing library and the javax.comm package. The application consists of a JFrame containing a JTextArea which is updated periodically (approximately once per second) via input over a serial cable connection (See Figure 21). The data transmitted to the application arrives in a prearranged manner. That is that the bytes are sent in an order which is specified to arrive in a predetermined order. The first 11 bytes to arrive are the values of the sensor inputs. Next are the 16 bytes from the eight actuators. The data is parsed by the status monitor program, and displayed on the screen in a format that is human friendly. For example, rather than a hexadecimal number, the user sees the temperature in degrees Fahrenheit. The state of lights throughout the home is decoded from 0, 1, 2, or 3 to off, dim, medium, or bright. By running this program on a Host PC, a user can check on the status of the system without having to run all over her home to see whether the sprinkler is on in the front yard, or to check if the lights in the bathroom are off.

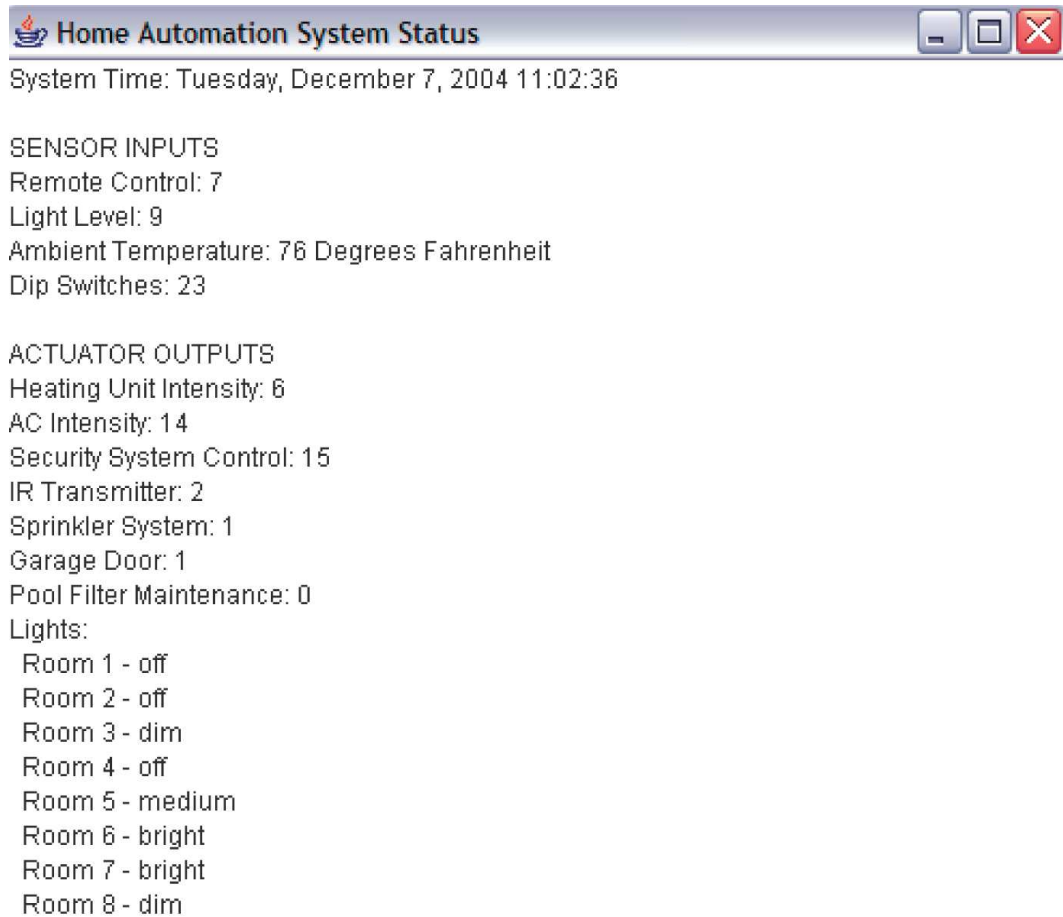


Fig. 21. Status Monitor Screen Shot

### 2.3 Actuator Layer

*[Written by Javier Castro]*

In order for the Home Automation System to perform its function and be useful to a home owner, it needs to interact with real world systems. Examples of such systems are lights, heating and A/C units, and sprinkler systems. A user needs to have the ability to tell the HAS that the sprinkler system should be turned on between 5:00am and 6:00am every Tuesday and Thursday. In order to realize this command, the Home Automation System has actuators which format output from the control layer to a form that household systems can understand. If a user of the home automation system has placed the system

into vacation mode, and the television is scheduled to turn on at 7:30pm, the control layer simply outputs the 6 bit power signal command to an infrared transmitter module that is connected to the system. The transmitter module then takes the address, 0x0C, and outputs the corresponding infrared signal that results in a TV actually turning on.

Due to a pin limitation on the FPGA's, there is a HAS status monitor, implemented in Java, that runs on a host computer. The state of the HAS's actuators is transmitted to the host computer via serial cable connection so that the data may be formatted and displayed on the monitor of the host computer.

### 3. TESTING

*[Written by Javier Castro and James Psota]*

Due to the modular nature of the design of the HAS, it was straight forward to test each module individually. In doing so, bugs were easy to spot, locate, and fix. Thorough simulation testing was performed on each module before being programmed onto an FPGA. Then, once the simulations ran according to specification, an FPGA was programmed with the architecture, and more component testing took place. The biggest problem in testing was due to faulty HEX displays and burned out pins on the lab kits.

#### 3.1 IR Module

The IR Module had a tricky bug which was hard to locate, due to successful simulations. The module simulated according to what was expected to occur given an input that timed out (did not adhere to RC-5 encoding), and did would successfully read a properly formed signal. The problem was, that if allowed to start up normally, it would enter a state where the internal reset signal to the timer was constantly asserted. This was found by progressively showing more and more signals in the simulator, and by starting the input signal which was received from the IR receiver a little bit later. The problem was solved by placing a level-to-pulse module before the reset signal to left the IR Decoding FSM so that the IR Timer would only receive a pulse, and therefore be allowed to start upon receiving a new start pulse. This solved the grid lock issue. In order to prove that the IR module was in fact decoding IR signals, the output from the IR Module was displayed on the HEX displays of the FPGA lab kits and simultaneously compared to an oscilloscope reading of the pins on the IR module. Since the address bits of the waveforms matched the output shown on the HEX displays, it was verified that the IR Module performed properly and according to its specification.

#### 3.2 Temperature Module

Some experimentation was involved in testing the Temperature Module. In order to ensure that the thermistor-voltage divider was working properly, the voltage across the non-variable resistor was measured using an oscilloscope. Once it was verified that the voltage was as expected, the FPGA was programmed with the digital portion of the Temperature Module. The AD670 was wired up with its controls from the FPGA, appropriate ground and power signals were supplied to its formatting pins, and the voltage from the voltage divider was fed to the A/D. Being wary of a load resistance across the leads of the AD670, the voltage was, again, measured with an oscilloscope. It was discovered at this point, that the load resistance of the AD670 was indeed something to take into consideration as it was on the same magnitude as the resistor that the voltage was being divided across. After

much thought, an LM741 Op Amp was placed in a voltage follower configuration, so that the voltage of the output would match that of the inputs. This idea worked, and the Op Amp was included as a permanent part of the Temperature Module. Once this issue had been addressed, physical testing of the digital part of the Temperature module was performed. This involved having the FPGA display the output from the Temperature Module on its HEX displays by assigning the output pins appropriately. Since a sample is taken every second, the display was steady and displayed the ambient temperature in hexadecimal. If the thermistor was heated up, the temperature would rise; if it was cooled down, tested by grounding the  $V^+$  lead, the temperature bottomed out to 50°F.

### 3.3 Light Module

Since most of the Light Module was implemented by reusing parts from the Temperature Module, most of the testing was completed when the Temperature module was created. All that was left to test, at this point, was that the conversion module performed adequately enough, for the purposes of the Truth Engine located in the control layer.

### 3.4 Control Layer

The control layer was thoroughly tested in simulation incrementally. A small subset of the control system was built first to work out details as the design was being finalized. For instance, the initial system only had one sensor input, one actuator output, one opcode, executing one instruction. After this minimal system was debugged, more and more features and components were added until the final desired system was realized.

After thorough simulation testing, the control layer logic was placed on an FPGA for field testing. The first test involved the execution of a one-instruction program. The program would take the data from one sensor as input, and output the appropriate result to the appropriate actuator. Next, more and more instructions were added to the simple program so that the full capabilities of the truth engine could be tested. At this point, actuators were asserted or not asserted by turning the LEDs of the FPGA lab kits on or off. Once a multi-instruction program could be executed, with dependence on several inputs, testing was complete and ready for integration.

### 3.5 Actuator Layer

Since the results of the actuator layer are output over a serial cable. By connecting this cable to a host computer, where a Java program resides, the information being sent over the cable could be displayed onto the monitor for simple debugging. One of the problems that occurred while implementing this process was that the information being sent over the serial cable, whenever the lab kit was initially turned on, was junk. The method discovered for initiating the HAS/Host Computer communication was to start up the HAS, place it into the reset state, and then initiate the program on the Host Computer. At this point, whenever the HAS was taken out of the reset state, information would be transmitted in an expected manner to the program running on the Host Computer, and as a result, displayed on the computer monitor in a decipherable manner. A second problem which was encountered, while interfacing with the Java program, was that the screen was flickering. This was the result of updating the screen too often. In order to solve this problem, the period at which data was sent over the serial cable was slowed down, and the amount of times that the Java program would update the screen was slowed down. After these changes, the monitor would display new data about once a second.

### 3.6 Integration Testing

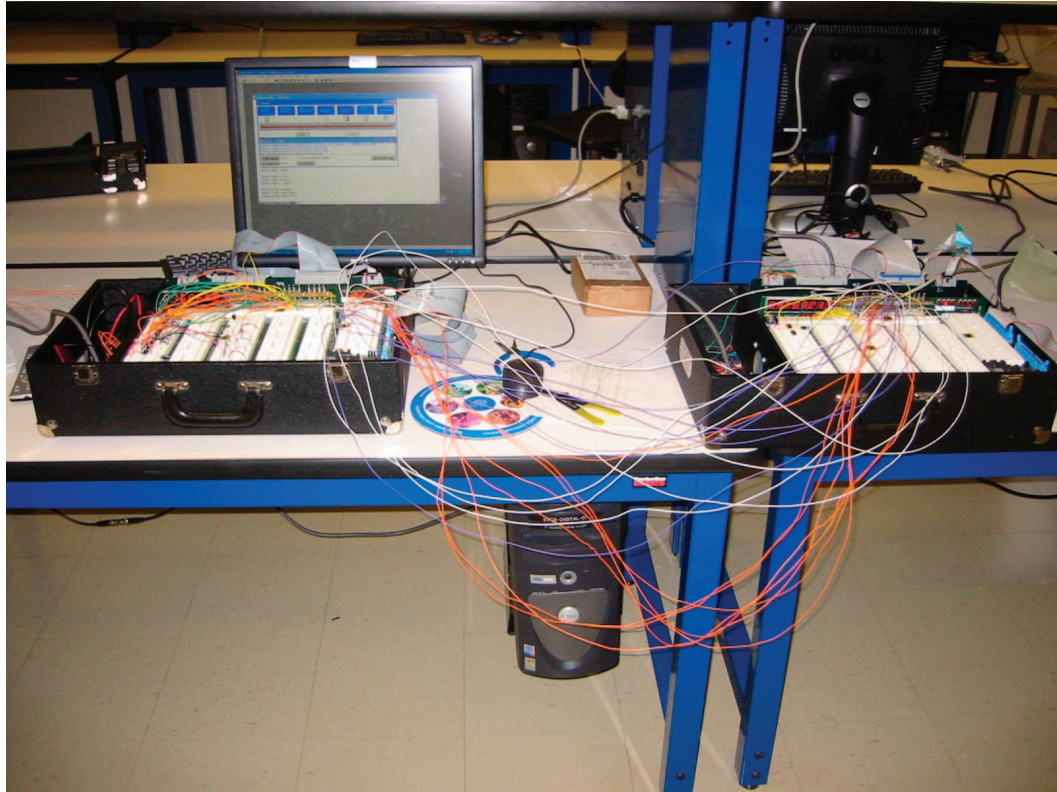


Fig. 22. Lab Kit Integration Testing Setup

Once the above modules were thoroughly testing both in hardware and software, the system was gradually assembled. Because the system's components were thoroughly tested, integrating the entire system proved to be much simpler than one would anticipate. When the two lab kits were connected as shown in Figure 22, the entire system simply worked on the first try. The sensor inputs showed up on the status monitor screen, and were updated in real-time as desired. Furthermore, relatively complicated programs were run and produced proper output.

## 4. CONCLUSION

*[Written by Javier Castro]*

By designing the Home Automation System in a modular manner, it was possible to create a system that was complete with sensors, running program, and status monitor in an incremental fashion. Building the system in this way made finding bugs early on an easy task to accomplish. After completing a group project such as the HAS, it becomes apparent that a strong emphasis on hierarchy and modularity early in the design process



results in a clear specification. This specification allows the implementers of the system to keep a clear idea of what tasks need to be accomplished and never get confused by the complexity of the task at hand. If, while constructing the system, any part failed to be realized, it was possible to work around the problem and still manage to implement a system that was working towards the final goal. The lesson learned is that more time spent designing results in less time wondering what went wrong with the implementation.

### Acknowledgements

The authors would like to sincerely thank the 6.111 course staff for being a tremendously supportive resource throughout the course of this project. In particular, many of Chris Terman's design suggestions ended up making their way into the final design. Likewise, HyungBin Son's RS232 interface and guidance throughout the project was invaluable, and Colin Weltin-Wu's support throughout the project was always helpful and is much appreciated.